Bridging the Semantic Gap between Abstract Models of Embedded Systems*

Jagadish Suryadevara, Eun-Young Kang, Cristina Seceleanu, and Paul Pettersson

Mälardalen Real-Time Research Centre,
Mälardalen University, Västerås, Sweden.
E-mail: {jagadish.suryadevara, eun.young.kang, cristina.seceleanu, paul.pettersson}@mdh.se.

Abstract. In the development of embedded software, modeling languages used within or across development phases e.g., requirements, specification, design, etc are based on different paradigms and an approach for relating these is needed. In this paper, we present a formal framework for relating specification and design models of embedded systems. We have chosen UML statemachines as specification models and ProCom component language for design models. While the specification is event-driven, the design is based on time triggering and data flow. To relate these abstractions, through the execution trajectories of corresponding models, formal semantics for both kinds of models and a set of inference rules are defined. The approach is applied on an autonomous truck case-study.

1 Introduction

Embedded systems (ES) are increasingly becoming control intensive, and time sensitive. To ensure predictable behaviors, the development phases of an ES require extensive modeling and analysis. These development phases/ abstraction layers e.g., requirements, specification, design, and implementation, provide opportunities for applying different predictability analysis techniques. Such models have to be precise enough to support formal analysis, and must ensure inter-operability during design. However, they may use paradigms for describing behavior that cannot be immediately compared and related, due to their apparently incompatible nature.

There exist several paradigms for behavior specification of embedded systems. For example, statemachine based approaches, such as UML statemachines [11], are intended to specify timed aspects of computation and communication, besides functionality. They often use an aperiodic, *event-triggered* representation of behavior, since such a paradigm facilitates easy changing of a model's configuration or set of events. On the other hand, behavior models might use a different

* This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS and Q-ImPrESS EU project.

modeling paradigm, e.g., a periodic, *time-triggered* behavioral description, instead of an event-triggered representation. With time-triggered communication, the data is read from a buffer, according to a triggering condition generated by, e.g., a periodic clock. Although these modeling capabilities are invaluable to obtaining a mature ES development process tailored for predictability, in order to ensure the correctness of the process, one needs to guarantee that the behavioral models are indeed consistent.

In this paper, we present a formal framework and a methodology for relating event-based and time triggered, data-flow driven models of behavior, which may be used at the same abstraction layer, e.g., at specification level, or across various layers of abstraction, from specification, to, e.g., the design level of embedded system development. Concretely, we consider UML statemachines [11] for event-based specification models and the ProCom component language [15] for design models. Hence, as it stands now, the framework is tailored to a specific class of embedded systems, which employ the above mentioned formalisms for modeling behavior. However, the framework and the methodology could be generalized to include other similar classes of systems (e.g., component based systems) and other behavioral paradigms (e.g., finite state machines).

The proposed framework is based on comparison of execution trajectories of corresponding behavior models. To accomplish this, the formal semantics of both kinds of models is defined in terms of underlying transition systems. As the execution trajectories generated by above described models can be extremely large and incomprehensible, they need to be reduced to more readable and analyzable forms. Hence, we propose two sets of inference rules, one for simplification of specification trajectories and other for simplification of design trajectories. Moreover, in order to be able to relate and compare the above two sets of simplified trajectories, we introduce a set of transformation rules that lets one relate an event-triggered trajectory with corresponding time-triggered one.

We apply our approach on an autonomous truck system, by comparing some trajectories of its specification with those of corresponding component-based design model. By virtually simulating the models, we show a "run" of each model, respectively, by outlining corresponding sets of representative trajectories. Then, we show that, by applying our rules, we can first simplify the design model trajectory and then transform it into a trajectory equivalent to the one generated by the specification model. The timing aspects of both runs are also apparent in the respective trajectories, hence we show how to relate them too. For creating the truck's design model, we use the development environment of SaveIDE [12], an integrated design environment for ES. SaveIDE is developed as part of the PROGRESS project [1] for component-based development of predictable ES in the vehicular domain. It supports the subset of ProCom modeling language used for the case study design of the paper.

The rest of the paper is organized as follows. In Section 2, we describe event-based, and time triggered formalisms for modeling embedded systems. Corresponding to these formalisms we formally define semantics of a subset of both UML statemachines and ProCom design languages. In Section 3, we present the

case study details. In Section 4, we describe our methodology, and introduce three sets of inference rules for simplification and comparison of trajectories of specification and design models. Some related work is discussed in Section 5. In section 6, we make conclusions and some aspects of the future work of the paper.

2 Abstract Models of Embedded Systems

In this section, we define the modeling formalisms for model-based specification and design of embedded systems used in this paper. As specification language, we will consider UML statemachine notation with timing annotations [11], and for design models, we will use the ProCom component modeling language [3].

2.1 Specification model of embedded systems

We specify embedded systems using the UML statemachine notation [11]. In order to model timing, we will use the notion of timeouts provided in UML. An example of a UML statemachine is shown in Fig. 1. We now give a formal definition of the model:

Definition 1 (Statemachine Syntax). *A statemachine is a tuple* $\langle L, l_0, A, E, M \rangle$ *where*

- L *is a finite set of locations,*
- $l_0 \in L$ *is the initial location,*
- $A = \{a_0, ..., a_n, tm\}$ *is a set of events, where*
  - a_i *is an external event with zero or more parameters,*
  - tm *is a timeout representing the expiry of a timer, and*
- $M : L \rightarrow \{\varepsilon\} \cup \mathbb{N}$ *is a mapping from locations to the natural numbers (including zero), or* ε *denoting absence of timeout,*
- $E \subseteq L \times A \times L$ *is a set of edges.* □

Fig. 1 shows a UML statemachine with the three locations Follow, Turn, and Find. The edges from Follow to Turn and from Find to Follow are labeled with the external events e_o_l() and line_found(), respectively. The edge from Turn to Find is labeled with event after(4), intuitively denoting a timeout that expires after four time units[1].

We now give the semantics of a UML statemachine specification model defined in terms of a finite state transition system.

Definition 2 (Statemachine Semantics). *The semantics of a statemachine is defined as a transition system* $\langle S, s_0, T \rangle$ *where*

- S *is a finite set of states of form* $\langle l, m \rangle$ *with* $l \in L$ *and* $m \in \{\varepsilon\} \cup \mathbb{N}$,
- $s_0 \in S$ *is the initial state* $\langle l_0, M(l_0) \rangle$,

[1] In the figures, we use timeout events of the form *after*(n), where $n \in \mathbb{N}$, instead of annotating the source location (e.g., location Turn in Fig. 1) with timeout value n.

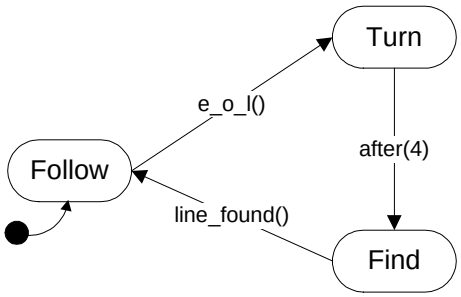

Fig. 1. A UML statemachine specification model of the autonomous truck.

– $T \subseteq S \times A \cup \{tick\} \times S$ *where tick is a periodic internal event, is a transition relation such that*
  - $\langle l, m\rangle \xrightarrow{a_i} \langle l', m'\rangle$ *if* $\langle l, a_i, l'\rangle \in E$, *and* $m' = M(l')$
  - $\langle l, m\rangle \xrightarrow{tick} \langle l', m'\rangle$ *if* $l = l'$, $m \neq 0$, *and* $m' = \begin{cases} \varepsilon & \text{if} \quad m = \varepsilon \\ m - 1 & \text{otherwise} \end{cases}$
  - $\langle l, m\rangle \xrightarrow{tm} \langle l', m'\rangle$ *if* $\langle l, tm, l'\rangle \in E$, $m = 0$, *and* $m' = M(l')$

□

Intuitively, the initial state represents the initial location, and its timeout value, in the statemachine. The first rule describes the state change when an external event specified over an edge from current location, and in the current state, occurs. By second rule, if a timeout is defined at current location, the current value of the timeout decreases in steps of one corresponding to each occurrence of an internal periodic *tick* event. The *tick* event is ignored in the current state if no timeout is associated with the corresponding location. The third rule describes the occurrence of timeout event, and hence the location and corresponding state change, when the timeout duration associated with the current location expires i.e. becomes zero.

A *trajectory* of a UML specification model is an infinite sequence

$$\tau = \langle l_0, m_0\rangle \xrightarrow{\lambda_0} \langle l_1, m_1\rangle \xrightarrow{\lambda_1} \langle l_2, m_2\rangle \ ...$$

where $\langle l_0, m_0\rangle$ is the initial state, and $\langle l_i, m_i\rangle \xrightarrow{\lambda_i} \langle l_{i+1}, m_{i+1}\rangle \in T$ and $\lambda_i \in \{a_0, ..., a_n, tick, tm\}$ for all $i \in \mathbb{N}$.

2.2 Design model of embedded systems

As design modeling language we will use ProCom [3], a component model for embedded systems. It consists of the two sub-languages: ProSys, which is designed to model systems at high level (i.e., in terms of large-grained components called *subsystems*), and ProSave [15] which is designed to model detailed functionality of the subsystems. In this paper, we will focus on the ProSave model as it is

better suited for our purposes. A ProSave model consists of atomic or composite components connected through ports (divided into input and output ports), and connections. Ports and connections represent data flow between components.

Definition 3 (Component Syntax). *A component C is a tuple $\langle I, O, P, in, out, f, e\rangle$, where*

- *I, O, and P are mutually disjoint sets of input, output, and private variables respectively,*
- *$in : I \rightarrow Bool$ is a boolean expression over input variables I that triggers the execution of the component,*
- *$out : O \rightarrow Bool$ is a boolean expression over output variables O that indicates that the component has completed its execution,*
- *$f : I \times P \rightarrow P \times O$ is a function that maps input and private values to the private and output values, and*
- *$e \in \mathbb{N}$ is a constant representing the execution time of the component.*

□

We denote by $X = I \cup O \cup P$ the set of all variables with size $|X| = |I|+|O|+|P|$. We will further use $C.n$ to denote the elements of a component, hence e.g., $C.I$ denotes the input variables of component C. We now introduce the formal syntax of the ProSave model.

Definition 4 (ProSave Syntax). *A ProSave design model is a tuple $\langle \mathbb{C}, \rightarrow\rangle$, where*

- *$\mathbb{C} = \{C_0, ..., C_n\}$ is a set of components,*
- *$\rightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is a set of component connections, such that output variables $C_i.O$ may be connected to input variables $C_j.I$* □

We will write $C_i.O_m \rightarrow C_j.I_n$ to represent the connection from output variable m of component C_i to input variable n of component C_j.

A ProSave system is typically driven by a periodic clock which periodically generates a control (or trigger) signal. A clock component is defined as follows:

Definition 5 (Clock Component). *A component $C = \langle I, O, P, in, out, f, e\rangle$ is a clock component with period p iff $\mid I \mid = \mid O \mid = 1$, $e = p$, and $C.O \rightarrow C.I$.*

□

Fig. 2 shows a ProSave design model consisting of seven components (depicted as boxes) interconnected by data and control flow connections (depicted as solid arrows indicating the flow direction). Component SystemClock is a clock component with period 40. The other six components have execution time 10. Their internal behavior may be specified using a formalism based on statecharts [14] or timed automata [2], which we are not explicitly concerned with in this paper. A component starts execution when it receives control input. It then reads its input and proceeds with internal computation. When the internal execution is completed, data and control output is generated for other components.

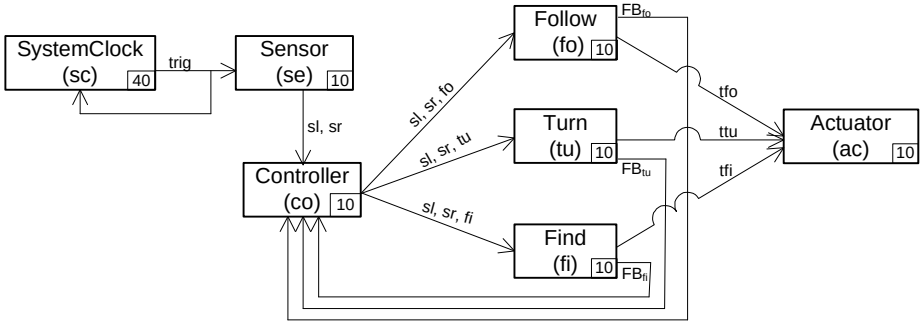

Fig. 2. Schematic view of a ProSave design model of the autonomous truck.

We will now give the formal semantics of the subset of ProSave used in this paper. For the semantics of the full ProCom language, we refer the reader to [15].

For a ProSave model consisting of components $C_0, ..., C_n$, we use V to denote the set of all variables in a model, i.e., $V = X_0 \cup ... \cup X_n$. The semantics is defined using valuations α mapping each variable in V to values in the type (or domain)[2] of V, and vectors $\bar{\beta}$ of $\beta_i \in \{0, ..., e_i, \perp\}$ representing the remaining execution time of all components C_i.

We use $f_i(\alpha)$ to denote the valuation α' in which $\alpha'(x_i)$ for each $x_i \in P_i \cup O_i$ is the value obtained by applying the function $C_i.f$ in the valuation α, and $\alpha'(x') = \alpha(x')$ for all other variables x'. To update the execution time vector $\bar{\beta}$ we use $\bar{\beta}[\beta_i := n]$ to denote the $\bar{\beta}'$ in which $\beta'_i = n$ and $\beta'_j = \beta_j$ for all $j \neq i$, and we write $\bar{\beta} \ominus n$ to denote the $\bar{\beta}'$ in which $\beta'_i := \beta_i - n$ for all $\beta_i \geq n$.

Definition 6 (ProSave Semantics). *The semantics of a ProSave design model* $\langle \{C_0, ..., C_n\}, \rightarrow \rangle$ *is defined as a transition system* $\langle \Sigma, \sigma_0, \mathcal{T} \rangle$ *where*

- Σ *is a set of states of the form of a pair* $\langle \alpha, \bar{\beta} \rangle$,
- $\sigma_0 \in \Sigma$ *is the initial state* $\langle \alpha_0, \bar{\beta}_0 \rangle$ *which is such that* $\alpha_0 \models C_i.in$ *for all clock components* C_i *and* $\alpha_0 \models \neg C_j.in$ *for all other component* C_j, *and* $\bar{\beta}_0 = \bar{\perp}$,
- $\mathcal{T} \subseteq \Sigma \times \{CD_i, CS_i, TP\} \times \Sigma$ *is a set of transitions such that the following conditions hold:*
  - *(component start)* $\langle \alpha, \bar{\beta} \rangle \xrightarrow{CS_i} \langle \alpha', \bar{\beta}' \rangle$ *if* $(C_i.in \wedge (\beta_i = \perp))$, $\bar{\beta}' = \bar{\beta}[\beta_i := e_i]$, *and for all* $i \neq j : \beta_j \neq 0$,
  - *(component done)* $\langle \alpha, \bar{\beta} \rangle \xrightarrow{CD_i} \langle \alpha', \bar{\beta}' \rangle$ *if* $\beta_i = 0$, $\alpha' = f_i(\alpha)$, *and* $\bar{\beta}' = \bar{\beta}[\beta_i := \perp]$,
  - *(time passing)* $\langle \alpha, \bar{\beta} \rangle \xrightarrow{TP} \langle \alpha', \bar{\beta}' \rangle$ *if for all* $i : \neg(C_i.in \wedge (\beta_i = \perp))$ *and* $\beta_i \neq 0$, $\bar{\beta}' = \bar{\beta} \ominus 1$, *and* $(\alpha' = \alpha)$.

where $CS_i \in \{CS_0, ..., CS_n\}$ *and* $CD_i \in \{CD_0, ..., CD_n\}$. □

[2] We assume all variables in V are of type Boolean or finite domained integers.

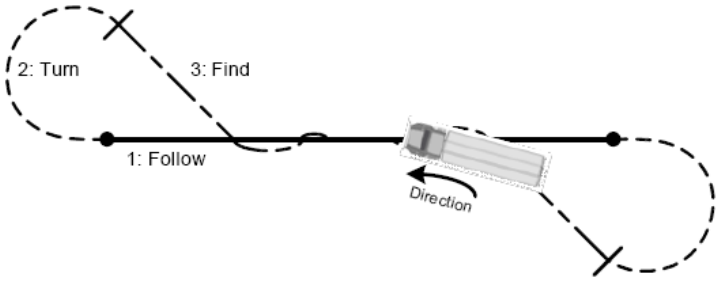

Fig. 3. Path of the truck movement.

Intuitively, in the initial state only the clock components are triggered and the remaining execution time of all components are undefined. The "component start" rule describes how components are started. A component C_i may start its execution provided that all completed components have written their output. When C_i starts, its execution time is set to e_i. The "component done" rule describes that when a component C_i completes its execution, its output values are generated and mapped to the input values of the connected components according to connection relation $\rightarrow$, and its remaining execution time is updated to $\perp$ to reflect that it is inactive. Rule "time passing" describes how time progresses in the design model. As time progresses the remaining execution time β_i of each active component C_i is decremented by 1.

A *trajectory* of a design model is an infinite sequence

$$\pi = \langle \alpha_0, \beta_0 \rangle \xrightarrow{\gamma_0} \langle \alpha_1, \beta_1 \rangle \xrightarrow{\gamma_1} \langle \alpha_2, \beta_2 \rangle \ ...$$

where $\langle \alpha_0, \beta_0 \rangle \in \sigma_0$ is an initial state, and $\langle \alpha_i, \beta_i \rangle \xrightarrow{a_i} \langle \alpha_{i+1}, \beta_{i+1} \rangle \in \mathcal{T}$ is a transition such that $\gamma_i \in \{CD_i, CS_i, TP\}$ for all $i \in \mathbb{N}$.

3 Case Study: Autonomous Truck

The autonomous truck is part of a demonstrator project conducted at the Progress research centre[3]. The truck moves along a specified path (as illustrated in Fig. 3), according to a specified application behavior. In this section we give an overview of the truck application followed by a specification, and a design model, described in the modeling languages introduced in the previous section.

We will study a simplified version of the case study, in which the truck should simply follow a line. When it reaches the end of the line, it should try to find back to the line, follow the line again in the opposite direction, and repeat its behavior. The truck will have the following operational modes (see also Fig. 1):

[3] For more information about Progress, see http://www.mrtc.mdh.se/progress/.

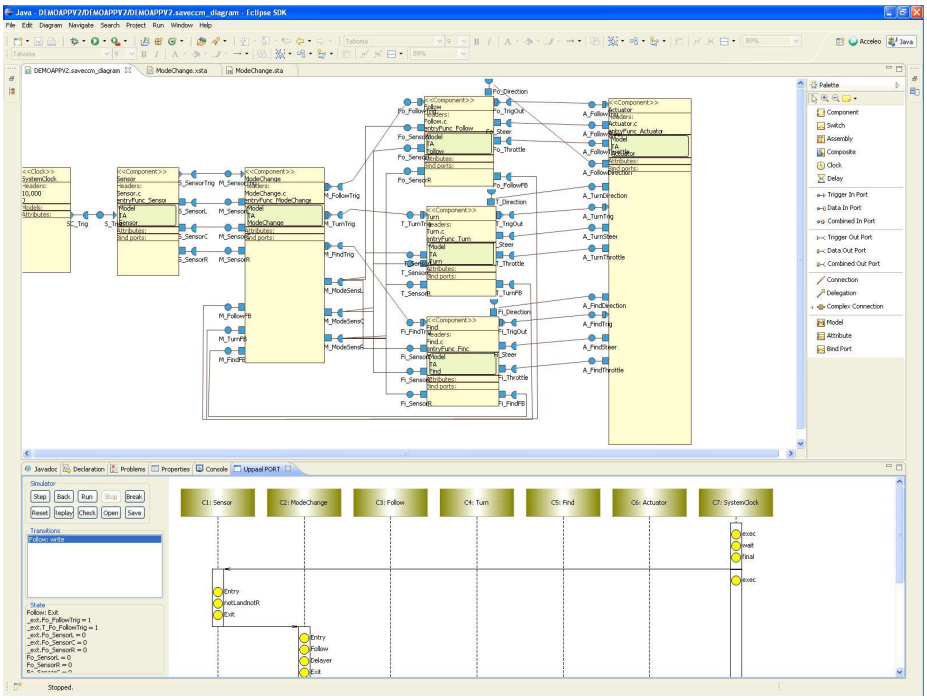

Fig. 4. The design model of the autonomous truck in SaveIDE.

- *Follow*: in which the truck follows the line (the thick line of Fig. 3) using light sensors. When the end of the line is detected, it changes to *Turn* mode.
- *Turn*: the truck turns right for a specified time duration, and then changes to *Find* mode.
- *Find*: the truck searches for the line. When it is found, the truck returns to *Follow* mode.

A specification model of the case study is given in Fig. 1. It starts in location Follow. The end of the line is modeled using external event e_o_l(). In location Turn, it turns for four seconds, and then proceeds to location Find when the timer expires. The external event line_found() models that the line is found and control switches back to the initial location Follow.

The schematic view of a ProSave deign model of the case study is given in Fig. 2. The original model (as shown in Fig. 4) was developed using SaveIDE [12], an integrated development environment supporting the subset of ProSave used in this paper. As shown in Fig. 2, the design model consists of components SystemClock (a periodic clock), Sensor, Controller, Follow, Turn, Find and Actuator. Component SystemClock triggers the complete model periodically through the component Sensor which reads the light sensors of the truck. The sensor values (left, right) are communicated through the data ports sl and sr. Note, a connection between two components as shown in Fig. 2, denotes a collection

of independent port connections between corresponding data or trigger ports of the components. Component Controller acts as a control switch for triggering the components Follow, Turn, and Find selectively , through control ports fo, tu, fi respectively, which contain the functionality of the corresponding modes of the truck behavior. The completion of execution of each operational mode (the corresponding component) is indicated by data (port) values FB_{fo}, FB_{tu}, and FB_{fi} respectively. Component Actuator, triggered by control port tfo, ttu, or tfi, actuates the corresponding hardware to cause the physical activity of the truck movement. As discussed previously, the periodicity of the SystemClock is 40 time units and the execution times of each of other components is 10 time units.

4 Methodology Description

In Section 2, we have described the syntax and semantics of two models used in the development of embedded system software: the event-based model of UML statemachines, and the time-triggered and data-flow oriented model of ProCom. These are examples of modeling languages that are aimed at providing different views of embedded systems, used in different stages or at different abstraction levels during system development. The common use of different models creates a need for comparing descriptions of systems made in different modeling languages.

In this section, we propose a method for comparing event-based and time-triggered models of embedded systems. The method will be described and illustrated on UML statemachines and ProCom models of the autonomous truck case study described in the previous section. Constructing a semantic bridge between the two models requires a series of steps that need to be systematically applied. Our methodology for bridging the gap between the paradigms consists of the following five steps: (*i*) given a specification trajectory, generate a corresponding design trajectory by e.g, simulating the model; (*ii*) simplify the specification trajectory (can be omitted); (*iii*) simplify the design trajectory; (*iv*) transform the design trajectory into one comparable to the event-based specification trajectory; (*v*) compare the reduced specification and design trajectories.

To support above described steps (*ii*) to (*iv*) of the method we will present in Sections 4.1 to 4.3 a number of inference rules for simplifying specification and design trajectories, and for transforming between the two. In the latter transformation step, we need to take two crucial steps. One is to relate events in the UML statemachine model to the data-flow of the ProCom model. This is done by mapping events observed in the specification trajectories to predicates over the data variables used in the design model. We expect that a designer will easily be able to provide this mapping based on his insights and knowledge in the models. For the autonomous truck system, we can assume a mapping given in Table 4.2 in section 4.2. A second important step in relating two models of embedded systems regards the different time scales that may be used. We take a rather straightforward approach and assume a δ, as defined in section 4.3, for characterizing the sampling period in design models, in comparison to the time base used in the specification model.

4.1 Specification simplification inference rules

In the following rules, we denote by $s_i \in S, i \in \mathbb{N}$, the states of an arbitrary specification model trajectory.

Skip time rule. This rule states that a sequence of *tick* transitions corresponding to a location without an associated timeout can be ignored.

$$\frac{s_i \xrightarrow{tick} s_i \xrightarrow{tick} \quad \dots \quad \xrightarrow{tick} s_i}{s_i} \tag{skip}$$

By applying this rule to the original specification trajectory of the Autonomous truck (omitted due to space limitations), we get the simplified trajectory shown in Fig. 5.(a).

Time passing rule. The intuition behind this rule is that one can collapse a sequence of *tick* transitions corresponding to a timeout location in the specification model, into a single transition that collects all the ticks. Consequently, the intermediate states generated by the individual ticks become hidden.

$$\frac{s_i \xrightarrow{tick} s_{i+1} \xrightarrow{tick} \dots \xrightarrow{tick} s_{i+n}}{s_i \xrightarrow{n.tick} s_{i+n}} \tag{n\_tick}$$

To show the rule at work, we have used it to reduce the sequence of tick transitions (s_1 to s_5) displayed in Fig. 5.(a), to the corresponding sequence in Fig. 5.(b).

Timeout start rule. Here, we introduce the virtual event *tm_start* needed to distinguish the transition leading to the corresponding timeout annotated location, from the one fired when the timeout countdown starts. Although not a simplification rule by itself, its usefulness is shown in the rules skip and n_TP, presented later.

$$\frac{s_i \xrightarrow{event\_label} s_{i+1} \quad m = value \quad m' \neq \varepsilon \wedge m' \neq 0}{s_i \xrightarrow{event\_label} s_{i+1} \xrightarrow{tm\_start} s_{i+2}} \tag{tm\_start}$$

In the above rule, $value \in \{0, \varepsilon\}$. In case $value = 0$, that is, $m = 0$, it follows that $event\_label = tm$; on the other hand, if $value = \varepsilon$, that is, $m = \epsilon$, then $event\_label = a$.

Timeout rule. A sequence of *n-tick* transitions beginning at a location having timeout n that is then followed by a timeout transition can be reduced to a single transition denoted by $tm(n)$, as shown below:

$$\frac{s_i \xrightarrow{n.tick} s_{i+1} \xrightarrow{tm} s_{i+2}}{s_i \xrightarrow{tm(n)} s_{i+2}} \tag{tm}$$

After applying the timeout rule, the sequence of the *4-tick* transitions (s_1 to s_5) followed by the *tm* transition (s_5 to s_6), depicted in Fig. 5.(b), is reduced to transition (s_1 to s_6), as in Fig. 5.(c).

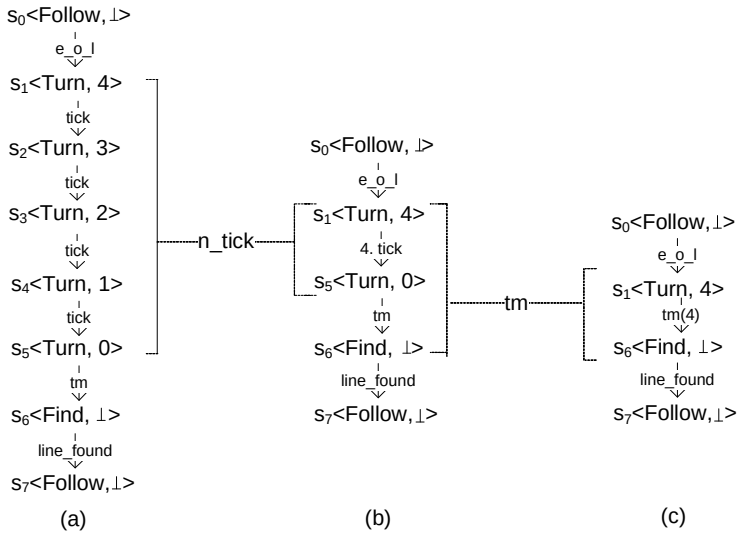

Fig. 5. Examples of specification trajectories simplifications of the autonomous truck.

4.2 Design simplification inference rules

As already mentioned, in order to be able to relate the specification and design models formally, we require the detailed mapping of the external and timeout events of the specification model onto predicates over data values of the corresponding design model. In addition to the observable events, such mapping should also include the virtual timeout start event, tm_start. We assume that such a mapping is provided by the ProSave designer, as he/she "implements" the specification model. For the current design model of the autonomous truck, one such mapping is given in Table 4.2.

Events	**Predicates**
e_0_l	$sl \wedge sr \wedge FB_{fo}$
$line\_found$	$(sl \vee sr) \wedge FB_{fi}$
tm(timeout event)	FB_{tu}
tm_start	ttu

Table 5. Events and corresponding predicates of the autonomous truck models.

Below, we denote by $\sigma_i \in \Sigma, i \in \mathbb{N}$, the states of an arbitrary design model trajectory. In Fig. 6, we give an excerpt of a design trajectory of the autonomous truck, and, on the right-hand side of the figure, we explain the used notation in terms of Definition 6 of Section 2.

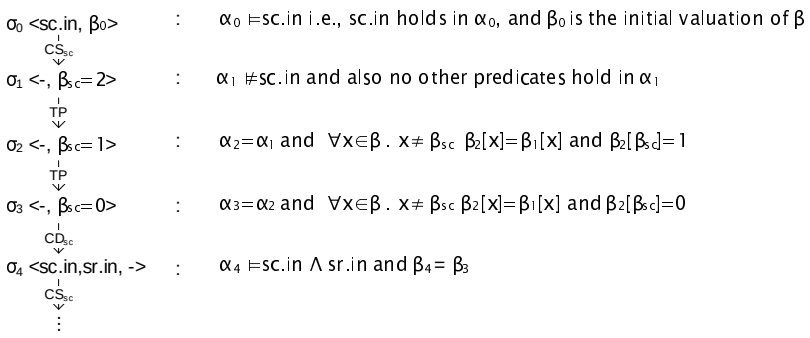

Fig. 6. Interpretation of example design trajectory notation w.r.t. Definition 6.

Skip time rule. This rule states that a sequence of TP-transitions from states that do not satisfy the predicate corresponding to the virtual event **tm_start** can be ignored. Such transitions correspond to time passing in the design trajectory, which are of no interest, that is, not related to observable timeout events.

$$\frac{\sigma_i \xrightarrow{TP} \sigma_{i+1} \quad \sigma_i \not\models Pred_{tm\_start}}{\sigma_i} \qquad \text{(Skip)}$$

We apply the skip time rule on a design trajectory of the autonomous truck (see Fig. 7), and, as a result, we simplify the trajectory by reducing states $\sigma_1, \sigma_2, \sigma_3$, and σ_4, to state σ_1 only. The complete trajectory is given in the Appendix.

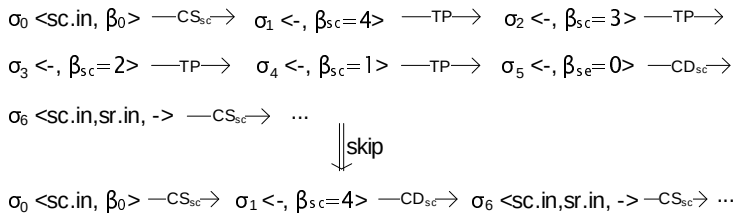

Fig. 7. Application of *skip rule* on a design trajectory of the autonomous truck model.

Hide CS rule. By this rule, a CS-transition, hence the target state, can always be ignored.

$$\frac{\sigma_i \xrightarrow{CS_i} \sigma_{i+1}}{\sigma_i} \qquad \text{(hide\_CS)}$$

Assuming a design trajectory of our case-study, the application of the above rule on this trajectory is shown in Fig. 8.

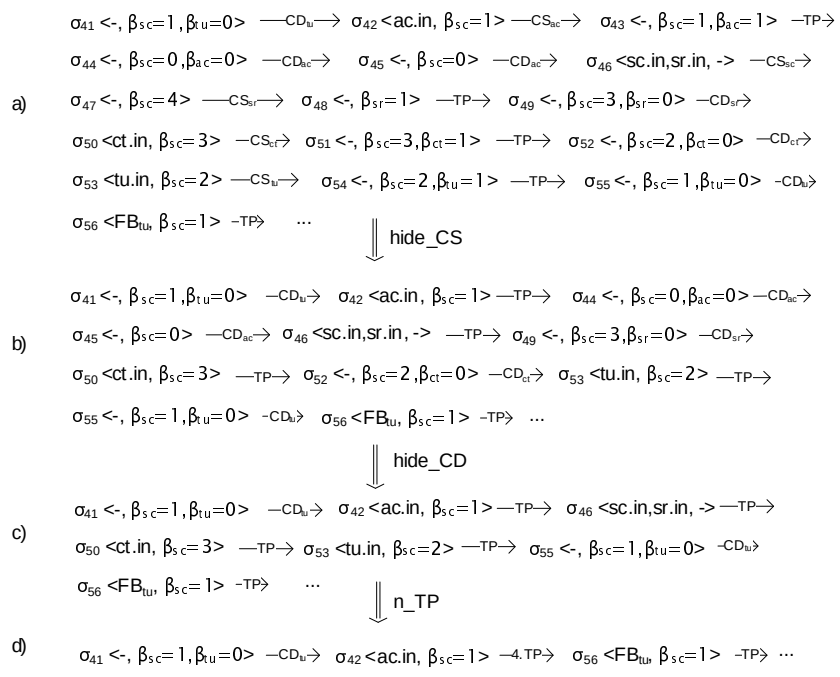

Fig. 8. (a) a partial design trajectory of the autonomous truck, and (b) to (d) corresponding reduced trajectories after application of the inference rules of Section 4.2.

Hide CD rule. This rule stipulates that a CD-transition and the corresponding source state can be ignored if the target state does not satisfy any event occurrence predicate.

$$\frac{\sigma_i \xrightarrow{CD_i} \sigma_{i+1} \quad \forall a \in A \cdot \sigma_i \not\models Pred_a}{\sigma_{i+1}} \qquad \text{(hide\_CD)}$$

An example application of the above rule is given in Fig. 8.

Time passing rule. A sequence of TP transitions starting in a state satisfying the predicate corresponding to *tm_start*, and ending in a state where the corresponding timeout occurs, can be collected into a single transition, while the intermediate states are ignored.

$$\frac{\sigma_i \xrightarrow{TP} \sigma_{i+1} \ldots \xrightarrow{TP} \sigma_{i+n} \xrightarrow{CD_j} \sigma_{i+n+1} \quad \sigma_i \models Pred_{tm\_start} \quad \sigma_{i+n+1} \models Pred_{tm}}{\sigma_i \xrightarrow{n.TP} \sigma_{i+n+1}} \qquad \text{(n\_TP)}$$

We have applied the above rule on a design trajectory of our Autonomous Truck, in Fig. 8. The rule works on the states $\sigma_{42}, \sigma_{46}, \sigma_{50}, \sigma_{53}, \sigma_{55}$ and σ_{56}.

Precedence of inference rules. In order to get the correct simplified design trajectory, we assume the following precedence rule when applying the above inference rules over design trajectories (rule hide_CS binds the strongest):

hide_CS **precedes** hide_CD **precedes** n_TP **precedes** Skip

4.3 Rules for transforming the design model trajectories

The following rules let one obtain design trajectories that are comparable to the event-based specification model trajectories. The first rule focuses on relating the time scales in both models; in order to achieve the goal, we assume a fixed quanta of time (number of time units), called δ, which can be viewed as the minimum amount of time guaranteed to be free of events. Then, this smallest amount of time becomes the basic time-unit that all time-related elements in both trajectories can be expressed by.

TimeOut Rule. We assume that a *TP*-transition "consumes" δ time units, the time duration associated with a *tick* event is *(m* δ)*$(m \in \mathbb{N})$, and an '*n*' time units timeout in a specification trajectory, *tm(n)*, equals (*n* tick*). Then, it follows that an *n.m.TP* transition in the design trajectory is equivalent to the '*n*' timeout event, *tm(n)*:

$$\frac{\exists \quad \sigma_k, \sigma_{k+1} \quad . \quad \sigma_i \xrightarrow{n.m.TP} \sigma_{i+1}}{\exists \quad s_k, s_{k+1} \quad . \quad s_k \xrightarrow{tm(n)} s_{k+1}} \qquad \text{(TO)}$$

EventOccur Rule. An event occurrence in a specification trajectory corresponds to a CD-transition in the design trajectory, such that the predicate associated to the event holds in the target state of the design trajectory.

$$\frac{\exists \quad \sigma_i, \sigma_{i+1} \quad . \quad \sigma_i \xrightarrow{CD_j} \sigma_{i+1} \quad \sigma_{i+1} \models Pred_a}{\exists \quad s_k, s_{k+1} \quad . \quad s_k \xrightarrow{a} s_{k+1}} \qquad \text{(EO)}$$

Next, we apply the above rules on a (simplified) design trajectory of our example, in order to obtain a trajectory comparable to the corresponding specification trajectory.

4.4 Applying the methodology

Here, we show our methods at work, on the Autonomous Truck case study, presented in Section 3. We do this by transforming a trajectory of the design model (Fig. 2), which we present in the Appendix, into one that is comparable to the corresponding specification model (Fig. 1) trajectory. First, the design trajectory is simplified by applying the inference rules introduced in section 4.1. Similarly, a trajectory of the specification model is then simplified to a minimal form by applying inference rules in 4.2. Both simplified trajectories are shown in Fig. 9. Next, we relate these trajectories by using the inference rules of transformation (section 4.3), as follows:

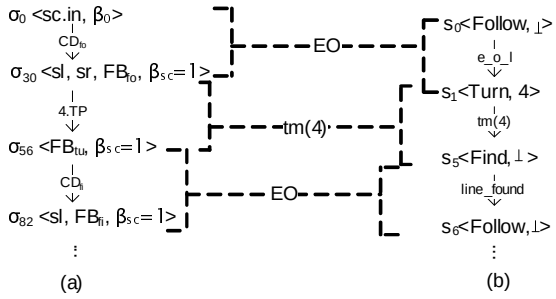

Fig. 9. Comparison of completely reduced trajectories of (a) the design model, and (b) the specification model, of the autonomous truck.

- by EO rule, the CD_{fo}-transition to state σ_{30} corresponds to the occurrence of event e_o_l, since the (e_o_l) predicate, that is, $FB_{fo} \wedge sl \wedge sr$, holds in the target state σ_{30}. Further, σ_{30} corresponds to the completion of the **Follow** mode of the truck behavior, as FB_{fo} holds (by design).
- similarly, by EO rule, the CD_{fi}-transition to state σ_{82} corresponds to the occurrence of event *line_found*, since $Pred_{line\_found}$, that is, $FB_{fi} \wedge sl$, holds in the target state σ_{82}. Further, σ_{82} corresponds to the completion of the **Find** mode of the truck behavior, as FB_{fi} holds (by design).
- by TO rule, the timeout event, $tm(4)$, between specification states s_2 and s_6 corresponds to the $4.TP$-transition between design states σ_{30} and σ_{56} that satisfy $Pred_{tm\_start}$, and $Pred_{tm}$, respectively. Further, σ_{56} corresponds to the completion of the **Turn** mode, as FB_{tu} holds (by design).

By applying the rules on the truck example, we have shown that, at least with respect to this example, it is possible to transform and compare a simplified design model trajectory of the Autonomous Truck with a simplified specification model trajectory. The transformation correlates also the time scales in both models. In this particular case, we have reduced the design model trajectory to an event-based trajectory identical to the specification one.

The above steps are necessary in proving the correctness of design with respect to specification, however they are not sufficient. To get closure, one has to first consider all possible design behaviors for transformation, and then possibly apply refinement techniques to prove that the design does implement the functional and timing requirements represented by the specification model (see Section 6).

5 Related Work

The problem of relating design to specification models is a topic with a growing interest in the research community. For synthesizing executable programs from timed models, a timed automata [2] based semantic framework, relying

on non-instant observability of events is proposed [6]. Time-triggered automata (TTA) - a sub class of timed automata (TA) - are used to model finite state implementations of a controller that services the request patterns specified by a TA. This technique enables deciding whether a TTA correctly implements a TA specification. In comparison, although ProCom oriented, our methodology can be applied within a generic component-based framework, and is not being tied to any particular formal verification framework either.

Sifakis et al. propose a methodology for relating the abstractions of both real-time application software and corresponding implementation [13]. The related formal modeling framework integrates event-driven, and time triggered paradigms by defining *untiming* functions. Problems of correctness, timing analysis, and synthesis are considered in the methodology. In contrast to our approach, this one does not address the intermediate design layer commonly used in system development.

In recent years, component and architecture based developments have been recognized as a viable way of building software systems [5]. Plasil and Visnovsky describe a formal framework based on *behavior protocols*, in order to formally specify the interplay between components [10]. This allows for formal reasoning about the correctness of the specification refinement and about the correctness of an implementation, in terms of the specification. Further, the framework is validated in the SOFA component model environment [4]. While the approach provides much needed formal correctness in component-based development, it does not address timing issues and vertical layers of abstractions in real-time system development.

UML has emerged as an industrial standard notation in system development and provides various sub-languages namely statemachines, sequence diagrams, etc [11]. For specification and design of real-time systems, a sub-language called UML/MARTE has been proposed [9]. In [7], the expressiveness of MARTE for event-triggered, and time-triggered communication is described. MARTE-based approaches facilitate various analytical methods for analysis, e.g., schedulability, system performance analysis; however, it falls short in providing formal support for comparing models at different abstraction levels.

Egyed, A. et al. [8] develop a methodology to mainly bridge the information gap created by heterogeneous models across the software life-cycle by transforming architecture description into (high-level) UML designs. The latter are then further refined into lower level designs. In contrast to our approach, their work does not provide details on the behavioral transformations. Indeed, a formal approach for establishing the semantic links between different terminologies and concepts across an architectural and a number of design models is not sufficiently addressed during the transformation.

6 Conclusions and Future work

In this paper, we have presented a formal approach for relating system models used in different design stages of embedded systems. For the early specification

phases, we chose the UML statemachine language in which system behaviors are described in terms of abstract states, event triggered state changes, and timeouts relating to the external system and timing behavior. For the later design stages, we use the ProCom component design model in which systems are specified using data-flow connectors and time-triggered component behaviors closer to the timing granularity and behavior exhibited on the target platform.

As a main result, we have described a formal way of comparing behavioral models of a system modeled in the two different languages. The solution is based on a set of inference rules that can be applied to gradually transform trajectories of a ProCom design model into a trajectory of a UML specification model. This enables a designer to make sure that a component-based and time-triggered ProCom design model implements the behavior of a more abstract and event-triggered UML specification of the same system.

Our initial experiences from applying the proposed technique to a truck control system, indicates that the design model trajectories can often be manually transformed into trajectories of the specification model. However, as this is not the case in general, we plan as future work to apply simulation relation checking to the specification trajectories, to prove (or disprove) conformance between non-identical trajectories. We will apply proof assistant tools to support these techniques.

References

1. Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkansson, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE approach to component-based development of vehicular systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.
2. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
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, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.
5. Ivica Crnkovic and Magnus Larssom. Challenges of component-based development. *J. Syst. Softw.*, 61(3):201–212, 2002.
6. Pavel Krčál, Leonid Mokrushin, P.S. Thiagarajan, and Wang Yi. Timed vs time triggered automata. In Philippa Gardner and Nobuko Yoshida, editors, *Proc. of CONCUR'04.*, number 3170 in Lecture Notes in Computer Science, pages 340–354. Springer–Verlag, 2004.
7. Frédéric Mallet, Robert de Simone, and Laurent Rioux. Event-triggered vs. time-triggered communications with UML Marte. In *FDL*, pages 154–159, 2008.
8. Nenad Medvidovic, Paul Grünbacher, Alexander Egyed, and Barry W. Boehm. Bridging models across the software lifecycle. *J. Syst. Softw.*, 68(3):199–215, 2003.

9. OMG. Unified modeling language (uml) profile for modeling and analysis of real-time and embedded systems (marte). In *Document ptc/07-08-04*. OMG, 2007.
10. Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Eng.*, 28(11):1056–1076, 2002.
11. James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.
12. Sverine Sentilles, Anders Pettersson, Dag Nyström, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-IDE - a tool for design, analysis and implementation of component-based embedded systems. In *Proceedings of the Research Demo Track of the 31st International Conference on Software Engineering (ICSE)*, May 2009.
13. Joseph Sifakis, Stavros Tripakis, and Sergio Yovine. Building models of real-time systems from application software. In *In Proceedings of the IEEE Special issue on modeling and design of embedded*, pages 100–111. IEEE, 2003.
14. Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, and Paul Pettersson. Analyzing a pattern-based model of a real-time turntable system. In Barbora Zimmerova Jens Happe, editor, *6th International Workshop on Formal Engineering approaches to Software Components and Architectures(FESCA), ETAPS'09, York, UK, March*, pages 161–178. Electronic Notes in Theoretical Computer Science (ENTCS), Vol 253, Elsevier, September 2009.
15. Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal semantics of the procom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, August 2009.

APPENDIX

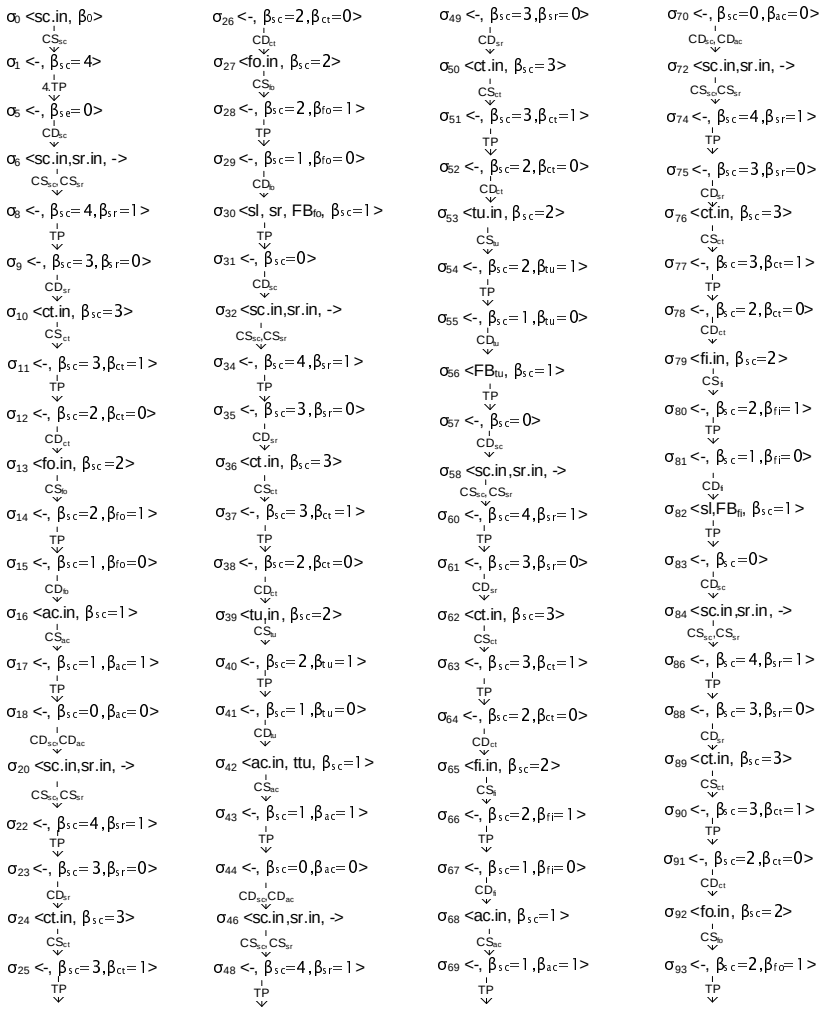

Fig. 10. An execution trajectory of the design model of the autonomous truck.