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. The TASM language is an extension of Abstract State Machines (ASM), that includes facilities for specifying non-functional behavior namely time and resource consumption. In the engineering of 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 key aspects can be specified and analyzed. 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 durative instead of being instantaneous. This paradigm captures the realistic behavior of real-time systems where actions are never instantaneous. The concurrency semantics of TASM is synchronous with respect to durative steps. We present the syntax and semantics of the language and illustrate the concepts using the production cell case study.

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. More specifically, system designs can be analyzed 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 [1]. If the language used to represent models has formal underpinnings, the analysis of models, in order to uncover defects, can be automated. For example, consistency and completeness were identified as useful properties of specifications that can be verified automatically, as shown in [2] and in [3]. Furthermore, the use of a formal modeling language can automate engineering activities such as test case generation and code generation [4].

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. Controllers for vehicles, 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 [5]. Abstract State Machines have also been used to automate engineering activities, including verification using model checkers [6] and test case generation [7]. The Timed Abstract State Machine (TASM) language is an extension of the ASM language that includes facilities for specifying non-functional properties, namely time and resource consumption [8]. The TASM language and its associated toolset [9] have been used to model and simulate real-time systems [10], [11], and [12]. 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 from the initial design stages all the way through system maintenance. In this paper, we present the TASM language in the context of past approaches to specify real-time systems using ASM. We review past approaches and situate the TASM language as an adequate language to specify embedded reactive real-time systems.

This paper is divided into 4 sections in addition to this Introduction. Section 2 gives related work in the ASM community. Section 3 provides concepts and definitions that are used throughout the paper, including the definition of the production cell system used to illustrate the language. Section 4 presents the syntax and semantics of the language. 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 incorporating time in the Abstract State Machine (ASM) formalism in the present work incorporates concepts from a variety of previous approaches in the ASM community. Related work has revolved around two main paradigms: instantaneous actions with time constraints, also called *timed constrained ASM*, and durative actions. In timed constrained ASM, 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 as been used to specify and analyze real-time concurrent algorithms such as the railway crossing problem [13] and the Kermit protocol. The approach presented in this work also contains a function analogous to the *currtime* function, called *now*, but the underlying semantics are highly dependent on the moves of agents being durative.

An approach using durative actions has been used in [14] 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 [15]. 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 contrast, the approach adopted in this work follows a durative action paradigm but specifies moves of agents in terms of relative durations of moves. The duration of a run is thus the summation of the moves of an agent. Furthermore, the concurrency semantics in this approach is related to synchronous multi-agent ASM since the moves of agents are synchronized using a global system clock. In this approach, 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. 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 approach in this work resembles the *real-time controller ASM* where runs are extended with state changes that are occur at *computationally significant real-time moments* [16].

3 Definitions and Concepts

The Timed Abstract State Machine (TASM) language [8] is a formal language for the specification and analysis of real-time systems. The TASM language is an extension of the Abstract State Machine (ASM) language. The TASM language keeps the same basic concepts as the ASM formalism. Like the ASM formalism, the ASM formalism revolves around the concepts of an abstract machine and an abstract state. System behavior is specified as the computing steps of the abstract machine. A computing *step* is the atomic unit of computation, defined as a set of parallel updates made to global *state*. A *state* is defined as the values of all variables at a specific step. A machine *executes* a step by yielding a set of state updates. A *run*, potentially infinite, is a sequence of steps. The subset of ASM included in the TASM language is the same as explained in [6], which includes conditional statements, assignments, but excludes the *for all* construct and the *choose* constructs. The *for all* statement is excluded because the duration of this construct depends on dynamic conditions and cannot be statically assigned. The *choose* construct is omitted because safety-critical real-time systems must be deterministic. The TASM language also excludes the *import* because safety-critical real-time

systems prevent dynamic memory allocation. The structure of a machine in TASM is an ASM in "block form" [7], that is, a finite set of rules, written in precondition-effect style. For a TASM that contains n rules, the machine has the following structure:

$$\begin{aligned} R_1 &\equiv if\ g_1\ then\ e_1 \\ R_2 &\equiv if\ g_2\ then\ e_2 \\ &\vdots \\ R_n &\equiv if\ g_n\ then\ e_n \end{aligned} \tag{1}$$

The guard g_i is the condition that needs to be enabled for the effect of the rule, e_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. For a complete description of the theory of abstract state machines, the reader is referred to [17].

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. A sample controller is shown in Figure 1.

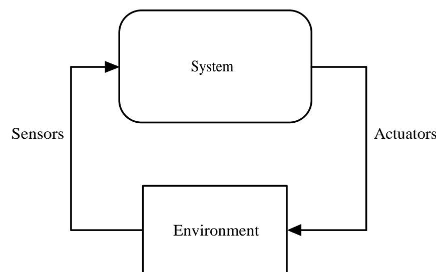

Fig. 1. Sample target system

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, represented as *external* functions, cannot be completely determined a priori since it depends on the dynamics of the system. Furthermore, 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. The end-to-end latency refers to the maximum time that it takes for the controller to produce a corrective action in response to an environment 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's design [18].

3.2 Example

The production cell system is an industrial case study that has been used to evaluate formal methods [19]. 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". 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.

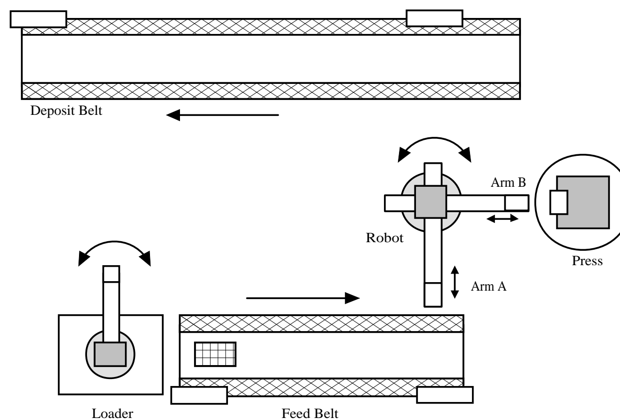

Fig. 2. Top view of the production cell

All components operate concurrently and must be synchronized to achieve the system's goals. The robot has two arms, arm A and arm B, which move in tandem and can pick up and drop blocks in parallel. For example, the robot can drop a block in the press while picking up a block from the feed. To pick up or drop a block, the robot arms must extend and magnets attached to each arm must be turned on and off. A *controller* coordinates the actions of the system by using actuators to operate the various components. Some simplifications and extensions have been made to the original problem definition from [19]. 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 controller reads the state of the various components through a set of sensors and commands the various components through actuators.

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 is 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.

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°	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

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 [19].

4 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 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.

4.1 Syntax

The specification of time and resource consumption is achieved through annotations of individual rules. The concrete syntax of TASM is similar to the syntax of ASM presented in [4], with extensions for time and resource annotations. A sample rule 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 [19]. The

feed belt carries blocks from the loader to the robot. According to the description of the model from Section 3.2, 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;
}
```

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 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 $R\}$
- CV is the set of *Controlled Variables* = $\{cv \mid cv \in EV$ and cv is read-write in $R\}$
- R is the set of *Rules* = $\{(n, r) \mid n$ is a name and 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.

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 to the environment. Syntactically, time gets specified for each rule in the form of an annotation. The specification is specified as an interval $[t_{min}, t_{max}]$. The lack of a time annotation for a rule is assumed to mean $t = 0$. Semantically, a time annotation is interpreted as a closed interval over $\mathbb{R}_{\geq 0}$. A rule execution will last an amount t_i where t_i is taken randomly from the interval. 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.

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. The specification of resource consumption takes the form of an annotation, where the resource usage is specified either as an interval or as a single value. The omission of a resource consumption annotation is assumed to mean zero resource consumption. 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.

Formally, a rule R of a machine ASM is extended to reflect time and resource annotations:

$$R = \langle n, t, RR, r \rangle$$

Where:

- n and r keep the same meaning
- t denotes the duration of the rule execution is 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 $\mathbb{R}_{\geq 0}$

Hierarchical Composition: The examples given so far have dealt only with a single sequential ASM. However, for more 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 language are based on the XASM language [20]. 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. 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. 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

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. 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.

4.2 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. A rule is *enabled* if its guarding condition, C, evaluates to the boolean value $True$. The *update set* for the ith 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.

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 were 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 randomly selected 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 paper, the term *update set* refers to an update set of the TRU_i form.

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.

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:

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

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 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. The hierarchical composition semantics maintain the semantics of ASM in that everything that occurs within a step happens in parallel. As for ASM, conflicting updates to variables yield update set inconsistency.

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:

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

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 multi-agent ASM. For a TASM specification where all machines have the same duration $dt \neq 0$ for all steps, the specification is essentially a synchronous multi-agent ASM specification with linear time progression. For a TASM specification where all machines 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.

4.3 Step

In TASM, a step is the application of an update set. The update set is applied atomically to global state. For the single agent case, the duration of the update sets obtained through hierarchical composition dictates the progression of time. The environment E is updated by applying an update set TRU. For a complete description of the TASM language, the reader is referred to [21].

5 Example

The first rule of the loader for the production cell is shown in Listing 2.

Listing 2 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 - 1 and feed_belt = empty then
    feed_belt     := loaded;
    loaded_blocks := loaded_blocks + 1;
    feed_begin    := True;
}
```

If the rule from Listing 1 and the rule from Listing 2 are executed in parallel with no other rule executing concurrently, the power consumed will be 700 units for $0 \leq$

$now < 2$ and will be 200 units for $2 \leq now < 5$. At $now = 2$, the update set from Listing 2 will be applied and the update set from Listing 1 will be applied at $now = 5$. The complete production cell model is made up of 8 main machines, 3 function machines, and 16 sub machines and has been documented in [12].

6 Conclusion and Future Work

The Timed Abstract State Machine (TASM) language represents 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.

6.1 Future Work

The TASM language is being implemented into a toolset for the specification, verification, and validation of real-time systems [9]. Future work will utilize the UPPAAL model checker [22] to verify execution time of TASM models. Futhermore, the TASM toolset will be extended to be able to automatically generate test cases from TASM specifications.

References

1. Boehm, B.W.: Software Engineering Economics. Prentice-Hall (1981)
2. Heimdahl, M.P.E., Leveson, N.G.: Completeness and Consistency in Hierarchical State-Based Requirements. Software Engineering **22**(6) (1996) 363–377
3. Ouimet, M., Lundqvist, K.: Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. In: Proceedings of the 3rd International Workshop on Model-Based Testing (MBT '07), Satellite Workshop of ETAPS '07. (April 2007)
4. Gargantini, A., Riccobene, E.: Encoding Abstract State Machines in PVS. In: Proceedings of the International Workshop on Abstract State Machines – ASM 2000. Volume 1912 of LNCS., Springer-Verlag (2000)
5. Börger, E.: Why Use Evolving Algebras for Hardware and Software Engineering? In: Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics, SOFSEM '95. Volume 1012 of LNCS., Springer-Verlag (1995)
6. Winter, K.: Model Checking for Abstract State Machines. Journal of Universal Computer Science **3**(5) (1997) 689–701
7. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating Finite State Machines From Abstract State Machines. In: Proceedings of the 2002 ACM SIGSOFT international symposium on Software Testing and Analysis. (2002) 112–122
8. Ouimet, M., Lundqvist, K.: The Timed Abstract State Machine Language: An Executable Specification Language for Reactive Real-Time Systems. In: Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS '07). (March 2007)

9. Ouimet, M., Lundqvist, K.: The Timed Abstract State Machine Toolset: Specification, Simulation, and Verification of Real-Time Systems. In: Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07). (July 2007)
10. Ouimet, M., Berteau, G., Lundqvist, K.: Modeling an Electronic Throttle Controller using the Timed Abstract State Machine Language and Toolset. In: Modeling in Software Engineering. Volume 4364 of LNCS. (2007)
11. Ouimet, M., Lundqvist, K.: The Hi-Five Framework and the Timed Abstract State Machine Language. In: Proceedings of the 27th IEEE Real-Time Systems Symposium - Work in Progress Session. (December 2006)
12. Ouimet, M., Lundqvist, K.: Modeling the Production Cell System in the TASM Language (January 2007) Technical Report ESL-TIK-000209, Embedded Systems Laboratory, Massachusetts Institute of Technology.
13. Beauquier, D., Slissenko, A.: A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class. Annals of Pure and Applied Logic **113**(1–3) (2002) 13–52
14. Börger, E., Gurevich, Y., Rosenzweig, D.: The Bakery Algorithm: Yet Another Specification and Verification. In: Specification and Validation Methods, Oxford University Press (1995) 231–243
15. Gurevich, Y., Rosenzweig, D.: Partially Ordered Runs: A Case Study. In: Abstract State Machines: Theory and Applications. Volume 1912 of LNCS., Springer-Verlag (2000) 131–150
16. Cohen, J., Slissenko, A.: On Verification of Refinements of Asynchronous Timed Distributed Algorithms. In: International Workshop on Abstract State Machines (ASM 2000), Springer-Verlag (2000) 100–114
17. Börger, E., Stärk, R.: Abstract State Machines. Springer-Verlag (2003)
18. Ouimet, M., Lundqvist, K.: Verifying Execution Time using the TASM Toolset and UPPAAL (January 2007) Technical Report ESL-TIK-000212, Embedded Systems Laboratory, Massachusetts Institute of Technology.
19. Lewerentz, C., Lindner, T.: Production Cell: A Comparative Study in Formal Specification and Verification. In: KORSO - Methods, Languages, and Tools for the Construction of Correct Software. (1995)
20. Anlauff, M.: XASM - An Extensible, Component-Based Abstract State Machines Language. In: Abstract State Machines - ASM 2000, International Workshop on Abstract State Machines, TIK-Report 87 (2000)
21. Ouimet, M.: The TASM Language Reference Manual, Version 1.1. Available from http://esl.mit.edu/tasm (November 2006)
22. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. International Journal on Software Tools for Technology Transfer **1** (1997) 134–152