Journal of Universal Computer Science, vol. 14, no. 12 (2008), 2007-2033
submitted: 1/8/07, accepted: 1/6/08, appeared: 28/6/08 © J.UCS

The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering

Martin Ouimet and Kristina Lundqvist
(Embedded Systems Laboratory
Massachusetts Institute of Technology
Cambridge, MA, 02139, USA
{mouimet, kristina}@mit.edu)

Abstract: In this paper, we present the Timed Abstract State Machine (TASM) language, which is a language for the specification of embedded real-time systems. In the engineering of embedded real-time systems, the correctness of the system is defined in terms of three aspects - function, time, and resource consumption. The goal of the TASM language and its associated toolset is to provide a basis for specification-based real-time system engineering where these three aspects can be specified and analyzed. The TASM language is built on top of Abstract State Machines (ASM) by including facilities for compact and legible specification of non-functional behavior, namely time and resource consumption. The TASM language provides a notation which is well-suited to the specification needs of embedded real-time systems. We begin the presentation of the language with a historical survey on the use of ASM in specifying real-time systems. The core difference between the TASM language and ASM is that steps are inherently durative instead of being instantaneous and steps consume resources. These concepts capture the reality of physical systems in a flexible abstract model. We present the syntax and semantics of the language and illustrate the concepts using an extended version of the production cell case study.

Key Words: Real-Time Systems, Embedded Systems, Specification, Verification, Formal Methods, Abstract State Machines
Category: D.2.1, D.4.7, I.6, D.2.2

1 Introduction

Specification-based engineering, also called model-based engineering, is an approach to the engineering of hardware and software systems where engineering is conducted with the help of models. Models represent high-level abstractions that are used to represent and analyze system designs throughout the engineering lifecycle of the system. Some of the benefits of using models include the ability to analyze system designs during the early stages of the lifecycle, before the system is implemented. The philosophy of specification-based engineering relies on the economics of software engineering where uncovering and fixing defects during the early phases of the engineering lifecycle realizes significant savings in terms of time and cost [Boehm 1981]. If the language used to represent models has formal underpinnings, the use of models can be leveraged through tool support to automate analysis and engineering activities. For example, consistency and completeness were identified as useful properties of specifications that can be verified automatically, as shown in [Heimdahl and Leveson 1996] and in [Ouimet and Lundqvist 2007(c)]. Furthermore, the use of a formal modeling language can automate engineering activities such as test case generation and code generation [Gargantini and Riccobene 2001].

Reactive systems are a special class of systems that typically do not terminate and are in constant interaction with the environment. The correctness of a reactive system is defined as the system exuding correct behavior in its continued interaction with the environment. Furthermore, a reactive system can also be a *real-time* system if the system's reaction needs to occur within an acceptably bounded amount of time. Vehicle controllers, such as avionics systems, are examples of reactive real-time systems. Such controllers are also typically *embedded*, meaning that they are integrated as part of a larger system. An embedded system differs from standalone computer systems since embedded systems cannot be directly manipulated or controlled by the operator. Because embedded systems share functionality with other components, these types of systems are typically limited in the amounts of resources that they can utilize. For example, embedded controllers in the avionics sector are limited in terms of memory and communication bandwidth. The correctness of an embedded reactive real-time system depends on three key factors - functional correctness, adequately bounded response times, and adequately bounded resource utilization. Such systems are also typically of the *safety-critical* nature, meaning that incorrect behavior could result in a serious hazard. For such systems, specification-based engineering can provide desired benefits in terms of predictability of implementation and automation of engineering activities.

Abstract State Machines (ASM) have been used to specify, analyze and verify hardware and software systems at different levels of abstraction [Börger 1995]. Abstract State Machines have also been used to automate engineering activities, including verification using model checkers [Winter 1997] and test case generation [Grieskamp et al. 2002]. The Timed Abstract State Machine (TASM) language [Ouimet et al. 2007] is built on top of ASMs, by presenting a compact syntax that includes facilities for specifying non-functional properties, namely time and resource consumption. The TASM language and its associated toolset, the TASM toolset [Ouimet and Lundqvist 2007(d)], have been used to model and simulate embedded real-time systems in [Ouimet and Lundqvist 2006] and in [Ouimet et al. 2006]. The goal of the TASM language is to provide a specification-based approach to engineer embedded real-time systems, where the three key aspects of system behavior can be specified, analyzed, tested, and traced bi-directionally, from the initial design stages through system maintenance. Furthermore, the ability to evaluate the performance of system designs provides a valuable means to explore design alternatives. In this article, we present the TASM language in the context of past approaches to specify real-time systems using ASMs. The TASM language includes the ability to express resource consumption, an area that has yet to be addressed using ASMs. Furthermore, the TASM language provides a unified formalism that can capture synchronous and asynchronous concurrent systems without modifying the underlying concurrency model. We review past approaches and situate the TASM language as an improved language to specify embedded reactive real-time systems. The use of the language is illustrated through the modeling of an extended version of the production cell case study [Lewerentz and Lindner 1995].

This paper is divided into 5 sections in addition to this Introduction. Section 2 gives related work in the ASM community. Section 3 presents the syntax and semantics of the language. Section 4 describes a mapping between TASM and Timed ASM. Section 5 describes the production cell model in the TASM language. Finally, section 6 recapitulates the contribution of the paper and addresses issues to be covered in future work.

2 Related Work

The approach to incorporate time in the Abstract State Machine (ASM) formalism in the present work incorporates concepts from a variety of previous approaches from the ASM community. In the ASM community, related work has revolved around two main paradigms: instantaneous actions with time constraints, also called *timed constrained ASMs* [Gurevich and Huggins 1996], and durative actions. In time constrained ASMs, all actions are instantaneous but rule guards can contain predicates over an external function called *currtime*, which denotes a wall clock. The *currtime* function is a monotone function which takes no argument and returns a value from the Real domain. This approach has been used to specify and analyze real-time concurrent algorithms such as the Generalized Railroad Crossing Problem [Beauquier et al. 2000], [Beauquier 2002] and the Kermit protocol. The approach presented in this work also contains the *currtime* function, renamed *now*.

In contrast, the TASM language provides facilities to specify the duration of actions performed by the specified system. A similar approach using durative actions has been used in [Börger et al. 1995] to analyze Lamport's bakery algorithm. In this approach, an untimed version of the algorithm is presented and is refined with durative actions. The refinement is shown to preserve the correctness of the untimed version. The approach is based on asynchronous ASM and the notion of partially ordered runs [Gurevich and Rosenzweig 2000]. The durative moves are specified to occur during an open real interval (a, b) where a and b are time values on the global time axis. Using the time specification, the moves of agents are ordered linearly and the requirements of partially ordered runs are extended to include conditions for overlapping moves. In this approach, there is no structured syntax to capture the duration of actions and the analysis of the specification relies on creative proof methods. Furthermore, the moves of agents are specified on the global time axis.

In contrast, the approach adopted in the TASM language follows a durative action paradigm but specifies moves of agents in terms of relative durations of moves. The duration of a run is thus related to the summation of the moves of agents. Furthermore, the concurrency semantics in the TASM approach is related to synchronous multi-agent ASMs since the moves of agents are synchronized using a global system clock. In the TASM language, there are no external functions that are not controlled by an agent of the specification. External functions are included into the behavior of agents that represent the environment. While the lack of external functions might seem counterintuitive to model embedded controllers, the external functions have been replaced by functions controlled by agents representing the environment. In this way, the system can be simulated completely without the need to hardcode the values of external functions since the values in the environment can depend on the behavior of the system. The TASM approach resembles the *real-time controller ASM* where runs are extended with state changes that are occur at *computationally significant real-time moments* [Cohen and Slissenko 2000]. However, the computation of the significant real-time moments is a result of the actions of agents and is not determined a priori.

The production cell case study [Lewerentz and Lindner 1995] is a popular case study to evaluate formal methods. The case study is used in Section 5 as an illustrative example for a specification expressed in the TASM language. The system has previously been modeled and analyzed in details using ASMs in [Börger and Mearelli 1997]. The purpose of the model in this article is to demonstrate how non-functional properties can be included into the production cell model using the TASM language. In our model and analysis, we purposefully shy away from analyzing functional correctness (safety and liveness), since those properties have been heavily studied in [Lewerentz and Lindner 1995] and in [Börger and Mearelli 1997]. By including timing concepts and power consumption in the production cell model the system design can be analyzed for worst-case and best-case behavior, such as Worst-Case Execution Time (WCET) [Englomb et al. 2003]. The ability to analyze performance properties provides the ability to evaluate design alternatives alongside functional correctness.

3 The Timed Abstract State Machine Language

The key difference between the Timed Abstract State Machine (TASM) language and ASM is that steps are *durative* in TASM. In ASM, machine steps are instantaneous. Furthermore, in TASM, durative steps can consume a finite amount of resources. In the case of single agent specifications, the durative steps of the agent dictate the progression of time in the specification. In the case of multi-agent specifications, the durative steps are used to synchronize agents with respect to one another. In TASM, a step is the execution of a rule, which produces an update set. The update set is applied atomically to global state. For the single agent case, the duration of the step, reflected in the update set obtained through a rule execution dictates the progression of time. At the completion of a step, the environment is updated by applying the update set once the step duration has elapsed.

The concept of step is fundamental in the definition of ASM and in computation theory in general since a step defines the atomic unit of progression of an abstract machine. In TASM, the concept of step is augmented with a duration and a set of resources consumed during the step execution to capture the physical reality of embedded real-time systems. This abstract model adequately captures the physical reality of computer systems where steps are never instantaneous. The durations and resource consumptions can be easily modified to capture behavior at different levels of abstraction, to document system assumptions, and to relate models at different levels of abstraction, including non-atomic refinement. In concrete computer systems, the notion of step varies depending on the level of abstraction. For example, a step could be considered a clock cycle, a machine operation, or a statement execution in a high level programming language. Throughout this article, the terms *step*, *rule execution*, *move of an agent*, and *action of an agent* are used interchangeably.

3.1 Target Systems

The target systems that are targeted by the TASM language are embedded real-time systems. These systems include embedded controllers that monitor the environment periodically, through sensors, and take action on the environment through actuators. The important characteristics of such systems is that the values of sensors as read by the system are directly related to the actions taken by the system. Consequently, the behavior of the environment, typically represented as *external* functions in previous ASM approaches [Cohen and Slissenko 2000], cannot be hardcoded a priori since they depend on the actions of the controller. Furthermore, in these target systems, the behavior of the system must produce a correct action in an adequately bounded amount of time. The *end-to-end latency* of the controller is of special interest in avionics systems. End-to-end latency refers to the maximum time that it takes for the controller to produce a corrective action in response to a certain environmental change. In an embedded controller, the end-to-end latency depends on the frequency of sensor readings, the execution time of the control software, and the dynamics of the actuators.

Verifying the end-to-end latency of controllers during the design stages can provide valuable insight into a system designs [Ouimet and Lundqvist 2007(b)].

3.2 Preliminaries

The subset of ASM included in the TASM language is the same as explained in [Winter 1997], which includes conditional statements and assignments, but excludes the *forall* construct and the *choose* construct. The *forall* statement is excluded because the duration of this construct depends on dynamic conditions and cannot be statically assigned. The *choose* construct is omitted for similar reasons because it is counterintuitive to assign a static duration to non-deterministic choice. The TASM language also excludes the *import* construct because safety-critical real-time systems discourage dynamic allocation. The omission of these three constructs is not too restrictive since many ASM specifications have not used these constructs, e.g., the production cell system in [Börger and Mearelli 1997]. The syntactic structure of a machine in the TASM language is an ASM in *block* form [Grieskamp et al. 2002]. In this form, a machine is structured into a finite set of rules, written in precondition-effect style. Conceptually, block form is convenient for structuring specifications and analysis but it is not necessary since any ASM can be converted to block from by introducing a program counter variable. For a TASM that contains n rules, a block machine has the following structure:

$$\begin{aligned} R_1 &\equiv if\ cond_1\ then\ effect_1 \\ R_2 &\equiv if\ cond_2\ then\ effect_2 \\ &\vdots \\ R_n &\equiv if\ cond_n\ then\ effect_n \end{aligned} \tag{1}$$

The guard $cond_i$ is the condition that needs to be enabled for the effect of the rule, $effect_i$, to be applied to the environment. The effect of the rule is grouped into an *update set*, which is applied atomically to the environment at each computation step of the machine.

3.3 Syntax

The syntax of the TASM language includes the syntax of ASM, including arithmetic and the *now* external function. The TASM language extends the block form of equation 1 to include time and resource consumption. The specification of time and resource consumption is achieved through annotations of individual rules. The concrete syntax of TASM resembles the syntax presented in

[Gargantini and Riccobene 2001], with extensions for time and resource annotations. A sample rule of a block TASM is shown in Listing 1, expressed in the concrete syntax of the TASM language. The rule describes the behavior of the feed belt from the production cell case study [Lewerentz and Lindner 1995]. For a description of the production cell system and a graphical representation of its layout, the reader is referred to Section 5. The feed belt carries blocks from the loader to the robot. According to the description of the system from Section 5, moving a block from the loader to the robot takes 5 time units and consumes 500 units of power.

Listing 1 Rule 1 of Main Machine Feed

```
R1: Block goes to end of belt
{
  t     := 5;
  power := 500;

  if feed_belt = loaded and feed_begin = True and
     motor_feed = on and motor_feed_p = positive then
    feed_begin  := False;
    feed_end    := True;
}
```

3.3.1 Time

The TASM approach to time specification is to specify the duration of a rule execution. In the TASM world, this means that each step will last a finite amount of time before an update set is applied atomically to the environment. Syntactically, time gets specified for each rule in the form of an annotation. The annotation is specified as an interval $[t_{min}, t_{max}]$. The lack of a time annotation for a rule is assumed to mean $t = 0$, an instantaneous rule execution. Semantically, a time annotation is interpreted as a closed interval over $\mathbb{R}_{\geq 0}$. For a given run, a rule execution will last an amount t_i where t_i is taken non-deterministically from the interval $[t_{min}, t_{max}]$. The approach uses relative time between steps since each step will have a finite duration. The total time for a run of a given machine is simply the summation of the individual step times over the run. A special time annotations, denoted "*t := next*", is used to denote that a given machine will not produce update sets until another agent executes a move. This construct is a way to ensure that a machine does not terminate, even if none of its rules are enabled.

3.3.2 Resources

The specification of non-functional properties includes timing characteristics as well as resource consumption properties. A *resource* is defined as a global quantity that has a finite size. Power, memory, and communication bandwidth are examples of resources. Resources are used by the machine when the machine executes rules. Similarly to time specification, syntactically, each rule specifies how much of a given resource it consumes as an annotation. The annotation is specified as an interval $[rr_{min},\ rr_{max}]$. The omission of a resource consumption annotation is assumed to mean zero resource consumption. Semantically, a resource annotation is interpreted as a closed interval over $\mathbb{R}_{\geq 0}$. For a run, for each resource, a rule execution will consume an amount rr_i where rr_i is taken non-deterministically from the interval $[rr_{min},\ rr_{max}]$. The semantics of resource usage are assumed to be *volatile*, that is, usage lasts only through the step duration. For example, if a rule consumes 128 kiloBytes of memory, the total memory usage will be increased by 128 kiloBytes during the step duration and will be decreased by 128 kiloBytes after the update set has been applied to the environment. Time elapses and resources are consumed only when a rule is executed. Determining whether a given rule is activated is instantaneous and consumes no resources.

3.3.3 Abstract Syntax

In the abstract syntax of the TASM language, a specification $ASMSPEC$ is a pair:

$$ASMSPEC = \langle E, ASM \rangle$$

Where:

- E is the environment, which is a triple:

$$E = \langle EV, TU, ER \rangle$$

  Where:

  - EV denotes the *Environment Variables*, a set of global typed variables
  - TU is the *Type Universe*, a set of types that includes:
    - Reals: $RVU = \mathbb{R}$
    - Integers: $NVU = \{\ldots, -1, 0, 1, \ldots\}$
    - Boolean constants: $BVU = \{True,\ False\}$
    - User-defined types: $UDVU$
  - ER is the set of named resources:

  - $ER = \{(rn,\ rs) \mid rn$ is the resource name, and rs is the resource size, a value $\in \mathbb{R}_{\geq 0}\}$

- ASM is the machine, which is a triple:

$$ASM = \langle MV, CV, R \rangle$$

Where:

- MV is the set of *Monitored Variables* $= \{mv \mid mv \in EV$ and mv is read-only in $R\}$ [Börger and Stärk 2003]
- CV is the set of *Controlled Variables* $= \{cv \mid cv \in EV$ and cv is read-write in $R\}$ [Börger and Stärk 2003]
- R is the set of *Rules* $= \langle\ n,\ t,\ RR,\ r\ \rangle$ Where:
  - n is the name of the rule
  - t is the duration of the rule execution, a closed interval over $\mathbb{R}_{\geq 0}$
  - RR is the set of resources used by the rule where each element is of the form $(rr,\ ra)$ where $rr \in ER$ is the resource name and ra is the resource amount consumed, specified as a closed interval on $\mathbb{R}_{\geq 0}$
  - r is a rule of the form *if C then A* where C is an expression that evaluates to an element in BVU and A is an action$\}$

An action A is a sequence of one or more updates to environment variables, also called an *effect expression*, of the form $v := vu$ where $v \in CV$ and vu is an expression that evaluates to an element in the type of v.

3.3.4 Hierarchical Composition

In complex systems, structuring mechanisms are required to partition large specifications into smaller ones. The partitioning enables bottom-up or top-down construction of specifications and creates opportunities for reuse. The composition mechanisms included in the TASM language are based on the XASM language [Anlauff 2000]. In the XASM language, an ASM can use other ASMs in rule effects in two different ways - as a *sub* ASM or as a $function$ ASM. A *sub* ASM is a machine that is used to structure specifications hierarchically, similar to a *Turbo ASM* [Börger et al. 2001]. A $function$ ASM is a machine that takes a set of inputs and returns a single value as output, similarly to a function in programming languages, and similar to an ASM *macro* [Börger et al. 2001]. These two concepts enable abstraction of specifications by hiding details inside of auxiliary machines.

The definition of a sub ASM is similar to the previous definition of machine ASM:

$$SASM = \langle n, MV, CV, R \rangle$$

Where n is the machine name, unique in the specification, and other tuple members have the same definition as mentioned in previous sections. The execution and termination semantics of a sub ASM are different than those of a main ASM. When a sub ASM is invoked, one of its enabled rules is selected, it yields an update set, and it terminates.

The definition of a function ASM is slightly different. Instead of specifying monitored and controlled variables, a function ASM specifies the number and types of the inputs and the type of the output:

$$FASM = \langle n, IV, OV, R \rangle$$

Where:

- n is the machine name, unique in the specification
- IV is a set of named inputs (ivn, it) where ivn is the input name, unique in IV, and $it \in TU$ is its type.
- OV is a pair (ovn, ot) specifying the output where ovn is the name of the output and $ot \in TU$ is its type
- R is the set of rules with the same definition as previously stated, but with the restriction that it only operates on variables in IV and OV.

A function ASM cannot modify the environment and must derive its output solely from its inputs. The only side-effect of a function ASM is time and resource consumption.

A specification, $ASMSPEC$, is extended to include the auxiliary ASMs:

$$ASMSPEC = \langle E, AASM, ASM \rangle$$

Where:

- E is the environment
- $AASM$ is a set of auxiliary ASMs (both sub ASMs and function ASMs)
- ASM is the main machine

3.3.5 Parallel Composition

To enable specification of multiple parallel activities in a system, the TASM language allows parallel composition of multiple abstract state machines. Parallel composition is enabled through the definition of multiple top-level machines, called *main* machines, analogous to multiple agents [Börger et al. 2001]. Formally, the specification $ASMSPEC$ is extended to include a set of main machines $MASM$ as opposed to the single main machine ASM for basic ASM specifications:

$$ASMSPEC = \langle E, AASM, MASM \rangle$$

Where:

- E is the environment
- $AASM$ is a set of auxiliary ASMs (both sub ASMs and function ASMs)
- $MASM$ is a set of main machines ASM that execute in parallel

The definition of a main machine ASM is the same as from previous sections. Other definitions also remain unchanged.

3.4 Semantics

The semantics of the TASM language extend the update set concept with time and resource consumption. Updates to environment variables are organized in *steps*, where each step corresponds to a *rule execution*. In the rest of this paper, the terms *step execution* and *rule execution* are used interchangeably. The state is defined at all time instants. A rule is *enabled* if its guarding condition, C, evaluates to the boolean value $True$. The *update set* for the i^{th} step, denoted U_i, is defined as the collection of all updates to controlled variables for the step. An update set U_i will contain 0 or more pairs *(cv, v)* of assignments of values to controlled variables. A *run* of a basic ASM is defined by a sequence of update sets.

3.4.1 Update Set

In TASM, when a machine executes a step, the update set that is produced contains the duration of the step, as well as the amounts of resources that are consumed during the step execution. We use the special symbol $\perp$ to denote the absence of an annotation, for either a time annotation or a resource annotation. Update sets are extended to include the duration of the step, $t \in \mathbb{R}_{\geq 0} \cup \{\perp\}$ and a set of resource usage pairs $rc = (rr, rac) \in RC$ where rr is the resource name and $rac \in \mathbb{R}_{\geq 0} \cup \{\perp\}$ is a single value denoting the amount of resource usage for the step. If a resource is specified as an interval, rac is a value selected non-deterministically from the interval.

The symbol TRU_i is used to denote the timed update set, with resource usages, of the ith step of a machine, where t_i is the step duration, RC_i is the set of consumed resources, and U_i is the set of updates to variables:

$$TRU_i = (t_i, RC_i, U_i)$$

For the remainder of this article, the term *update set* refers to an update set of the TRU_i form.

3.4.2 Hierarchical Composition

Semantically, hierarchical composition is achieved through the composition of update sets. A rule execution can utilize sub machines and function machines in its effect expression. Each effect expression produces an update set, and those update sets are composed together to yield a cumulative update set to be applied to the environment. To define the semantics of hierarchical composition, we utilize the semantic domain $\mathbb{R}_{\geq 0} \cup \{\perp\}$. The special value $\perp$ is used to denote the absence of an annotation, for either a time annotation or a resource annotation.

We define two composition operators, $\otimes$ and $\oplus$, to achieve hierarchical composition. The $\otimes$ operator is used to perform the composition of update sets produced by effect expressions within the same rule:

$$\begin{aligned} TRU_1 \otimes TRU_2 &= (t_1, RC_1, U_1) \otimes (t_2, RC_2, U_2) \\ &= (t_1 \otimes t_2, RC_1 \otimes RC_2, U_1 \cup U_2) \end{aligned}$$

The $\otimes$ operator is commutative and associative. The semantics of effect expressions within the same rule are that they happen in parallel. This means that the time annotations will be composed to reflect the duration of the longest update set:

$$t_1 \otimes t_2 = \begin{cases} t_1 & \text{if } t_2 = \perp \\ t_2 & \text{if } t_1 = \perp \\ max(t_1, t_2) & \text{otherwise} \end{cases}$$

The composition of resources also follows the semantics of parallel execution of effect expressions within the same rule. The $\otimes$ operator is distributed over the set of resources:

$$\begin{aligned}
RC_1 \otimes RC_2 &= (rc_{11}, \ldots, rc_{1n}) \otimes (rc_{21}, \ldots, rc_{2n}) \\
&= (rc_{11} \otimes rc_{21}, \ldots, rc_{1n} \otimes rc_{2n}) \\
&= ((rr_{11}, rac_{11}) \otimes (rr_{21}, rac_{21}), \ldots, \\
&\quad (rr_{1n}, rac_{1n}) \otimes (rr_{2n}, rac_{2n})) \\
&= ((rr_{11}, rac_{11} \otimes rac_{21}), ... \\
&\quad ((rr_{1n}, rac_{1n} \otimes rac_{2n}))
\end{aligned}$$

In the TASM language, resources are assumed to be *additive*, that is, parallel consumption of amounts r_1 and r_2 of the same resource yields a total consumption $r_1 + r_2$:

$$rac_1 \otimes rac_2 = \begin{cases} rac_1 & \text{if } rac_2 = \bot \\ rac_2 & \text{if } rac_1 = \bot \\ rac_1 + rac_2 & \text{otherwise} \end{cases}$$

Intuitively, the cumulative duration of a rule effect will be the longest time of an individual effect, the resource consumption will be the summation of the consumptions from individual effects, and the cumulative updates to variables will be the union of the updates from individual effects.

The $\oplus$ operator is used to perform composition of update sets between a *parent* machine and a *child* machine. A parent machine is defined as a machine that uses an auxiliary machine in at least one of its rules' effect expression. A child machine is defined as an auxiliary machine that is being used by another machine. For composition that involves a hierarchy of multiple levels, a machine can play both the role of parent and the role of child. To define the operator, we use the subscript p to denote the update set generated by the parent machine, and the subscript c to denote the update set generated by the child machine:

$$\begin{aligned}
TRU_p \oplus TRU_c &= (t_p, RC_p, U_p) \oplus (t_c, RC_c, U_c) \\
&= (t_p \oplus t_c, RC_p \oplus RC_c, U_p \cup U_c)
\end{aligned}$$

The $\oplus$ operator is *not* commutative, but it is associative. The duration of the rule execution will be determined by the parent, if a time annotation exists in the parent. Otherwise, it will be determined by the child:

$$t_p \oplus t_c = \begin{cases} t_c & \text{if } t_p = \bot \\ t_p & \text{otherwise} \end{cases}$$

The distribution of the $\oplus$ operator over the set of consumed resources is the same as for the $\otimes$ operator:

$$\begin{aligned} RC_p \oplus RC_c &= (rc_{p1}, \ldots, rc_{pn}) \oplus (rc_{c1}, \ldots, rc_{cn}) \\ &= (rc_{p1} \oplus rc_{c1}, \ldots, rc_{pn} \oplus rc_{cn}) \\ &= ((rr_{p1}, rac_{p1}) \oplus (rr_{c1}, rac_{c1}), \ldots, \\ &\quad (rr_{pn}, rac_{pn}) \oplus (rr_{cn}, rac_{cn})) \\ &= ((rr_{p1}, rac_{p1} \oplus rac_{c1}), ... \\ &\quad ((rr_{pn}, rac_{pn} \oplus rac_{cn})) \end{aligned}$$

The resources consumed by the rule execution will be determined by the parent, if a resource annotation exists in the parent. Otherwise, it will be determined by the child:

$$rac_p \oplus rac_c = \begin{cases} rac_c & \text{if } rac_p = \bot \\ rac_p & \text{otherwise} \end{cases}$$

Intuitively, the composition between parent update sets and child update sets is such that the parent machine overrides the child machine. If the parent machine has annotations, those annotations override the annotations from child machines. If a parent machine doesn't have an annotation, then its behavior is defined by the annotations of the auxiliary machines it uses. These semantics enables the abstraction of timing analysis common in the real-time community [Englomb et al. 2003] where program units, such as function calls, are annotated with timing bounds that override child units, without analyzing the underlying behavior of the units. Furthermore, these semantics enable bottom up construction of specifications where the timing behavior can be defined by as the sum of the parts. The hierarchical composition semantics maintain the semantics of ASM where everything that occurs within a step happens in parallel. As in the case of ASM, conflicting updates to variables yield update set inconsistency.

Figure 1 shows a hierarchy of machines for a sample rule execution. Each numbered square represents a machine. Machine "1" represents the rule of the main machine being executed; all other squares represent either sub machines or function machines used to derive the update set produced by the main machine. Machine "3" is an example of a machine that plays the role of parent (of machine "7") and child (of machine "1").

Each machine generates an update set TRU_i, where i is the machine number. The derivation of the produced update set is done in a bottom-up fashion, where TRU_{ret} is the update set returned by the main machine:

$$\begin{aligned} TRU_{ret} = TRU_1 \oplus (\quad & (TRU_2 \oplus (TRU_5 \otimes TRU_6))\otimes \\ & (TRU_3 \oplus TRU_7)\otimes \\ & TRU_4) \end{aligned}$$

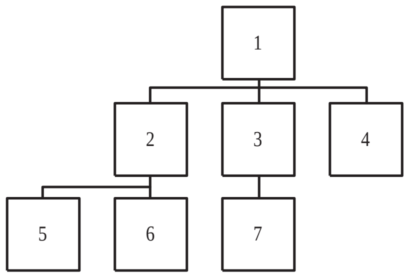

Figure 1: Hierarchical composition

3.4.3 Parallel Composition

The semantics of parallel composition regards the synchronization of the main machines with respect to the global progression of time. We define *tb*, the global time of a run, as a monotonically increasing function over $\mathbb{R}_{\geq 0}$. Machines execute steps that last a finite amount of time, expressed through the duration t_i of the produced update set. The *time of generation*, tg_i, of an update set is the value of tb when the update set is generated. The *time of application*, ta_i, of an update set for a given machine is defined as $tg_i + t_i$, that is, the value of tb when the update set will be applied. A machine whose update set, generated at global time tg_p, lasts t_p will be *busy* until $tb = tg_p + t_p$. While it is busy, the machine cannot perform other steps. In the meantime, other machines who are not busy are free to perform steps. This informal definition gives rise to update sets no longer constrained by step number, but constrained by time. Parallel composition, combined with time annotations, enables the specification of both synchronous and asynchronous systems.

We define the operator $\odot$ for parallel composition of update sets. For a set of update sets TRU_i generated during the same step by i different main machines:

$$\begin{aligned} TRU_1 \odot TRU_2 &= (t_1, RC_1, U_1) \odot (t_2, RC_2, U_2) \\ &= \begin{cases} (t_1, RC_1 \odot RC_2, U_1) & \text{if } t_1 < t_2 \\ (t_2, RC_1 \odot RC_2, U_2) & \text{if } t_1 > t_2 \\ (t_1, RC_1 \odot RC_2, U_1 \cup U_2) & \text{if } t_1 = t_2 \end{cases} \end{aligned}$$

The operator $\odot$ is both commutative and associative. The parallel composition of resources is assumed to be additive, as in the case of hierarchical composition using the $\otimes$ operator:

$$\begin{aligned} RC_1 \odot RC_2 &= (rc_{11}, \ldots, rc_{1n}) \odot (rc_{21}, \ldots, rc_{2n}) \\ &= (rc_{11} \odot rc_{21}, \ldots, rc_{1n} \odot rc_{2n}) \\ &= ((rr_{11}, rac_{11}) \odot (rr_{21}, rac_{21}), \ldots, \\ &\quad\ (rr_{1n}, rac_{1n}) \odot (rr_{2n}, rac_{2n})) \\ &= ((rr_{11}, rac_{11} \odot rac_{21}), ... \\ &\quad\ ((rr_{1n}, rac_{1n} \odot rac_{2n})) \end{aligned}$$

The parallel composition of resources is assumed to be additive, as in the case of hierarchical composition using the $\otimes$ operator:

$$rac_1 \odot rac_2 = \begin{cases} rac_1 & \text{if } rac_2 = \bot \\ rac_2 & \text{if } rac_1 = \bot \\ rac_1 + rac_2 & \text{otherwise} \end{cases}$$

At each global step of the simulation, a list of pending update sets are kept in an ordered list, sorted by time of application. At each global step of the simulation, the update set at the front of the list is composed in parallel with other update sets, using the $\odot$ operator and the resulting update set is applied to the environment. Once an update set is applied to the environment, the step is completed and the global time of the simulation progresses according to the duration of the applied update set.

The concurrency semantics of the TASM language reduce to the concurrency semantics of synchronous and asynchronous multi-agent ASMs. For a TASM specification where all machine steps have the same duration $dt \neq 0$, the specification is essentially a synchronous multi-agent ASM specification with linear time progression. For a TASM specification where all machine steps have the same duration $dt = 0$, the specification is essentially an asynchronous multi-agent ASM specification. In TASM, time plays the role of delaying moves of a machine until the delay of the rule execution has elapsed and acts as a synchronization mechanism.

3.5 Non-Determinism

While the TASM language does not include the *choose* construct from ASM, non-determinism is intrinsic to the TASM language. For example, time and resource annotations can vary non-deterministically. Input/Ouput non-determinism, in terms of assignments to variables, can occur in TASM if one or more rules are enabled for a given step of a given machine. In this case, a rule is chosen non-deterministically from the enabled rules and it is executed. This type of non-determinism differs from ASM where multiple enabled rules are executed within the same step and the update sets are combined. In the TASM language, such semantics would be confusing since durations would have to be added. Furthermore, the ability to non-deterministically chose an enabled rule is convenient when modeling the environment to capture different simulation scenarios. The environment is inherently non-deterministic [Parnas and Madey 1995] and modeling this behavior is paramount to achieve realistic simulation scenarios.

4 Relation to Timed ASM

In [Gurevich and Huggins 1996], the authors present a specification and verification of the railroad crossing problem using a combination of ASM and the *currtime* external function (most recently called *now*). The algebra presented in [Gurevich and Huggins 1996] provides a general approach to timed system modeling. In order to demonstrate that TASM provides a more concise notation, the semantics of the TASM language are expressed using Timed ASM. In order to map a TASM specification into the Timed ASM language, we introduce two domains, namely $DTASM$ and $DASM$ to denote the domains of specifications expressed in the TASM language and the ASM language respectively. We also introduce a function *Desug* that maps a TASM specification into an ASM specification. The "desugaring" function is defined for all individual elements of the TASM language (specifications, variables, types, rules, etc.) and maps the TASM elements into elements of the ASM language.

$$\mathit{Desug}: \; DTASM \rightarrow \; DASM$$

Each resource definition, $Rdef$, in the environment is desugared into a global shared dynamic function:

$$\mathit{Desug}[[\; Rdef \;]] = \; \textbf{shared} \; Rdef$$

Type definitions, $Tdef$ get desugared into static finite domains:

$$\mathit{Desug}[[\; Tdef \;]] = \; \textbf{static domain} \; Tdef$$

Controlled and monitored variables inside of machines get desugared into nullary controlled and dynamic functions, respectively.

The desugaring of the rules is the most complex desugaring in the TASM language, because this is where time and resource utilization play a role. To illustrate the desugaring of rules, we define abstract syntax for a rule definition:

- $Rules = (R_i^+)$

– $R_i = (t_i\ r_i^*\ if\ cond_i\ then\ effect_i)$

In the TASM, the set of rules for a given machine is implicitly mutually exclusive. In the ASM language, the mutual exclusion is explicit. The desugaring introduces two variables, one to keep the time when the rule application will finish executing and one to denote that the machine is "busy" doing work. We denote these two variables by $tcomp_{fresh}$ and $mbusy_{fresh}$. The $fresh$ underscore is used to indicate that the variable name is introduced by the desugaring and enforces that it does not clash with existing names. Both of these variables also desugar into controlled dynamic functions:

$$\begin{aligned} &Desug[[\ tcomp_{fresh}\ ]] && = \textbf{controlled}\ tcomp \text{ initially } \ \textit{-1} \\ &Desug[[\ mbusy_{fresh}\ ]] && = \textbf{controlled}\ mbusy \text{ initially } \ False \end{aligned}$$

Conceptually, once a rule is triggered, a machine sets the $mbusy$ variable to $True$ and will not do anything until the rule duration has elapsed. Once the rule duration has elapsed, the machine will generate the appropriate update set atomically and will be free to execute another rule. The desugaring of a rule is expressed as:

$$\begin{aligned} Desug[[\ Rule\ ]] \quad &= Desug[[\ (t_i\ r_i^*\ if\ cond_i\ then\ effect_i)\ ]] \\ &= \textbf{if/else if}\ cond_i\ \wedge\ \neg mbusy_{fresh}\ \textbf{then} \\ &\quad mbusy_{fresh} := True; \\ &\quad tcomp_{fresh} := now + getDuration(t_i); \\ &\quad r_{i_{fresh}} := getResourceConsumption(r_i); \\ &\quad \textbf{else if}\ now = tcomp_{fresh} \wedge mbusy_{fresh}\ \textbf{then} \\ &\quad effect_i; \\ &\quad mbusy_{fresh} := False; \\ &\quad tcomp_{fresh} := -1; \\ &\quad r_{i_{fresh}} := 0; \\ &\quad \ldots \end{aligned}$$

The function $getDuration$ is a macro that is created using the condition and the time annotation of the rule. It returns the duration of the rule by selecting a duration non-deterministically from the rule annotation. The introduction of the two auxiliary variables and the time conditions will guarantee that the machine will not produce any update sets and that no other rules will be enabled while the machine is executing a rule. This behavior is exactly the desired behavior to simulate "durative" actions. Function machines are desugared as macros and sub machines are desugared just like main machines and they are "inlined" inside the rule where they are invoked. The desugaring of the "*t := next*" construct is fairly straightforward albeit tedious. It involves caching the state at the beginning of the rule execution and creating an extra rule which compares the cached state to the current state. If there is a mismatch, the machine immediately resumes executing rules. If there is no mismatch, the machine simply waits until there is a mismatch.

The one area that remains to be formally specified is the execution semantics of resources. For each resource that is defined in the environment, an agent is created that is used to sum up all of the resources used by other agents. These new agents, symbolically depicted in Listing 2, are used to ensure that resource usage falls within the specified bounds.

Listing 2 Machine to compute resources

```
Agent $RESOURCE_i$
 controlled $last_{fresh}$ initially 0
 controlled $totalresource_{i_{fresh}}$ initially 0

  if $now = last_{fresh} + dt$ then
  $totalresource_{i_{fresh}} := sum(r_i)$
 else
  if $totalresource_{i_{fresh}} > resource_{i_{max}}$ then
  $RESOURCE\_EXHAUSTED$
```

The role of the sum macro is to sum up all of the resource annotations from executing agents. The $RESOURCE\_EXHAUSTED$ macro simply halts execution to note that a given resource has been exhausted.

5 Example

The production cell system is an industrial case study that has been used to evaluate formal methods [Lewerentz and Lindner 1995]. The system is based on an industrial metal processing plant near Karlsruhe in Germany. The production cell consists of a series of components that need to be coordinated to achieve a common goal of stamping metal blocks. Blocks come into the system as "raw" and must leave the system as "stamped". Some simplifications and extensions have been made to the original problem definition from [Lewerentz and Lindner 1995]. For example, the elevating rotatory table has been omitted. The *traveling crane* has been replaced by a *loader*, which is a component that simply puts blocks on the feed belt. The schematic view of the production cell system is shown in Figure 2.

Blocks are introduced into the system via the *loader*, which puts blocks on the *feed belt*. The feed belt carries blocks from one end of the belt to the other. Once a block reaches the end of the feed belt, the *robot* can pick up the block and insert it into the *press*, where the block is stamped. Once a block has been stamped, the robot can pick up the block from the press and unload it on the *deposit belt*, at which point the stamped block is carried out of the system.

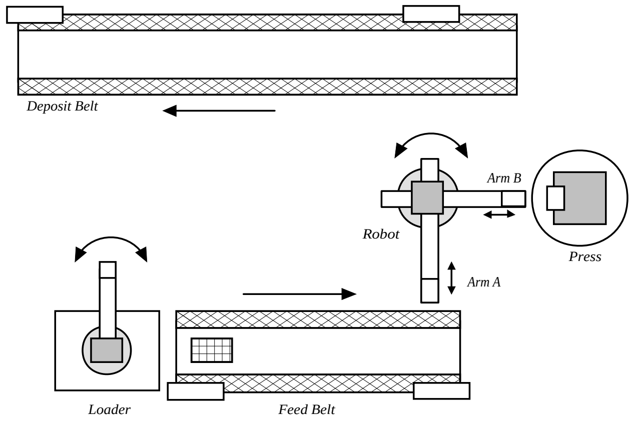

Figure 2: Top view of the production cell

The original example has been extended to reflect the reality that certain actions are *durative*, that is, they take a finite amount of time to complete. For example, the time that it takes for the press to stamp a block could be 11 time units. The example has also been extended to include a resource, *power consumption*. For example, turning on the press motor consumes 1500 units of power per time unit while the press stamps a block. Resources are consumed through the duration of a step and are given back after the step. The list of durative actions, with their power consumptions, are shown in Table 1.

All other actions are assumed to be instantaneous and are assumed to consume no power. The controller actions are assumed to be instantaneous. While this assumption does not reflect reality, it is nevertheless reasonable because the timing of the software is fast enough in relation to the timing of other components. The software operates on the order of micro seconds while the hardware components operate on the order of tenths of a second. This simplification is part of the original case study definition in [Lewerentz and Lindner 1995] and has been formulated as a set of assumptions in [Börger and Mearelli 1997].

The TASM model of the production cell includes a model of the controller and a model of each hardware component. Depending on where the system

Component	Action	Duration	Power
Loader	Put a block on the belt	2	200
Feed	Move block	5	500
Deposit	Move block	7	500
Robot	Rotate 30^o	2	1000
Robot	Extend arm	3	1200
Robot	Retract arm	2	1100
Robot	Drop a block	2	800
Robot	Pickup a block	3	1000
Press	Stamp a block	11	1500

Table 1: Durative actions

boundary is drawn, the hardware components could be considered part of the environment. For the sake of the analysis, the behavior of the hardware components are paramount to the performance of the system and hence are included in the model. The TASM model ensures that the controller can interact with the environment solely through sensors and actuators. The controller reads the state of the various components through a set of sensors and commands the various components through actuators. The state can only be read by the controller through sensors only and actuators can be commanded only by the controller. This convention is congruent with the controller-environment separation principle [Parnas and Madey 1995].

The rules of the loader for the production cell is shown in Listing 3. The loader component is used to "drive" the system by putting blocks on the feed belt when it is empty, up to a pre-determined number of blocks. According to the action description of Table 1, the loader takes 2 time units to load a block into the system and consumes 200 units of power to achieve its task. Once the rule execution completes, the *feed_begin* sensor is set to true. The loader keeps track of how many blocks have been loaded into the system and stops after all blocks have been loaded.

Rule *R1* is used to put the blocks on the feed belt. Rule *R2*, is used to wait and elapse time until the next state change. The "*t := next*" construct is used to keep the machine alive until the next state change. Once all blocks have been loaded in the system, no rule will be enabled for the Loader machine and the machine will stop. The semantics of the "*t := next*" construct is such that the machine will wait until another agent makes a move. This construct is necessary to prevent the machine from stopping because no rule is enabled.

The Robot main machine is used to describe the rotation of the base of the robot. The machine, whose rules are shown in Listing 4, uses the *robot_wait* variable to give a chance for the controller to stop the motor before rotation resumes. This behavior could also have been enforced by the use of a communication channel. The Robot main machine differs from other machines described so far because it uses a sub machine called *ROBOT_MOTION*. As a refresher, a

Listing 3 Rules of the Loader main machine

```
R1: The feed belt is empty, put a block on it
{
  t     := 2;
  power := 200;

  if loaded_blocks < number and feed_belt = empty then
    feed_belt       := loaded;
    loaded_blocks   := loaded_blocks + 1;
    feed_begin      := True;
}

R2: The feed belt is loaded, do nothing {
  t     := next;

  if feed_belt = loaded and loaded_blocks < number then
    skip;
}
```

sub machine is a unit of hierarchical composition. The behavior of the main machine is defined in terms of the main machine by merging the update set yielded by the sub machine with update sets yielded by other sub machines, if applicable. For rule *R1*, the updates to variables yielded by the *ROBOT_MOTION* sub machine will be merged with the update to the *robot_wait* variable. Since rule *R1* does not have a time or resource annotation, the duration and resource consumption of the rule execution are defined by the sub machine annotations, if any.

Listing 4 Rules of the Robot main machine

```
R1: do
{
  if robot_wait = False then
    ROBOT_MOTION();
    robot_wait := True;
}

R2: wait
{
  t := next;

  if robot_wait = True then
    robot_wait := False;
}
```

The use of a sub machine can be viewed as a nested *if* statement. Sub machines are nothing more than syntactic sugar to help structure specifications. Nevertheless, they are helpful because they can be reused and they can be analyzed in isolation [Ouimet and Lundqvist 2007(c)]. The rules of sub machine *ROBOT_MOTION* are shown in Listing 5.

In Listing 5, rules *R1* and *R2* are used to rotate the robot clockwise and counter clockwise depending on the polarity of the motor. Per the data in Ta-

Listing 5 Sample rule of the ROBOT_MOTION sub machine

```
R1: rotate clockwise
{
  t         := 2;
  power     := 1000;

  if motor_robot = on and motor_robot_p = negative then
    robot_angle := rotateClockwise();
    armapos     := armPosition(ARM_A_FEED_ANGLE, ARM_A_DEPOSIT_ANGLE,
                               ARM_A_PRESS_ANGLE, rotateClockwise());
    armbpos     := armPosition(ARM_B_FEED_ANGLE, ARM_B_DEPOSIT_ANGLE,
                               ARM_B_PRESS_ANGLE, rotateClockwise());

}
```

ble 1, a rotation of 30 degrees takes 2 units of time to complete and consumes 1000 units of power. Rule *R1* of the sub machine uses two function machines, *rotateClockwise* and *armPosition*.

The Controller main machine is the most complex machine of the model. In a similar fashion as the *Robot* main machine, the Controller machine uses a variable *wait* to wait for a change in the environment before performing an action. For the controller, this waiting is necessary for the controller to give the environment a chance to make progress. Otherwise, since all controller actions are instantaneous, the controller could perform an infinite number of steps before any environment changes happen. In real-time system terms, the Controller main machine can be viewed as a sporadic task which gets released whenever a sensor value changes. The rules of the Controller main machine, shown in Listing 6, make heavy use of sub machines. The semantics of sub machines and hierarchical composition are such that all sub machines operate in parallel and the resulting update sets of each machine are composed with one another. The commanding of all of the actuators are performed independently in parallel.

The rules of the *OPERATE_DEPOSIT* sub machine are shown in Listing 7. The listing shows how the contoller uses sensor values and the actuators to control the system.

The complete production cell model consists of 9 main machines, 3 function machines, and 16 sub machines. Simulation scenarios were designed to process 5 blocks.

5.1 Sample Run

We use Listing 1, Listing 3, and Listing 5 to illustrate the semantics of parallel composition. For the sample run, we do not show the moves of the controller to make the run easier to understand. The run begins in a initial environment partially defined by (($feed\_belt$, *empty*), (*number*, 5), (*loaded_blocks*, 0), (*robot_angle*, 60)). For a sample run using the aforementioned listings, the

Listing 6 Rules of the Controller main machine

```
R1: Issue Commands
{
  if wait = False then
    OPERATE_FEED();
    OPERATE_DEPOSIT();
    OPERATE_ROBOT();
    OPERATE_ARM_A();
    OPERATE_ARM_B();
    OPERATE_PRESS();
    wait := True;
}

R2: Wait for a step
{
  t := next;

  else then
    wait := False;
}
```

Listing 7 Rules of the OPERATE_DEPOSIT sub machine

```
R1: turn on motor
{
  if motor_deposit = off and deposit_begin = True then
    motor_deposit_p     := negative;
    motor_deposit       := on;
}

R2: turn off motor
{
  if motor_deposit = on and deposit_end = True then
    motor_deposit := off;
}

R3: nothing to do
{
  else then
    skip;
}
```

following update sets (simplified) are generated by the three main machines, ordered by the global time at which they are generated t_g:

$$\begin{aligned}
t_g = 0: \; TRU_{Loader,1} &= (2, ((\text{power}, 200)), ((feed\_belt, loaded),\\
&\quad (feed\_begin, True), (loaded\_blocks, 1)))\\
t_g = 0: \; TRU_{Robot,1} &= (2, ((\text{power}, 1000)), ((robot\_angle, 30)))\\
t_g = 2: \; TRU_{Robot,2} &= (2, ((\text{power}, 1000)), ((robot\_angle, 0)))\\
t_g = 2: \; TRU_{Feed,1} &= (5, ((\text{power}, 500)),((feed\_begin, False),\\
&\quad (feed\_end, True)))
\end{aligned}$$

The update sets are generated according to the rules of the different machines. At the beginning of the simulation, the loader will load a block into on the feed belt. At the same time, because the robot is not in a position to pick up a block from the feed belt ($robot\_angle = 0$), the controller will rotate the robot toward the feed belt. Once the feed belt is loaded, the belt is turned on to carry the block to the robot.

The parallel composition of these update sets yields the environment described below. For brevity, only the environment variables whose values are changed are shown. The variables changed by the controller to command the actuators are also omitted. The time t_g denotes the global simulation time:

$$\begin{aligned}
t_g < 2 :& (((\text{power}, 1200)), \emptyset) \\
t_g = 2 :& (((\text{power}, 1200)), ((feed\_begin, True), (robot\_angle, 30), \\
& (feed\_belt, loaded), (loaded\_blocks, 1))) \\
2 < t_g < 4 :& (((\text{power}, 1500)), \emptyset) \\
t_g = 4 :& (((\text{power}, 1500)), ((robot\_angle, 0))) \\
4 < t_g < 5 :& (((\text{power}, 500)), \emptyset) \\
t_g = 5 :& (((\text{power}, 500)),((feed\_begin, False), (feed\_end, True)))
\end{aligned}$$

For the first time interval, the power consumption is the summation of $TRU_{Loader,1}$ and $TRU_{Robot,1}$ update sets. The effect of both update sets occur at $t_g = 2$. In the time interval $2 < t_g < 4$, the power consumption is the summation of $TRU_{Robot,2}$ and $TRU_{Feed,1}$. At $t_g = 4$, the effect of $TRU_{Robot,2}$ occurs. AT $t_g = 5$, the effect of the $TRU_{Feed,1}$ update set occurs.

6 Conclusion and Future Work

In this article, we have presented the Timed Abstract State Machine (TASM) language, which is an extension of Abstract State Machines (ASM). The TASM language extends ASM with durative steps and resource consumption during steps. The TASM language is used to specify the behavior of embedded real-time systems where time and resource consumption are an integral part of the system's correctness. The TASM language also contains facilities for hierarchical composition and for parallel composition. The TASM language provides an intuitive and well-structured language to capture the three key aspects of embedded real-time systems – function, time, and resources. The language improves on previous attempts to apply ASM to real-time systems since it provides a concurrency model that can express both synchronous and asynchronous behavior. Furthermore, using the duration paradigm, the timing properties of system models can be expressed naturally. The structure of the language eases tool support and provides a basis for formal analysis of models for performance properties such as worst-case execution time, a property of systems critical in the real-time domain [Englomb et al. 2003].

The TASM language has been introduced to the real-time community in [Ouimet et al. 2007] and has been used to model flagship examples from the embedded domain [Ouimet et al. 2006].

6.1 Future Work

The TASM language is being implemented into a toolset, called the TASM toolset, used for the specification, verification, and validation of real-time systems [Ouimet and Lundqvist 2007(d)]. Future work will utilize the UPPAAL tool suite [Larsen et al. 1997] to verify execution time of TASM models for properties such as end-to-end latency, using a restricted subset of TASM that can be translated to timed automata. Furthermore, TASM models have also been analyzed for completeness and consistency [Ouimet and Lundqvist 2007(c)]. The TASM language and toolset will serve as a tool-supported formal basis for real-time system engineering, as part of a framework that will include formal verification, simulation, and test case generation [Ouimet and Lundqvist 2006].

References

[Anlauff 2000] Anlauff, M.: "XASM - An Extensible, Component-Based Abstract State Machines Language"; International Workshop on Abstract State Machines (ASM 2000)(2000).

[Beauquier et al. 2000] Beauquier, D., Crolard, T., and Slissenko, A.: "A Predicate Logic Framework for Mechanical Verification of Real-Time Gurevich Abstract State Machines: A Case Study with PVS"; University Paris 12, Department of Informatics, Technical Report 00–25 (2000).

[Beauquier 2002] Beauquier, D. and Slissenko, A.: "A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class"; Annals of Pure and Applied Logic, Vol. 113, Num. 1–3, pp. 13–52 (2002).

[Boehm 1981] Boehm, B.W.: "Software Engineering Economics"; Prentice-Hall, Englewood Cliffs, NJ (1981).

[Börger 1995] Börger, E.: "Why Use Evolving Algebras for Hardware and Software Engineering?"; In Proc. of the 22nd Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM '95, Lect. Notes in Comp. Sci., Vol. 1012 (1995).

[Börger et al. 1995] Börger, E., Gurevich, Y., and Rosenzweig, D.: "The Bakery Algorithm: Yet Another Specification and Verification"; Specification and Validation Methods, Oxford University Press, pp. 231–243 (1995).

[Börger et al. 2001] Börger, E.: "The Origins and the Development of the ASM Method for High Level System Design and Analysis"; Journal of Computer Science, Vol. 8, Num. 5 (2001).

[Börger and Mearelli 1997] Börger, E. and Mearelli, L.: "Integrating ASMs into the Software Development Life Cycle"; Journal of Universal Computer Science, Vol. 3, Num. 5 (1997).

[Börger and Stärk 2003] Börger, E. and Stärk, R.: "Abstract State Machines"; Abstract State Machines, Springer-Verlag (2003).

[Cohen and Slissenko 2000] Cohen, J. and Slissenko, A.: “On Verification of Refinements of Asynchronous Timed Distributed Algorithms”; International Workshop on Abstract State Machines (ASM 2000), pp. 100–114 (2000).

[Englomb et al. 2003] Englomb, J., Ermedahl, A., Nolin, M., Gustafsson, J., and Hansson, H.: “Worst-Case Execution-Time Analysis for Embedded Real-Time Systems”; Intl. Journal on Software Tools for Technology Transfer, Vol. 4, pp. 437–455 (October 2003).

[Gargantini and Riccobene 2001] Gargantini, A. and Riccobene, E.: “ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation”; Journal of Universal Computer Science, Vol. 7, Num. 11 (2001).

[Grieskamp et al. 2002] Grieskamp, W., Gurevich, Y., Schulte, W., and Veane, M.: “Generating Finite State Machines From Abstract State Machines”; In Proc. of the 2002 ACM SIGSOFT Intl. Symp. on Software Testing and Analysis, pp. 112–122 (2002).

[Gurevich and Huggins 1996] Gurevich, Y. and Huggins, J.: “The Railroad Crossing Problem: an Experiment with Instantaneous Actions and Immediate Reactions”; Selected Papers from CSL ’95, Vol. 1092 of LNCS (1996).

[Gurevich and Rosenzweig 2000] Gurevich, Y. and Rosenzweig, D.: “Partially Ordered Runs: A Case Study”; Abstract State Machines: Theory and Applications, Lect. Notes in Comp. Sci., Vol. 1912, pp. 131–150 (2000).

[Heimdahl and Leveson 1996] Heimdahl, M.P.E. and Leveson, N.G.: “Completeness and Consistency in Hierarchical State-Based Requirements”; Software Engineering, Vol. 22, Num. 6, pp. 363–377 (1996).

[Larsen et al. 1997] Larsen, K.G., Pettersson, P., and Yi, W.: “UPPAAL in a Nutshell”; Intl. Journal on Software Tools for Tech. Trans., Vol. 1, pp. 134–152 (1997).

[Lewerentz and Lindner 1995] Lewerentz, C. and Lindner, T.: “Production Cell: A Comparative Study in Formal Specification and Verification”; KORSO - Methods, Languages, and Tools for the Construction of Correct Software (1995).

[Ouimet 2006] Ouimet, M.: “The TASM Language Reference Manual, Version 1.1”; Available from http://esl.mit.edu/tasm (April 2007).

[Ouimet et al. 2006] Ouimet, M., Berteau, G., and Lundqvist, K.: “Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset”; Models in Soft. Eng., Lect. Notes in Comp. Sci., Vol. 4364 (2006).

[Ouimet et al. 2007] Ouimet, M. and Lundqvist, K.: “The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems”; In Proc. of the 15th International Conference on Real-Time and Network Systems (RTNS ’07) (March 2007).

[Ouimet and Lundqvist 2006] Ouimet, M. and Lundqvist, K.: “The Hi-Five Framework and the Timed Abstract State Machine Language”; In Proc. of the 27th IEEE Real-Time Systems Symposium - Work in Progress Session (December 2006).

[Ouimet and Lundqvist 2007(b)] Ouimet, M. and Lundqvist, K.: “Verifying Execution Time using the TASM Toolset and UPPAAL”; Technical Report ESL-TIK-000212, Embedded Systems Laboratory, Mass. Institute of Tech. (January 2007).

[Ouimet and Lundqvist 2007(c)] Ouimet, M. and Lundqvist, K.: “Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver”; In Proc. of the 3rd Intl. Workshop on Model-Based Testing (MBT ’07), Elec. Notes in Theor. Comp. Sci. (April 2007).

[Ouimet and Lundqvist 2007(d)] Ouimet, M. and Lundqvist, K.: “The Timed Abstract State Machine Toolset: Specification, Simulation, and Verification of Real-Time Systems”; In Proc. of the 19th International Conference on Computer-Aided Verification (CAV’07) (July 2007).

[Parnas and Madey 1995] Parnas, D.L. and Madey, J.: “Functional Documents for Computer Systems”; Science Of Programming, Elsevier (1995).

[Winter 1997] Winter, K.: “Model Checking for Abstract State Machines”; Journal of Universal Computer Science, Vol. 3, Num. 5 (1997).