Modeling and Verification of Distributed Real-Time Systems using Periodic Finite State Machines

R. Obermaisser, C. El-Salloum, B. Huber, H. Kopetz
Vienna University of Technology

Abstract: Finite State Machine (FSM) models are widely used to specify the operations of computer systems. Since the basic FSM model is timeless, it is not possible to model within the basic FSM framework system properties that are dependent on the progression of real time, such as the duration of computations or the limited temporal validity of real-time data. To overcome these limitations, efforts have been made to modify the FSM model to include some notion of time. It is the objective of this paper to expand existing work on basic FSMs and timed automata to include the concept of a sparse global time base as a central element of the model. We call such an extended FSM model a *Periodic Finite State Machine (PFSM)* model. The PFSM model incorporates the notions of state variables, global time, periodic clock constraints, and time-triggered activities. Thereby, PFSMs enable a concise and intuitive representation of distributed control systems and reduce the gap between a modeled system and its implementation.

Keywords: Finite State Machine, Real-Time, Distributed Systems, Time-Triggered, Control System, Fault Tolerance

1 Introduction

It is the purpose of a model to make reliable predictions about the modeled behavior of a *natural system* or about the future behavior of an *artifact* that will be implemented exactly according to the rules governing the model. *Finite State Machines (FSM)* are in wide use for modeling the behavior of computing systems. They provide a useful framework for segmenting a large problem into a set of smaller steps that can be analyzed sequentially, thus reducing the cognitive complexity of each step to a level that is commensurate to the human problem solving capability [1]. The application of FSM models to real-time applications is hampered by the fact that the basic FSM model is timeless and therefore no prediction about the temporal behavior or the performance of the modeled artifact can be made. To overcome these limitations, a number of researchers have augmented the basic FSM model to include temporal properties, building on the seminal work of Alur and Dill [2]. To our knowledge, so far it has not been tried to expand the basic FSM model to include a *sparse global physical time base* as a central element of the model. The objective of the present paper is to introduce a formal model based on the concept of a sparse time base for specifying distributed embedded real-time systems. The formal model serves as a foundation for a model-based development process and formal analysis for distributed embedded real-time systems.

The contributions of the paper are as follows:

- ***Novel formalism that simplifies modeling and supports determinism, physical time, and time-sensitive real-time data:*** By willingly constraining the expressive power, we guide designers towards constructing systems that are understandable and formally analyzable. The proposed formalism introduces periodic transactions for modeling distributed embedded real-time systems. Phase alignment of periodic transactions on a sparse global time base rules out by design any problems of concurrency and simultaneity. The timing and data transformations of these transactions are fully deterministic and defined with respect to a physical time base. Transactions operate on real-time data that is associated with information concerning its temporal validity. Thereby, it is ensured that real-time data is always valid at the time of use.
- ***Support for formal analysis through model checking:*** We enable designers to formally verify properties of a system that is expressed in the proposed formalism by presenting a mapping to the input language of a model checker. The introduced constraints, such as periodic transactions and state changes that occur on

the action lattice of a global sparse time, work against the state explosion problem. Due to the ability to analyze systems with higher complexity, the modeling of realistic, real-world systems becomes possible.

- ***Bridging the gap from node-level models expressed in synchronous languages to system-level:*** Today, synchronous programming languages (e.g., ESTEREL [3]) are used to model individual node computers. This paper shows how these node-level models can be integrated into a model of a distributed computer system. Therefore, we demonstrate the compatibility of the proposed formalism with existing synchronous programming languages and point out the modeling of the communication system and the connection of the node-level models.

This paper is organized as follows: In the next Section we set up the fundamental concepts that form the foundations for the rest of this paper. Section three introduces the basic time-free FSM model and applies the model to a small example. Section four gives an overview of related work and describes the relationship to the presented formalism. Section five discusses the relevant temporal relations that have to be included in an expanded time-aware FSM model. An expanded time-aware FSM model called periodic finite state machine (PFSM) is introduced that takes all identified temporal issues into consideration. In Section six design considerations, such as the composition of systems built out of a set of interacting PFSM models, are discussed. Section seven describes the formal analysis of PFSMs using model checking and illustrates this process through an example. Section eight summarizes the key contributions and discusses the results of the paper. The paper terminates with a conclusion in Section nine.

2 Basic Concepts

In this Section we introduce the basic concepts used in this paper. Where possible we follow the definitions established in the context of the European project Dependable Systems of Systems (DSOS) [4].

2.1 Model of Real Time

We model the *flow of time* by a directed timeline that extends from the *past* to the *future*. A *cut* of the timeline is called an *instant*. The instant *now* is a special instant, the *present*, which separates the *past* from the *future*. An interval between two instants is called a *duration*. An *event* is a *relevant happening* that occurs at an instant. A device for measuring the progression of time is called a *clock*. A *digital clock* contains a counter and a physical oscillation mechanism that periodically produces a *tick* to increment the counter.

Even if every clock of an ensemble of clocks, each with its own oscillation mechanism, starts with the same initial value, the clocks will diverge over time due to differences in the physical oscillation mechanisms, the differences in the *rates* of the clocks. In order to keep the clocks in approximate synchrony, we must resynchronize the clocks periodically. The *timestamp* of an event that is generated by a clock of the ensemble is called the *local timestamp* of this clock. Since the precision [5] of an ensemble of clocks is always greater than zero, the local timestamps of a single event, observed by two different *good* clocks of the ensemble, can be different. In general it is therefore not possible to establish the *temporal order* of events on the basis of their local timestamps.

The concept of a *sparse time* helps to overcome this difficulty. If the occurrence of events of significance (e.g., the *sending of a message* or the *observation of the environment)* is restricted to some *active intervals* on the timeline of duration ε, followed by an *interval of silence* of at least a duration Δ (which must be properly chosen [6]) between any two *active intervals*, we call the time base ε/Δ *sparse* and the events *sparse events*. We number the active intervals by the positive integers and call the integer number assigned to an active interval, the *global timestamp* (or *timestamp* for short) of events happening in that interval. We consider all events that happen within the same interval ε as having occurred *simultaneously*. A *system-wide consistent temporal order* of all sparse events occurring in a distributed system can now be established on the basis of their global timestamps. We assume that all events that are in the *sphere of control* of the system (e.g., the sending of messages) happen within the active intervals ε of the sparse time base and are therefore sparse events (see also Section 6.5 which discusses how to transform events that are outside the sphere of control of the system and occur on a dense time base to *sparse events*).

To summarize, we assume a *single global sparse time base* that is in full synchrony with the international standard of time TAI to timestamp all sparse events of relevance in a distributed system and are thus in the position to create a system-wide consistent temporal order of events on the basis of these timestamps. We therewith solve the challenging problem of *determining consistently the simultaneity of events* in a distributed system in order to be able to *avoid race conditions* and *maintain determinism* where required.

2.2 System Structure

According to DSOS [4] we define a *system* as *an entity that is capable of interacting with its environment and may be sensitive to the progression of time*. We assume that a system can be decomposed into a set of interacting *subsystems*. We call a subsystem a *component*, if a further decomposition of this subsystem is of no further interest (in the given context).

Our systems interact with their environments across *message interfaces*. A *message* is an atomic data structure that is formed for the purpose of inter-process communication. A message interface consists of one or more unidirectional *ports*, where at any instant only one message can be sent or received at a port. A message interface produces/consumes messages by *send/receive* operations. We call a sequence of (perhaps timestamped) send and receive operations at an interface the *behavior* of the system at this interface. The *intended behavior* is called the *service* of the system at this interface.

2.3 State

We introduce the notion of *state* in order to separate the *past behavior* from the *future behavior* of the system:

The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other word, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of a system. Knowing the state "supplants" knowledge of the past. […] Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered ([7],p.45).

The state is recorded in *state variables* that are selected variables either in the environment or the computer system. We call a state variable a *real-time entity* (RT entity) and a picture of an RT entity a *real-time image* (RT image) [8]. An example of a RT entity in a control system is the *flow* in a pipe or the *temperature* of a vessel. An example for a RT image is the representation of the flow in a computer system. In many applications, it is not the full history but only a *selected part of the history* of the system that is considered relevant for the future behavior. We call this selected part of the history the *declared state* of a system. The determination of the *declared state* of a given application is one of the most important decisions in the design of a real-time computer system.

In general, RT images in distributed real-time control system exhibit a *limited temporal validity* of the real-time data. *It is a fundamental property of any real-time system that the validity of real-time data is invalidated by the progression of real-time [9]*. We call the instant when an observation becomes invalid, the *invalidation instant* t_{inval}. If an observation is used after t_{inval}, a catastrophic failure can occur. We must make sure that real-time data used in any control action at the *instant of use* t_{use} is still temporally valid i.e., $t_{use} < t_{inval}$, otherwise state estimation has to be performed. The *unspecified durations* of an intermediate storage of real-time data, e.g., in the queues of a communication system or of an operating system, are thus hazardous in a hard real-time system.

In order to extend the temporal validity of state variables *state estimation* can be used. State estimation involves the building of a model of an RT entity to compute its probable state at a selected future point in time, and to update the corresponding RT image accordingly. The state estimation model is executed periodically within the system that stores the RT image. The control signal for the execution of the model is derived from the tick of the real-time clock associated with the RT entity. The most important future point in time where the RT image must be in close agreement with the RT entity is t_{use}, i.e. the point in time where the value of the RT image is used to deliver an output to the environment.

2.4 State and Event Messages

The recording of some relevant aspect of the state of the environment at an instant (on the sparse time base) is called an *observation*. An observation is thus an atomic triple consisting of

(*name_of_observation, instant_of_observation, value_of_observation)*

The *name_of_observation* denotes the *concept* that is the subject of observation and thus bridges the gap between syntax and semantics. The *instant_of_observation* records the point in time (on the sparse time base) when the observation was generated (took place). The *value_of_observation* records the values in the relevant data structure.

We call a message a *state-message* if its value results from the *observation* of a state variable, i.e., a RT-entity [8]. In agreement with *state semantics*, a new arriving version of a state message overwrites an old ver-

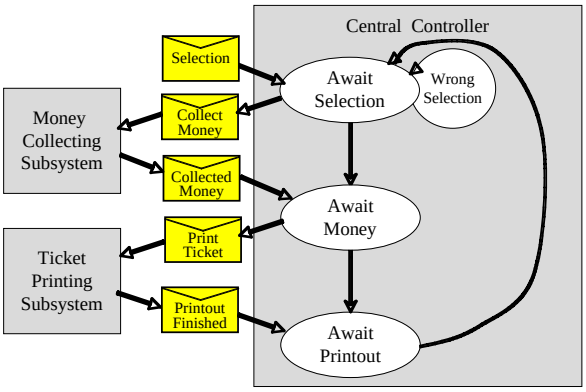

Fig. 1: FSM model of a simple vending machine

sion (update in place). State-messages are not consumed on reading. There is normally no control signal associated with the arrival of a state message.

We call a message an *event-message* if its value refers to the difference between an *old state* and a *new state*, i.e., *event information*, Event-messages must be handled according to the *exactly-once semantics*, i.e., they must be queued in the communication system and consumed on reading. The arrival of new event message is normally signaled to the receiver, e.g., by an interrupt mechanism.

3 Finite State Machines

This section describes the basic finite state machine model and analyses its limitations in the context of distributed real-time systems.

3.1 Basic Finite State Machines

We widely follow the notation of [10] p.7 to describe the basic finite state machine (FSM):

A basic FSM is five-tuple where

Q is a finite set of symbols denoting states
Σ is a set symbols denoting the possible inputs
Δ is a set of symbols denoting the possible outputs
σ is a transition function mapping $Q \times \Sigma$ to $Q \times \Delta$
$q_0 \in Q$ is the initial state.

*In one **transition**, an FSM maps a current state $p \in Q$ and an input symbol $a \in \Sigma$ to a next state $q \in Q$ and an output symbol $b \in \Delta$, where the transition function $\sigma(p,a) = (q,b)$. Given an input word, or sequence of symbols from the input alphabet Σ, and an initial state $q_0 \in Q$, a sequence of transitions will produce a sequence of states and an output word.*

It is common to represent FSMs by a directed graph, where the nodes represent *states* and each arc represents a *transition from one state to the next state*. A transition will fire as soon as there exists an enabled input symbol out of the current state.

The simple model of a vending machine for selling tickets of Fig. 1 is intended to demonstrate the application of the FSM to a small example. In this model we distinguish between three subsystems, the *central controller*, the *money collecting subsystem* that checks the legality of the coins and bills, and the *ticket printing subsystem*. Fig. 1 only shows the FSM model of the central controller. In the central controller we introduce three states (*await selection*, *await money*, *await printout*), three input messages (*selection*, *collected money*, *printout finished*) and two output messages (*collect money*, *print ticket*). Note that the words chosen for the interpretation of the states have a strong temporal connotation, although the model itself is free of real time. In a real vending machine, a number of physical time-outs (which are not part of this simple model of Fig. 1) and additional transitions will have to be added.

Referring to the discussion about state and event messages in the previous section the input messages (the input symbols $a \in \Sigma$) that are consumed when the corresponding transition fires, carry a control element and thus have the character of *event messages*, while the states $p \in Q$ have the character of *state messages*.

3.2 Critique of the Basic FSM Model

From the point of view of real-time systems, the main critique of the basic FSM model relates to its abstraction from *physical real-time*. Basic FSM models are *timeless*. It thus follows that notions that are directly related to the progression of real-time cannot be dealt with in the basic FSM world. Examples of such notions are: *duration of transitions, deadline, time-validity of real-time data*. But even notions that are only indirectly related to the progression of real time, many of them *fundamental*, cannot be expressed with the connotations that are commonly associated with these notions outside the FSM community. Examples of such notions are: *state, behavior, determinism, concurrency, simultaneity,* and *failure*. Note that the concept of *state*, as used in the basic FSM is not the same as the concept of *state* introduced in Section 2.3.

Let us look at a particular example, at the concept of *determinism*. In the FSM world a system is considered to be deterministic, if, given a defined initial state, the same sequence of input symbols will always produce the same sequence of output symbols. In more general usage, a system is considered *deterministic*, if the same sequence of output symbols is produced at *about the same future instants*. This temporal dimension of the concept of determinism is not part of the FSM definition; although this temporal dimension is needed if we intend to build fault-tolerant systems that are based on three replica-deterministic channels (triple-modular redundancy – TMR) and voting at an instant (see also Section 6.4). Let us define the determinism that is needed in a TMR system formally:

A computational system (*processing, communication*) is said to behave *TMR-deterministically* **iff, given** a sequence of *sparse real-time instants* t_i, the state of the system $q_0(t_0) \in Q$ *at* t_0 *(now),* and a sequence of *future* inputs $a_i(t_i) \in \Sigma$ **then** the sequence of *future* outputs $b_j(t_j) \in \Delta$ and the sequence of future states $q_j(t_j) \in Q$ is *entailed.*

Note that in a TMR system at least two of the three independent channels (the third channel can have a failure and thus an arbitrary behavior) must produce the same outputs at the same instant. This requires a *consistent definition of simultaneity* in a physically distributed system with *independent* fault-containment regions (this implies that each node must have its own synchronized clock). The *sparse time base* introduced in Section 2.1 makes such a consistent definition of simultaneity possible. The basic FSM model is not capable to deal with these delicate, but important, issues related to the temporal behavior of an artifact that is built according to the basic FSM model.

4 Related Work

This section describes related work on the extension of finite state machines for modeling of distributed real-time systems. Timed automata have been studied extensively and practically employed for model checking and code synthesis of distributed real-time systems.

4.1 Timed Automata and Extensions for Modeling Distributed Real-Time Systems

Timed automata have been introduced in [2] as an extension of finite state machines for modeling and analyzing real-time systems. A timed automaton is a five tuple $\langle Q, \Sigma, C, E, q_0 \rangle$, where

Q is a finite set of symbols denoting states
Σ is a set of symbols denoting the possible inputs
C is a finite set of clocks
$E \subseteq Q \times Q \times \Sigma \times 2^C \times \phi(C)$ is a set of transitions, where $\phi(C)$ is the set of clock constraints
$q_0 \in Q$ is the initial state.

The firing of a transition $< q, q', a, \lambda, \delta > \in$ E from the current state q to the state q' is controlled by the enabled inputs a of Σ and the clock constraint $\delta \in \phi(C)$. A timed automaton possesses a finite number of clocks, each of which measuring the elapsed time since they were started or reset. A transition can be associated with the reset of clocks, which is specified by an element λ of 2^C. The clocks of a timed automaton are synchronously incremented.

Based on the basic timed automaton model, extensions have been introduced to build more expressive models or to ease the modeling task for distributed real-time systems.

4.1.1 State Variables

While the timed automata defined by Alur and Dill [2] only contain clock variables, an extension of timed automata for the model checker Uppaal [11] introduces state variables with finite value domains.

In Uppaal a timed automaton is a tuple < L, l_o, E > where L is a finite set of nodes, l_o is the initial node and E is a set of edges. An edge $\langle l, g, a, r, l' \rangle \in E \subset L \times \mathfrak{B}(C,I) \times Act \times \mathfrak{A}(C,I) \times L$ leads from a location l to a location l'. Each edge possesses a guard *g*, which is a boolean constraint expressed using clock variables *C* and state variables *I*. An edge can only be taken, if the guard is satisfied. The action *a* specifies CSP-style [12] synchronization actions using blocking transmission and reception of messages. The reset operation *r* serves for the manipulation of clock variables and state variables. Clock variables can be set to a natural number, i.e., $x = n$, where x is a clock and $n \in \mathbb{N}$. An assignment to a state variable is of the form $i = i \cdot c_1 + c_2$, where *i* is a state variable and $c_1, c_2 \in \mathbb{N}$.

4.1.2 Periodic Clock Constraints

Based on the aperiodic clock constrains of timed automata, the timed automata model has been extended using periodic clock constraints in [13]. Periodic clock constraints include expressions such as $\phi := k \in \mathbb{N} : x \in [a + k\lambda, b + k\lambda]$, where *x* is a clock and $a, b, \lambda \in \mathbb{N}$. [13] recognizes that many processes in different fields have a periodic behavior, thus models are required that permit to express regularly repeated real intervals. Periodic constraints also increase the expressive power of the timed automaton model. The expressive power of timed automata with periodic clock constraints and no ε - transitions is between the expressive power of timed automata with aperiodic constraints and no ε - transitions and timed automata with aperiodic constraints and ε - transitions [13].

4.1.3 Triggering of Activities at Predefined Points in Time

Time-triggered automata [14] are a subclass of timed automata accepting digitalized timed languages. Transitions are taken only at integer time points determined by periodic and aperiodic timing constraints. The interaction with the environment is modeled using sequences of events with timing constraints. Thus, time-triggered automata are aimed at synthesizing code for time-triggered architectures from event-triggered specifications. Time-triggered automata enable a designer to express at what points in time a system shall react to events.

Another example for an extension of timed automata for modeling the triggering of activities at predefined points in time are calendar automata [15]. A calendar is a finite set $\{\langle e_1, t_1 \rangle, \ldots, \langle e_n, t_n \rangle\}$, where each event e_i is scheduled to occur at time t_i.

4.1.4 Global Time Base

Motivated by time-triggered systems (e.g., TTA [16]), which control all activities using a single global time base, models based on timed automata have been proposed that enforce the restriction of a single clock variable (e.g., time-triggered automata [14]). Besides the intuitive modeling of the class of time-triggered systems, this restriction reduces the computational complexity for verification in contrast to models constructed with timed automata using multiple clock variables.

4.2 Relationships to Presented Work

The Periodic Finite State Machines (PFSMs) described in this work combine these extensions, which have been individually addressed in previous work. In addition, the PFSMs exploit the concept of a sparse time base in order to model systems with a consistent view on the temporal order of events and a consistent distributed state.

The PFSMs apply the sparse time base model by distinguishing between stable states and activity states in correspondence to the two types of intervals in the action lattice of the sparse time base. The stable and activity states are periodically recurring time intervals that are defined w.r.t. to the global time base. Within the stable state, any event, i.e. a state change of the output variables that has occurred in the activity interval, is consistently perceivable to the PFSM's environment (e.g., the other PFSMs in the system).

The explicit capturing of an activity interval of bounded length and the demarcation from a stable interval in which no state changes of output variables are allowed to occur, ensures that the consistency properties of a model constructed with PFSMs also hold in an actual implementation. PFSM-based models directly map to time-triggered implementation platforms that are based on the sparse time base model (e.g., TTA [16]). In conjunction with such an implementation platform, PFSMs enable the designer to focus on the application requirements (e.g., functional, temporal, dependability properties), while being constrained by the formal framework in such a manner that a consistent distributed state is established. In particular, the formal framework is realistic in the sense that this consistent distributed state is not refuted by moving to a physical system.

Other formalisms based on timed automata either require the designer to devise synchronization protocols (e.g., process synchronization via channels [11]) or encourage unrealistic models, e.g., by assuming instantaneous

updates of data variables. The latter approach cannot be implemented in a physical system due to the impossibility of perfectly synchronizing an ensemble of clocks.

5 Periodic Finite State Machines (PFSM)

In this section we extend the previous work on finite state machines with timing constraints to support the sparse time model and incorporate the notions of state variables, global time, periodic clock constraints, and time-triggered activities. Since the control signal for the start of a transition is derived from the progression of time in the periodic control cycles, we call the extended FSM a *Periodic Finite State Machine* (PFSM).

The PFSM model provides a framework for modeling the temporal properties of real-time systems. Temporal parameters of a particular execution environment, such as the worst-case transition time of a state machine are part of the model. The input and output interfaces of a PFSM model are based on state semantics. These interfaces support unidirectional data-flow, do not contain control signals and thus have the characteristics of *temporal firewalls* [17]. The PFSM model is related to the computational model that is behind synchronous programming languages, such as ESTEREL [3] or LUSTRE [18] or the time-triggered model of computation [19].

5.1 Temporal Relations

The temporal structure of a typical distributed real-time control system is *cyclic*. Each *control cycle* can be decomposed into a number of steps, as shown in Fig. 2. After the start of the control cycle the distributed sensors are observed and the sensor data is preprocessed (step A) at the sensor nodes before the data is transmitted to the node (step B) that executes the control algorithm (*transition* of the model in step C). The outputs of the control algorithm are transmitted to the actuators (step D), which transform the data into physical actions (step E), e.g., by setting the control valves into the calculated positions.

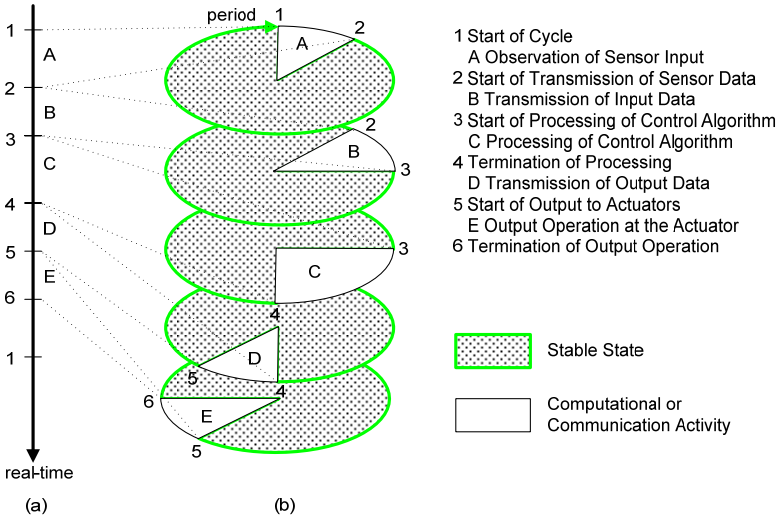

Fig.2: Typical Sequence of Activities in a Distributed Control System
(a) linear model of time (b) cyclic model of time

In Fig. 2(a) these phases are shown using a *linear model of time*, while 2(b) uses a *cyclic model of time* [20]. In the *cyclic model of time* – which we feel is more appropriate – an event can be specified by the cycle number and the phase, i.e., the offset from the *start_of_cycle*. The phase can be expressed in degrees. It is to be noted that the transmission and execution steps occur in *bursts*, i.e., they are not equally distributed over the full cycle.

With a PFSM, both computational and communication activities can be modeled. In the graphical representation of the distributed control system as shown in Fig. 2, a PFSM represents one of the discs for the communication and computational activities. From the temporal point of view, we have to consider the periods and durations of these computational and communication activities. The disc's circumference equals the period of the PFSM. The activation instant and the duration define the sector of activity. We call the maximum duration of the activity of a PFSM as the *Worst-Case Transition Time (WCTT).*

5.2 Formal Definition of Periodic Finite State Machines

A Periodic Finite State Machine (PFSM) is a tuple $\langle Q, q_0, d_{\text{cycle}}, d_{\text{phase}}, d_{\text{WCTT}}, I, O, H, \sigma \rangle$, *where*

Q *is a finite set of symbols denoting states,*

$q_0 \in Q$ *is the initial state,*

$d_{\text{cycle}} \in \mathbb{N}$ *denotes the period of the PFSM*

$d_{\text{phase}} \in \mathbb{N}$ *denotes the start phase of the transition function* σ

$d_{\text{WCTT}} \in \mathbb{N}$ *denotes the worst case transition time*

$I = \langle (v_x, \mathbb{D}_x), (v_y, \mathbb{D}_y), ... \rangle$ *is the tuple of input variable declarations, where* v_x, v_y are variable symbols and $\mathbb{D}_x, \mathbb{D}_y$ are the respective value domains

$O = \langle (v_u, \mathbb{D}_u), (v_v, \mathbb{D}_v), ... \rangle$ *is the tuple of output variable declarations, where* v_u, v_v are variable symbols and $\mathbb{D}_u, \mathbb{D}_v$ are the respective value domains

$H = \langle (v_a, \mathbb{D}_a), (v_b, \mathbb{D}_b), ... \rangle$ *is the tuple of history variable declarations, where* v_a, v_b are variable symbols and $\mathbb{D}_a, \mathbb{D}_b$ are the respective value domains

σ *is a transition function mapping*

$$Q \times T_{\text{act}} \times \underbrace{(\mathbb{D}_x \times \mathbb{T})}_{\text{input variable } v_x} \times \underbrace{(\mathbb{D}_y \times \mathbb{T})}_{\text{input variable } v_y} \times ... \times \underbrace{\mathbb{D}_a}_{\text{history variable } v_a} \times \underbrace{\mathbb{D}_b}_{\text{history variable } v_b} \times ...$$

to

$$Q \times (\underbrace{\mathbb{D}_u \times \mathbb{T}}_{\text{output variable } v_u}) \times (\underbrace{\mathbb{D}_v \times \mathbb{T}}_{\text{output variable } v_v}) \times ... \times \underbrace{\mathbb{D}_a}_{\text{history variable } v_a} \times \underbrace{\mathbb{D}_b}_{\text{history variable } v_b} \times ...$$

, where $T_{act} = \left\{ t \in \mathbb{T} \middle| t = k \cdot d_{cycle} + d_{phase} \wedge k \in \mathbb{T} \right\}$ *and* $\mathbb{T} \subset \mathbb{N}$ *is a finite set of instants of the global sparse time base*

The input and output variables of a PFSM are *state variables*. A current state of a state variable $(v_i, \mathbb{D}_i)$ is described by the tuple (a, t), where $a \in \mathbb{D}_i$ denotes the *value* of the variable w.r.t. its domain, and $t \in \mathbb{T}$ denotes the *invalidation instant* of the value, i.e., an instant in the future at which the value will become invalid. The state of a history variable $(v_h, \mathbb{D}_h)$ is described by a value $a \in \mathbb{D}_h$.

In one transition, a PFSM moves from a current state $q_c \in Q$ to a next state $q_n \in Q$, and generates the resulting output and history variables based on the transition function $\sigma(q_c, t_{act}, (a_x, t_x), (a_y, t_y), ..., a_a, a_b, ...) = (q_n, (a_u, t_u), (a_v, t_v), ..., a_a, a_b, ...)$ where $t_{act} \in T_{act}$ denotes the activation instant of the transition. $a_x, a_y, ...$ denotes the values, and t_x, t_y the invalidation instants of the input variables and, respectively, $a_u, a_v, ...$ denotes the values, and t_u, t_v the invalidation instants of the output variables of the PFSM. $a_a, a_b, ...$ denotes the values of the history variables of the PFSM.

In a *PFSM* the control signal for the initiation of *transition* is not derived from the arrival of an input message (input variables have state semantics), but from the progression of the *sparse global time*. Since the activities in a control system are periodic, the control signal can be derived from phase instants of the *periodic passage* of the control cycle. The periodic activation instants are defined by the period of the PFSM d_{cycle} and its start phase d_{phase}. In contrast to other finite state automata, time elapses in the transitions of a PFSM. This property captures real-world execution and communication delays in a realistic manner.

After the *periodic time-triggered* control signal associated with the transitions of a specific PFSM has occurred, the input state variables and the current state $q_c \in Q$ are accessed. The transition function σ is executed and after the completion of the *transition* the output *variables* and the next state $q_n \in Q$ are produced. The transition is always completed before d_{WCTT} – the worst case transition time. After the *WCTT*, the output variables and the state of the PFSM stay stable until the next activation at the phase d_{phase} of the transition function. The state of a PFSM is only defined during this stable interval. During the WCTT, input state variables must be stable and may not be written by another PFSM.

The transition function derives the invalidation instants of the output variables from the invalidation instants of the input variables, the activation instant of the transition and the current state of the PFSM. As an example, consider an output variable that is a function of multiple input variables. In the general case the invalidation instant of the output variable would be set to the minimum invalidation instant of all the input variables, but if for example *state estimation* (cf. Section 6.3) is applied, it could be greater than the invalidation instant of all the input variables.

The timestamp of the activation instant of a transition (the current real-time) is part of the input to the transition function σ. The current real-time appears inside a transition only as a *data element*, but not as a *control signal*. It is only assumed that a transition will terminate with its results before the *WCTT*. We assume that the

worst case transition time d_{WCTT} is at first allocated (during the *design*) and later validated (during the *test and validation*) for the targeted execution environment [21].

6 Design Considerations

In this Section some issues concerning the design and composition of large distributed systems that are built out of a plurality of interacting PFSM models are discussed.

6.1 Semantic Categories

When modeling a distributed real-time system, PFSMs with different semantic roles can be distinguished. These roles correspond to the following types of structural elements of a distributed real-time system:

Computational actions: A computational action is a PFSM that models a computational element, which performs an application-specific data transformation at a certain period and phase. In addition to providing a function that relates the outputs of the computational elements to its inputs, the computational action defines the validity of the generated outputs. The PFSM model makes *no assumptions* about the internal structure of a computational action (e.g., whether the execution within a computational action is *sequential* or *concurrent*, whether it is implemented in software on a CPU, in an FPGA, or directly in hardware in the form of a state machine). It is only assumed that a computational action will terminate with its results before the WCTT of the PFSM has passed. We denote the WCTT of a PFSM that models a computational element as the *Worst-Case Execution Time (WCET)*.

The duration of the processing activity depends on the complexity of the control algorithm and the processing capabilities of the available execution environment. There is a significant amount of literature that deals with estimating the WCET of a processing step [21]. In a real-time system a low upper bound on the maximum execution time of a processing step is more important than a low average execution time with extremes at either side.

Communication actions: The communication actions model the communication system that interconnects the computational elements. A communication action samples the output state variables of one or more computational actions at a certain period and provides these state variables as inputs for subsequent computational actions. The communication action introduces a delay, which depends on the design and performance of the available physical communication system. The maximum value of this delay is called the *Worst-Case Communication Time (WCCT)* and specified by the WCTT of the PFSM.

The WCCT depends on the design and performance of the available physical communication system. As shown in Fig. 2 the communication in a real-time control system occurs in *bursts*, where the instant of occurrence of each burst is known *a priori*. We call such a periodic bursty transmission pattern a *pulsed data stream* [22]. A proper real-time communication system will take advantage of the *a priori* knowledge about the occurrence of the bursts and will provide sufficient bandwidth during the *known burst intervals* in order to minimize the duration of the bursts.

6.2 Composition of PFSMs

A prerequisite for any composition of subsystems is the availability of a precise interface specification of the interfacing subsystems, both at the *operational level*, which includes syntax and timing, and at the *meta-level*, which specifies the semantics of the data [23]. In the context of this paper, we are mainly concerned with compositional issues relating to the timing at the interfaces. We explore the behavior of a composition of a set of independently developed PFSM models. The loose coupling between two PFSMs realized solely by the *unidirectional data exchange via state variables* [24] across the interfaces, the absence of any event-triggered control signals at the interfaces of the interacting PFSMs, and the abstraction from the internal behavior of transitions and communication steps simplifies the composition of PFSM models.

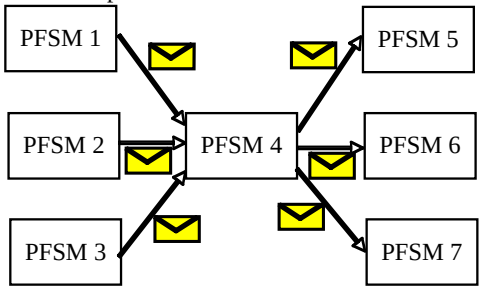

Fig. 3: Composition of PFSM models

We can compose PFSMs in an *m-to-n* topology as shown in Fig. 3. In this figure three PFSMs produce input data for PFSM 4 and three PFSM read the output data from PFSM4.

The composition optimizes the temporal behavior of an aggregate of interacting PFSM models and maintains the determinism of the ensemble. When composing PFSM models, the periods of the interacting PFSMs are synchronized with respect to duration and phase to generate *phase-aligned* transactions. Based on *the a priori knowledge* of the instants when data is produced and read by a PFSM (derived from the progression of the sparse global time), the schedules can be arranged to avoid any data-access conflicts between interacting PFSMs, thus avoiding race conditions and maintaining the determinism of an ensemble of interacting deterministic PFSMs. The periods of all PFSMs do not have to be same and they can be in a harmonic relationship. The composition of PFSMs needs to assure that PFSMs are aligned in such a way that the output of one PFSM is stable during the activity interval of any other PFSM that reads these state variables as inputs. The advantage gained by the introduction of these constraints is the maintenance of determinism and the optimal phase-alignment of real-time transactions covering multiple PFSMs.

6.3 Determination of the Periods of a PFSM

The period of a Periodic Finite State Machine (PFSM) that does not employ state estimation is determined by two aspects: the performance of the physical execution environment and the dynamics of the modeled application, causing the limited temporal validity of the real-time data.

Performance of the Execution Environment: According to Fig. 2, all processing and communication steps between the observation of a RT entity and the output operations to the actuators must be completed within a cycle in a non-pipelined system. The sums of the WCETs of all processing steps and the WCCTs of all communication steps must thus be smaller than the duration of a cycle, establishing a lower bound for a cycle duration:

$$d_{cycle} > \Sigma\, WCET + \Sigma\, WCCT$$

Temporal Validity of Real-Time Data: If we assume that a RT entity of the modeled application that has been observed at t_{gen} changes its value as a function of time according to $V(t)$, then a delayed use of the data at t_{use} will result in a first approximation in a data error of

$$\Delta V_{time} = dV/dt\,.\,(t_{use} - t_{gen})$$

This data error ΔV_{time} caused by the temporal delay should be in the same order of magnitude as the tolerated data error ΔV_{data} in the data domain (caused by the inaccuracy of the sensors or for other reasons). According to Fig. 2, the real-time data that is sampled at the beginning of a cycle will be used before the end of the cycle. We thus arrive at an upper bound for the cyle duration by

$$d_{cycle} < \Delta V_{data}/(dV/dt)$$

It follows that a cycle duration should be between the limits

$$\Sigma\, WCET + \Sigma\, WCCT < d_{cycle} < \Delta V_{data}/(dV/dt)$$

If such a short cycle duration cannot be implemented, due to the limited capability of the given execution environment (performance limitations of the processor and communication system), then the *method of state estimation* [8] for the instant of use, t_{use}, must be resorted to in the computation in order to extend the validity time of the real-time data to the time of use. This is possible if the modeled process has a regular behavior. For example, in an automotive engine the position of a piston in a cylinder can be extrapolated from t_{gen}, when the observation occurred, to a future use instant t_{use} by a Taylor expansion, given the interval $(t_{use} - t_{gen})$ corresponds to a few degrees of an engine cycle. This extrapolation is possible, because the speed of an engine does not change abruptly.

Computational models that are based on temporally unspecified message queues between processing steps and assume that each firing of a processing step consumes an input message and produces an output message are fundamentally different from our PFMS model. From the point of view of real-time performance, the duration that a message has to reside in such a queue has an effect on the temporal validity of the real-time data. This buffering delay cannot be neglected from the point of view of temporal correctness of the real-time data. In order to calculate an upper bound of the maximum duration that a message may reside in a queue (which must be less than $(t_{use} - t_{gen}) - (\Sigma\, WCET + \Sigma\, WCCT)$) many assumptions about the behavior of the environment and the parameters of the implementation must be made [25]. Since some of these assumption are difficult to enforce during the execution, the *assumption coverage* [26] of such a calculation will be limited. An example of such a model is the process network model originally introduced by Kahn [27].

6.4 Fault Tolerance

Fault-tolerance by replication and voting can be implemented by replicated PFMS as shown in Fig. 4. The unidirectional data transfer between PFSM models assures the independence of the failure modes of the replicated units, an *imperative* requirement of TMR configurations. Computational models that assume a tight bidirectional synchronization between the sender of a message and the receiver of a message, such as the CSP model [12] are not appropriate as a basis of a design pattern for fault-tolerant systems. In these models the sender and receiver are more tightly coupled than in the PFSM model, since progress of the sender depends on the *correct* progress of the receiver. In these models a failure of a receiver of a message can have a direct impact on the progress of the sender of the message. An implementation that is based on such a model cannot be used in a TMR (triple modular redundancy) configuration, since the *back-propagation of errors* from the receivers to the sender will invalidate the basic independence assumption that is necessary to achieve fault-tolerance.

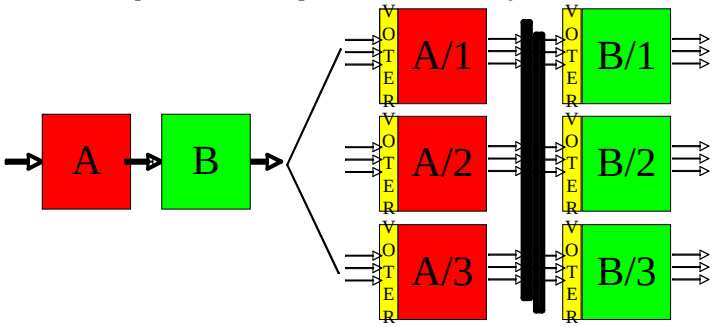

Fig. 4: Expansion of two PFSM, A and B into a Fault Tolerant Configuration

Due to trends in chip technology (e.g. shrinking feature sizes) the transient failure rates are steadily increasing [28]. Of particular importance in the presence of transient faults is the notion of *state restoration* in order to re-integrate a failed component. The challenge of a distributed system after the occurrence of a transient fault is to reach a correct state again after the transient fault has disappeared. In case a N-Modular Redundancy (NMR) system is built from PFSMs comprising only a single state symbol q, a corrupted state within a single PFSM (e.g., bit flips in input or output variables) can be corrected in every cycle, if the PFSM votes on all input variables disseminated by the PFSMs. For this purpose, all input variables must be redundantly acquired via redundant PFSMS as part of a NMR configuration.

6.5 Interaction with Environment

6.5.1 Sensors

The PFSM model is based on state messages. However there are situations where sensors will deliver event information. In these situations local intelligence and local storage must be provided to transform the event messages into state messages at the sensor node. For example, in the vending machine of Fig. 1, the input of coins at the user interface corresponds to event messages. The local logic of the money collecting subsystem has to transform the event messages into state messages, i.e. in the accumulated amount of money collected over the period of a transaction. Whenever a transaction has finally completed, the amount of money collected must be reset to zero.

The period of the PFSM model enforces a *minimum duration* between two successive activations of a processing step, i.e., a *transition*. The establishment of such a minimum duration is an absolute necessity for the correct temporal performance of a control system and for the generation of a feasible schedule. In event-based systems (e.g., an interrupt driven system) the establishment of this minimum duration between processing activations is a delicate task.

If events in the environment occur on a dense time base – as they normally do – then an *agreement protocol* [8] must be executed among replicated observers of a single event in order to assign the event consistently to a unique active interval of the sparse time base, and thus transform any arbitrary event to a *sparse event* (see also Section 2.1). An agreement protocol is also needed in the value domain if continuous physical quantities are digitalized.

6.5.2 Actuators

An actuator accepts the output variable from the preceding PFSM and exerts an effect on the environment at the activation instant t_{use}. For example, the release of the trigger of a gun is the output event at t_{use}. The action instant can be either periodic or part of the state variables representing the output message to the actuator.

The design must ensure, that $t_{use} < t_{inval}$ for all output state variable to this specific actuator. If this constraint is not satisfied by the selected control cycle (see Section 6.3), state estimation must be deployed.

6.6 Requirements on the Execution Infrastructure

The PFSM model requires an execution environment that establishes and maintains a (fault-tolerant) global time base of known precision. An example for such an execution environment is the time-triggered architecture (TTA) [16].

If the computations of a distributed control system are structured according to the PFSM model the requirements for computational and communication resources will not be uniform over time, but will occur in *bursts* as shown in Fig. 2(b). In particular, high-bandwidth communication services are needed for short *a priori known* intervals during each control cycle (depicted by the green area in Fig. 2(b)). A similar *pulsed behavior* is typical for many other real-time applications. For example, in a fault-tolerant system that masks failures by triple-modular redundancy, a high bandwidth communication service is required for short intervals to exchange and vote on the state data of the replicated channels. Similarly, in a multimedia scenario a high-bandwidth communication is needed between frames to exchange the frame data. A real-time communication network should take consideration of these pulsed communication requirements and provide appropriate services (e.g., using pulsed data streams [22]).

6.7 Stability of Inputs

The sparse time base model of PFSMs rules out by design any problems of concurrency and simultaneity. It divides time into an infinite set of activity and stable intervals. During the stable interval of a PFSM, the contents of the PFSM's output variables remain unchanged. During this so-called *stable state*, consistent input is provided to other PFSMs that use the output variables as their inputs. At the specified phase, the stable state transits into the activity state, where variables are updated using the transition function σ. The *activity state* is periodically entered at the phase of the PFSM $k \cdot d_{\text{cylce}} + d_{\text{phase}}$ in each period k $(k \in \mathbb{N})$. The activity state terminates by a transition into the stable state of the next PFSM state before the WCTT has elapsed at time $k \cdot d_{\text{cylce}} + d_{\text{phase}} + d_{\text{WCTT}}$.

7 Formal Analysis using PFSMs

Starting from the specification of an application as a set of PFSMs, this section describes the formal analysis by model checking. We use the SAL (Symbolic Analysis Laboratory) framework, which includes tools for symbolic and bounded model checking of discrete transition systems.

7.1 Expressing PFSMs in SAL

The PFSM as introduced in this paper, have been expressed in the SAL language [29]. The specification in the SAL language consists of separate modules, namely one *clock module* and multiple *PFSM modules*. The clock module generates the ticks of the physical clock that cause the taking of transitions within the PFSM modules. The overall model results from the synchronous composition of the clock module and the PFSM modules. The semantics of synchronous composition in SAL is that the overall model consists of initializations that are the combination of initializations from the individual modules, and the transitions are the combinations of the individual transitions of the individual modules.

```
[...]
table: array [0..table_length-1] of NATURAL = (
    [[i: [0..table_length-1]] 0]
      WITH [0]:= 1
      WITH [1]:= 1
      WITH [2]:= 1
      WITH [3]:= 2
      WITH [4]:= 5
);
[...]
clock: module =
 begin
    LOCAL index: [0..table_length-1]
    OUTPUT time: Time
    initialization
    index = 0;
    time = 0;
    transition
```

```
    [
       time_progress:
       TRUE --> time'=(time+table[index]) mod TIME_HORIZON;index'=(index+1) mod table_length;
    ]
end;
```

Fig. 5: Clock Module

Figure 5 shows the clock module expressed in the SAL language. This module contains an output-variable `time`, which stores the current global time with a granularity of one ms. Each transition is associated with a time progress of one or more ticks. In order to increase the efficiency of model checking, each time progress advances to the next value of the global time at which a transition function needs to be executed within a PFSM. The time increments for these updates of the global time are computed prior to model checking and stored in a table (cf. array variable `table` in Figure 5). Since PFSMs are periodic, the time increments in the table also recur periodically. The table has to define the time increments for the common multiple of the PFSM periods. In order to keep the common multiple of the PFSM periods low, harmonics can be used for the periods as suggest in Section 6.2.

```
  PFSM1_State: type = {pfsm1_q0};
  PFSM2_State: type = {pfsm2_q0, pfsm2_q1 };
[...]

  n: NATURAL =  6;
  index: TYPE = [1..n];
  T_PFSM: TYPE = [# period: NATURAL,phase: NATURAL,wctt: NATURAL #];

   % Definition of Periodic Finite State Machines
   PFSM: array index of T_PFSM = (
         [[i: index] empty_pfsm]
          WITH [1] := (# period:=10, phase:=1, wctt:=1 #)
          WITH [2] := (# period:=10, phase:=2, wctt:=1 #)
[...]
  );

pfsmX: module =
    begin
      INPUT time: Time
      INPUT v_variable_i1: Speed
      INPUT d_variable_i1: Speed
[...]
      OUTPUT v_variable_o1: Speed
      OUTPUT d_variable_o1: Speed
[...]
      LOCAL v_variable_h1: Speed
[...]
      LOCAL q: pfsmX_State
      initialization
        q = pfsmX_q0;
      transition
        [
          active:
          ((time mod PFSM[X].period + PFSM[X].wctt) = PFSM[X].phase) -->
[...]
          []
          idle:
          (NOT ((time mod PFSM[X].period + PFSM[X].wctt) = PFSM[1].phase)) -->
        ]
    end;
```

Fig. 6: PFSM Module

A PFSM module is depicted in Figure 6. The PFSM module takes a transition whenever a time increment within the clock module occurs (due to the synchronous composition). Depending on the value of the global time, either the transition function fires (cf. rule `active` in Figure 6) or the PFSM idles (cf. rule `idle` in Figure 6).

The transition function fires if the global time `time` is $k \cdot d_{\text{cycle}} + d_{\text{phase}} + d_{\text{WCTT}}$ $(k \in \mathbb{N})$, where d_{cycle}, d_{phase} and d_{WCTT} are part of the defining parameters of the PFSM (cf. Section 5.2). The transition function reads the current state (`q`), the input variables (`v_variable_i1, d_variable_i1, …`), the local variables

(`v_variable_h1, …`), and the current global time (`time`). In addition, the transition function writes the output variables (`v_variable_o1, d_variable_o1, …`), and the local variables (`v_variable_h1, …`).

Note that the SAL model deterministically executes the transition function after the worst-case transition time d_{WCTT}. Although, in an actual execution of a PFSM the output and local variables can be updated anywhere within the interval $[k \cdot d_{\text{cycle}} + d_{\text{phase}}, k \cdot d_{\text{cycle}} + d_{\text{phase}} + d_{\text{WCTT}}]$, the actual execution time of a transition does not affect the behavior of other PFSMS. The design constraints require PFSMs to read outputs of another PFSMs only after they have become stable, i.e., after the WCTT has elapsed. This modeling approach significantly simplifies the model and helps in inhibiting the state explosion problem.

7.2 Example

In the following, we will explain the model checking of an example system specified using PFSMs in SAL. The example system, which is depicted in Figure 7, is a control loop of an application controlling the revolutions per minute.

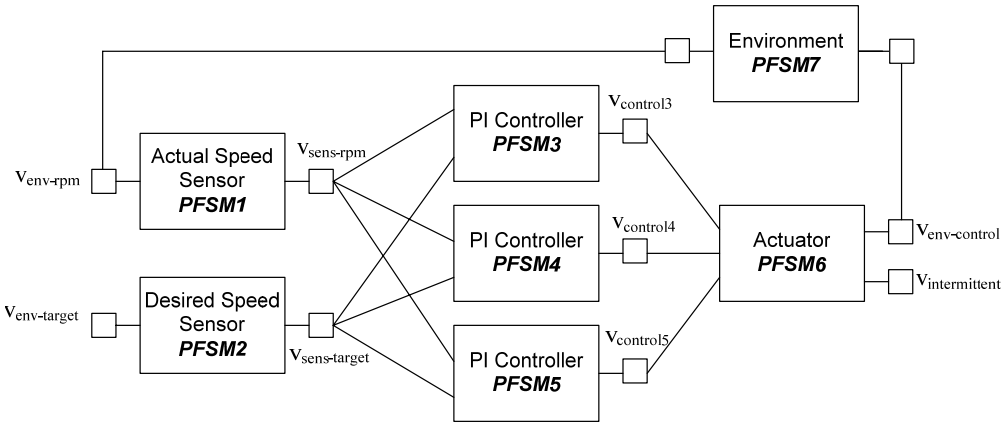

Fig. 7: PFSM Example

The example consists of seven PFSMs. Two PFSMs model the sensors to acquire the actual speed and the desired speed from the environment. The PI controller is realized using three PFSMs that form a TMR configuration. Each of the three PFSMs receives the sensed actual and desired speed and computes a control value. An actuator PFSM (i.e., PFSM6) receives the three control values and performs voting based on a bit-by-bit comparison. The foundation for being able to perform this bit-by-bit is the TMR determinism of the three replicas modeled by PFSMs. In addition, PFSM6 recognizes repeated transient failures for the detection of permanent faults (i.e., so-called intermittent faults). If two failures of the replicas in the TMR configuration are detected within 100 ms (i.e., the output of one PFSM does not match the other two), a counter variable $v_{\text{intermittent}}$ is incremented. The last PFSM (i.e., PFSM7) captures the behavior of the environment based on an environmental model, accepting the control value of the actuator and producing the actual speed for the sensor. PFSM7 accumulates the control values provided by the actuator PFSM. Due to limitations of SAL, which does not support rational numbers, PFSM7 computes the current rpm with a fixed point decimal number (factor 10 in σ_7).

Formally the PFSMs are defined as follows:

$$\text{PFSM1} = \left\langle \{q_0\}, q_0, 10\text{ms}, 0\text{ms}, 1\text{ms}, \left\langle (v_{env-speed}, \{0,1,...,100\text{kmh}\}), (v_{sens-speed}, \{0,1,...,100\text{kmh}\}) \right\rangle, \left\langle (v_{prev}, \{0,1,...,100\text{kmh}\}) \right\rangle, \sigma_1 \right\rangle$$

$$\sigma_1(q,t,(i_a,i_t),h) = q_0, (\lfloor (h+i_a)/2 \rfloor), 5ms, i_a$$

$$\text{PFSM2} = \left\langle \{q_0\}, q_0, 10\text{ms}, 1\text{ms}, 1\text{ms}, \left\langle (v_{\text{env-target}}, \{0,1,...,100\text{kmh}\}) \right\rangle, \left\langle (v_{\text{sens-target}}, \{0,1,...,100\text{kmh}\}) \right\rangle, \langle\ \rangle, \sigma_2 \right\rangle$$

$$\sigma_2(q,t,(i_a,i_t)) = q_0,\ (i_a, 5\text{ms}) \quad \text{// assumption: speed value remains}$$

$$\text{PFSM}[3,4,5] = \left\langle \{q_0\}, q_0, 10\text{ms}, 2\text{ms}, 1\text{ms}, \left\langle (v_{\text{sens-speed}}, \{0,1,...,100\text{kmh}\}), (v_{\text{sens-target}}, \{0,1,...,100\text{kmh}\}) \right\rangle, \right.$$
$$\left. , \left\langle (v_{\text{control}[3,4,5]}, \{-100,...,100\}) \right\rangle, \left\langle (v_{\text{int}[3,4,5]}, \{-100,...,100\}) \right\rangle, \sigma_{[3,4,5]} \right\rangle$$

$$\sigma_{[3,4,5]}(q,t,(i_{a1},i_{t1}),(i_{a2},i_{t2}),h) = q_0,\ (limit(k_i \cdot (h+i_{a2}-i_{a1}) + k_p \cdot (i_{a2}-i_{a1})), -100, 100), (limit(h+i_{a2}-i_{a1}, -100, 100))$$

$$\text{where } limit(n,a,b) = \max(a, \min(b,n))$$

$$\text{PFSM6} = \langle \{q_0, q_1\}, q_0, 10\text{ms}, 3\text{ms}, 1\text{ms}, \langle (v_{\text{control3}}, \{\text{-100},...,100\}), (v_{\text{control4}}, \{\text{-100},...,100\}), (v_{\text{control5}}, \{\text{-100},...,100\}) \rangle, \langle (v_{\text{env-control}}, \{\text{-100},...,100\}), (v_{\text{intermittent}}, \{0,...,100\}) \rangle, \langle (v_{timeout}, \{0,...,100\}) \rangle, \sigma_6 \rangle$$

$$\sigma_6(q, t, (i_{a1}, i_{t1}), (i_{a2}, i_{t2}), (i_{a3}, i_{t3}), h_{\text{intermittent}}, h_{\text{timeout}}) = ($$

$$\begin{cases} (i_{a1} = i_{a2} \wedge i_{a2} = i_{a3}) \wedge (h_{\text{timeout}} = 0) & q_0 \\ \text{else} & q_1 \end{cases},$$

$$(eq(i_{a1}, i_{a2}) \cdot i_1 + eq(i_{a2}, i_{a3}) \cdot i_{a3} + eq(i_{a1}, i_{a3}) \cdot i_{a1}, 5ms),$$

$$(h_{\text{intermittent}}, 5ms),$$

$$\begin{cases} q = q_1 \wedge (i_{a1} = i_{a2} \wedge i_{a2} = i_{a3}) & \max(0, h_{\text{timeout}} - 1) \\ q = q_1 \wedge (i_{a1} \neq i_{a2} \vee i_{a2} \neq i_{a3}) & 100 \\ \text{else} & 0 \end{cases},$$

$$\begin{cases} q = q_1 \wedge (i_{a1} \neq i_{a2} \vee i_{a2} \neq i_{a3})_0 & h_{\text{intermittent}} + 1 \\ \text{else} & h_{\text{intermittent}} \end{cases},$$

$$)$$

$$eq(a, b) = \begin{cases} 1 & a = b \\ 0 & a \neq b \end{cases}$$

$$\text{PFSM7} = \langle \{q_{01}\}, q_0, 10\text{ms}, 5\text{ms}, 1\text{ms}, \langle (v_{\text{env-control}}, \{\text{-100},...,100\}) \rangle, \langle (v_{\text{env-rpm}}, \{0,...,100\}) \rangle, \langle (v_{\text{rpm}}, \{0,...,1000\}) \rangle, \sigma_7 \rangle$$

$$\sigma_7(q, t, (i_a, i_t), h_{\text{rpm}}) = (q_0, h_{\text{rpm}} / 10, \max(\min(h_{\text{rpm}} + i_a \cdot 10, -1000), 1000))$$

Using the control example modeled in SAL, we have formally proved the stability of the control loop. Given any desired rpm as an input, the actual rpm in the control example reaches the desired rpm within 150 ms. The corresponding theorem in SAL with a tolerance of the rpm value of ±3rpm is as follows:

```
th2: theorem system |- AG((time>150) =>
        (((v_envtarget-v_envrpm)<3) AND ((v_envtarget-v_envrpm)>-3)));
```

In order to test the correctness of the fault-tolerance mechanisms in the example, the following faults were included in the model:

- ***Transient fault of the actual speed sensor:*** During the persistence of the transient fault at the actual speed sensor, the invalidation instant of the state variable with actual speed is not updated. The model assumes transient faults at the speed sensor (i.e., PFSM1) with a duration between 1 ms and 15 ms.
- ***Transient fault of one of the replicas:*** In order to test the TMR configuration, the model assumes transient faults at PFSM3 with a duration between 1 ms and 15 ms.

We have formally proved the stability of the control loop in the presence of these faults. Given any desired rpm as an input and one of the above faults, the actual rpm in the control example reaches the desired rpm within 150 ms.

Furthermore, we have visualized the behavior of the control loop for selected values of the input variables. Figure 8 shows the control value, as well as the measured and actual rpm over time for a simulation run with desired value of 39 rpm.

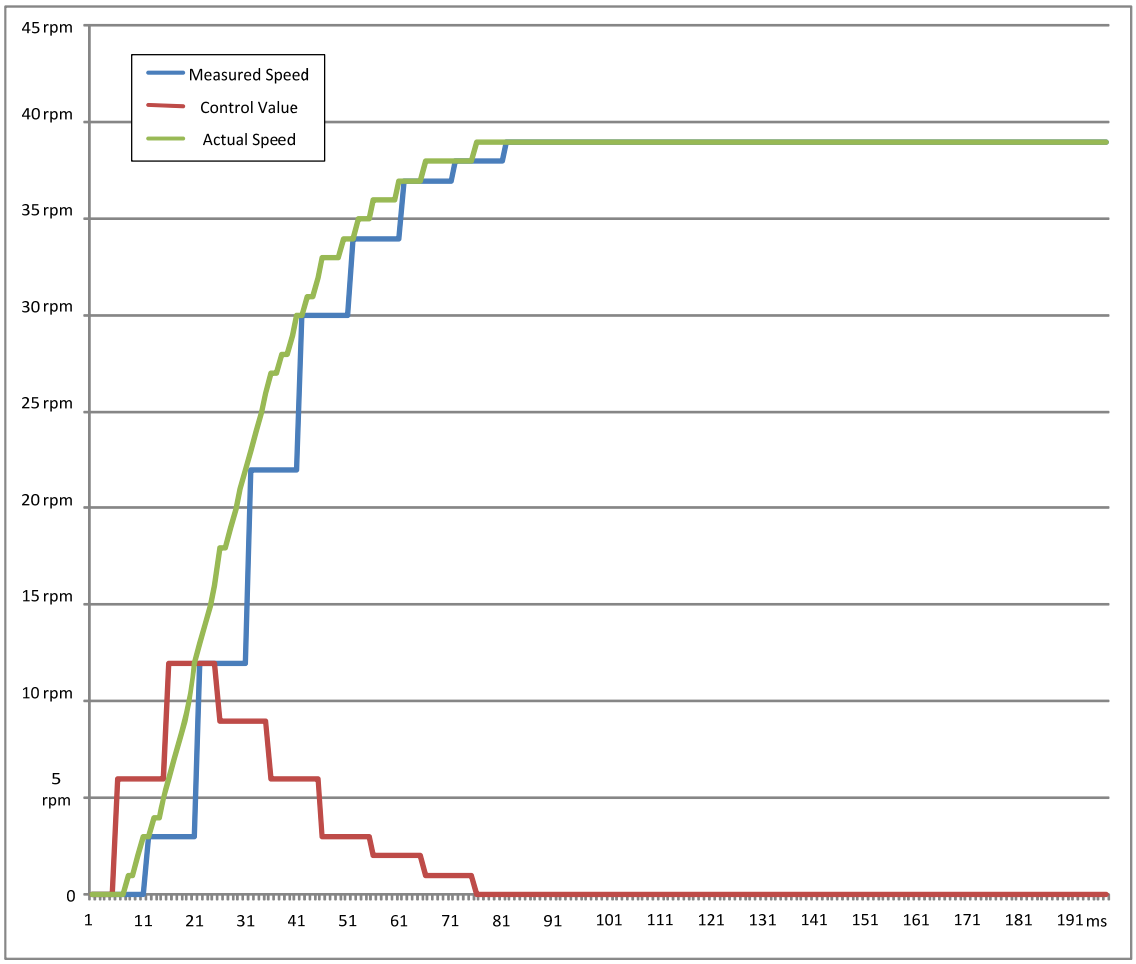

Fig.8: Simulation Run in SAL

8 Discussion

This section discusses the benefits of PFSMs and the ability of formal verification using modeling checking.

- **Expressive Means for Effectively Modeling Distributed Embedded Real-Time Systems.** The proposed PFSMs introduce expressive mechanisms targeted specifically at distributed embedded real-time systems. Systems are modeled as aggregates of periodic computational and communication activities that interact using state variables associated with temporal meta-information. The meta-information denotes when the value of a time-sensitive state variables looses validity due to the progression of time. This information concerning temporal accuracy allows to check during formal verification whether state variables are still valid at the point of use. In addition, PFSMs can model how data transformations influence the temporal accuracy. For example, a PFSM can be used to performing state estimation in order to extend the time interval in which a state variable is temporally accurate.
  Consequently, PFSMs are suited for a broad class of systems, namely systems comprising periodic transactions and using time sensitive data such as control systems, data monitoring systems, or multimedia systems.
- **Determinism and Consistent View on the Distributed State.** The update of the state variables occurs on the action lattice of a global sparse time base. Thereby, the model rules out any problems of *simultaneity and concurrency* and ensures a *consistent view on the distributed state* of the interacting PFSMs. The clear notion of state of the PFSMs is essential for the modeling of fault-tolerant systems with active redundancy. A deterministic behavior of the replicas in a TMR configuration is a prerequisite to perform exact voting on the redundant computational results.
- **Complexity Management.** The time-triggered activation of transitions in conjunction with the WCTT leads to a precise separation of the concerns which relate to the temporal correctness from the concerns

which relate to the correctness of the transformation in the data domain. This separation supports the segmentation of a problem into a set of independently solvable sub-problems as suggested by the Relational Cognitive Complexity Metric of Halford [1].

- **Modeling and Formal Analysis of Complex Real-World Systems.** Model-checking is one of the most successful techniques for the formal verification of finite-state systems. Nevertheless, the memory and execution time required for performing reachability analysis in the state space of real-world models limits the practical use of model checking. Two prominent means for coping with this *state explosion problem* are the introduction of optimizations in model checkers (e.g., partial order reduction) and the simplifications of models. The latter refers to the creation of models that abstracts over details that are not relevant for the formal analysis, while capturing all essential properties. The presented PFSMs provide a contribution in this manner. In particular, we significantly reduce the state space by exploiting a sparse time base and restriction the activation of computational and communication actions to predefined phases.

9 Conclusions

The Periodic Finite State Machine (PFSM) model expands the basic FSM model to include the temporal properties of the physical execution environment. It thus bridges the gap between the l*ogical world of modeling* and the *physical world of execution of an implementation of the model*. This expansion is achieved by assigning execution durations to computational and communication processes and by establishing and enforcing the limits of the temporal validity of the real-time data. The control signals for the initiation of computational and communication actions are derived from the progression of a globally synchronized sparse real-time base, which must be established by the execution environment.

This expansion of the basic FSM model reduces the gap between the modeled system and the implementation of the model on a physical distributed machine and thus improves the predictive powers of the model.

Acknowledgements

This work has been supported, in part by the European Integrated IST Project *DECOS* under project No. IST-511764 and the European IST project *ARTIST2* under project No. IST-004527.

References

1. Halford, G.S., et al., *How Many Variables Can Humans Process?* Psychological Science, 2005. **16**(1): p. 70-76.
2. Alur, A. and D.L. Dill, *A theory of timed automata.* Theoretical Computer Science, 1994. **126**(2): p. 183-235.
3. Berry, G. and L. Cosserat. *The Synchronous Programming Language ESTEREL and its Mathematical Semantics*. in *Proceedings of the Seminar on Concurrency (LNCS 197)*. 1985: Springer-Verlag.
4. Jones, C., et al., *DSOS Conceptual Model*. 2003: University of Newcastle upon Tyne, Techn. Report CS-TR-782, TU Vienna, Technical Report 54/2002, Qinetic Technical Report TR030434, LAAS Technical Report. p. 1-122.
5. Kopetz, H. and W. Ochsenreiter, *Clock Synchronisation in Distributed Real-Time Systems.* IEEE Trans. Computers, 1987. **36**(8): p. 933-940.
6. Kopetz, H. *Sparse Time versus Dense Time in Distributed Real-Time Systems*. in *Proc. 14th Int. Conf. on Distributed Computing Systems*. 1992. Yokohama, Japan: IEEE Press.
7. Mesarovic, M.D. and Y. Takahara, *Abstract Systems Theory*. Lecture Note in Control and Information Science. Vol. 116. 1989: Springer Verlag.
8. Kopetz, H., *Real-Time Systems, Design Principles for Distributed Embedded Applications; ISBN: 0-7923-9894-7, Seventh printing 2003.* 1997, Boston: Kluwer Academic Publishers.
9. Kopetz, H. and K. Kim. *Temporal Uncertainties in Interactions among Real-Time Objects*. in *Proc. 9th Symposium on Reliable Distributed Systems*. 1990. Huntsville, AL, USA: IEEE Computer Society Press.
10. Girault, A., B. Lee, and E. Lee, *A Preliminary Study of Hierarchical Finite State Machines with Multiple Concurrency Models*, in *Electronic Research Laboratory*. 1997, University of California, Berkeley.

11. Bengtsson, J., et al., *Uppaal - a Tool Suite for Automatic Verification of Real-Time Systems*, in *4th DIMACS Workshop on Verification and Control of Hybrid Systems*. 1995: New Brunswick, New Jersey.
12. Hoare, C.A.R., *Communicating Sequential Processes*. 1985: Prentice Hall.
13. Choffrut, C. and M. Goldwurm, *Timed Automata with Periodic Clock Constraints.* Journal of Automata, Languages and Combinatorics, 2000. **5**(4): p. 371-403.
14. Krcal, P., et al., *Timed vs Time-Triggered Automata*, in *Fifteenth International Conference on Concurrency Theory (CONCUR 2004)*. 2004: London, England.
15. Dutertre, B. and M. Sorea, *Modeling and Verification of a Fault-Tolerant Real-time Startup Protocol using Calendar Automata*, in *Formal Techniques in Real-Time and Fault Tolerant System (FTRTFT)*. 2004: Grenoble.
16. Kopetz, H. and G. Bauer, *The Time-Triggered Architecture.* Proceedings of the IEEE, 2003. **91**(January 2003): p. 112-126.
17. Kopetz, H. and R. Nossal. *Temporal Firewalls in Large Distributed Real-Time Systems.* in *Proceedings of IEEE Workshop on Future Trends in Distributed Computing*. 1997. Tunis, Tunesia: IEEE Press.
18. Caspi, P., et al. *LUSTRE, a Declarative Language for Real-Time Programming*. in *Proceedings of the Conference on the Principles of Programming Languages*. 1987. Munich.
19. Kopetz, H. *The Time-Triggered (TT) Model of Computation*. in *Real-Time System Symposium 1998*. 1998. Madrid, Spain: IEEE Computer Society Press.
20. Winfree, A.T., *The Geometry of Biological Time*. 2001: Springer Verlag New York.
21. Puschner, P. and A. Burns, *A Review of Worst-Case Execution Time Analysis.* Journal of Real-time Systems, 2000. **2**(3): p. 115-128.
22. Kopetz, H., *Pulsed Data Streams*, in *IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006)*. 2006, Springer: Braga, Portugal. p. 105-124.
23. Kopetz, H. and N. Suri. *Compositional Design of Real-Time System: A Conceptual Basis for the Specification of Linking Interfaces*. in *ISORC 2003--The 6th International Symposium on Object Oriented Real-Time Computing*. 2003. Hakodate, Japan: IEEE Press.
24. Kopetz, H. *Elementary versus Composite Interfaces in Distributed Real-Time Systems*. in *Proc. of ISADS 99, IEEE Press*. 1999. Tokyo, Japan.
25. Obermaisser, R. and P. Peti. *Specification and Execution of Gateways in Integrated Architectures*. in *10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2005. Catania, Italy: IEEE.
26. Powell, D. *Failure Mode Assumptions and Assumption Coverage*. in *Proc. 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22)*. 1992. Boston, MA, USA: IEEE Computer Society Press.
27. Kahn, G. *The Semantics of Simple Languages for Parallel Prgramming*. in *Proc. of the IFIP Congress* 1974: North Holland Publishing Company.
28. Constantinescu, C. *Impact of Deep Submicron Technology on Dependability of VLSI Circuits*. in *2002 International Conference on Dependable Systems and Networks*. 2002. Washington D.C.
29. Moura, L.d., S. Owre, and N. Shankar, *The SAL Language Manual*. 2003, SRI International.