The Specification of Architectural Languages with Abstract State Machines

Ryan Thibodeaux[1]

[1] Institute for Software Integrated Systems of Vanderbilt University, Nashville, Tennesse
rthibodeaux@isis.vanderbilt.edu

Abstract. The descriptions of software systems commonly manifest themselves as Architectural Description Language (ADL) specifications containing the systems constituent components/objects, the behavioral models of the components, and the allowed interaction patterns between components. The semantic anchoring framework uses a similar specification structure as the integral component of a framework for linking operational semantics of common behavioral and interaction patterns, called “semantic units”, to domain-specific modeling language (DSMLs). This paper gives an Abstract State Machine Language (AsmL) specification of a timed automata semantic unit and describes the other necessary components to integrate it into the semantic anchoring framework.

1 Introduction

The *semantic anchoring framework* [4,10] of the Model-Integrated Computing (MIC) tool suite [3] is intended to facilitate the specification of operational semantics of domain-specific modeling languages (DSMLs), pervasive elements of model-based design. Even though a DSML captures concepts and artifacts that are specific to a design’s application context, the behavioral and interaction patterns used in deployed systems are shared across a variety of DSMLs and design spaces. Recognizing these commonalities, a library of “semantic units” is being developed to capture these common forms of operational semantics, e.g. finite state machines, timed automata, dataflow networks, etc., with regard for extensibility and reusability. The main component of a semantic unit description is an architectural language [11] specification of the objects and structures of the semantic unit and the interpretative rules that operate over such entities to provide the behavior and interaction patterns. The semantic anchoring framework leverages the precision and clarity of the Abstract State Machine formalism [8] for these semantic unit descriptions. This paper provides a specification of a semantic unit for timed automata and discusses how to prescribe various forms of operational semantics to systems of composed automata within the semantic anchoring framework.

2 Timed Automata Semantic Unit

Timed Automata [5,6] model the behavior of real-time systems over the progression of time. This formalism extends the definition of finite state transition systems to include a finite set of real-valued clocks that track that passage of time within locations. A valuation function maps the set of clock variables defined in a system to their respective value domain. During the evolution of such a system, clocks can also be reset back to zero by transitions such that they forget the elapsed time since their last reset. The clock variables also further restrict the possible states or behaviors of a system by imposing clock constraints over the enabling of transitions, time guards, and the allowable amount of time that can be spent in a location, invariant conditions. In timed automata, the progression of time is constrained to be synchronous, i.e., time progresses at the same rate across all clocks.

2.1 Timed Automata Overview

A timed automaton of the semantic unit is defined as a 6-tuple $<L, l_0, C, I, E, Pr>$ over an event alphabet Σ and a set of clock constraints $\Phi(C)$:

- *(i)* L is a finite set of locations
- *(ii)* l_0 is an initial location, $l_0 \in L$
- *(iii)* C is a finite set of clock variables
- *(iv)* I maps a clock constraint $\phi \in \Phi(C)$ to each location $l \in L$
- *(v)* $E \subseteq L \times \Sigma \times \Phi(C) \times 2^C \times \Sigma \times L$ is a set of edges. An edge $<l, \alpha, \varphi, \lambda, \omega, l'>$ is a transition from l to l' on the set of input events $\alpha \in \Sigma$ and the satisfaction of the time guard $\varphi(c)$ over the valuation function v of the clocks $c \in C$, $v : C \rightarrow \mathbb{N}_0$. Upon the firing of a transition, the clocks $\lambda \in C$ are reset to 0, and the set of output events $\omega \in \Sigma$ are generated.
- *(vi)* Pr: $E \times \mathbb{N}_0^n \rightarrow \mathbb{N}_0$ is a map that assigns to each edge a priority, a non-negative integer value, with respect to a given clock valuation v, so that $Pr(e, v)$ is the priority of edge e at clock valuation v

Transitions are assigned a priority to model the properties of many popular real-time systems. This mechanism allows for dynamic priority assignment throughout the execution of a model. In this specification, the progression of time is modeled as a transition, and it is assigned a priority of 0, the lowest priority. This supports the notion of urgent transitions [13]; however, this semantic unit does not provide mechanisms for blocking time or most urgent transitions, i.e., the time transition is always enabled given the satisfaction of the current location's invariant condition after the progression of time by some value $\varepsilon \in \mathbb{N}$.

The behavioral model for a timed automaton fitting the structure above defines the state of a timed automaton as a pair of (l, v) where l is a location in L and v is a valuation of the clocks C given that the invariant condition $I(l)$ is satisfied over v. The

possible state changes are enabled over the union of location-switching transitions and the progression of time defined as:

- $(l, v) \xrightarrow{\alpha} (l', v[\lambda := 0])$ iff $\exists\ e = \langle l, \alpha, \varphi, \lambda, \omega, l' \rangle \in E$ given
  - $\varphi(c) = true$,
  - $I(l') = true$,
  - $\forall\ e' = \langle l, \alpha, \varphi', \lambda', \omega', l'' \rangle \in E, \varphi'(c) = true, I(l'') = true \Rightarrow Pr(e,v) \geq Pr(e',v)$
- $(l, v) \xrightarrow{t} (l, v+\varepsilon)$ implies
  - $I(l) = true$ given $v+\varepsilon$,
  - $\forall\ e' = \langle l, \alpha, \varphi', \lambda', \omega', l' \rangle \in E, \varphi'(c) = true, I(l') = true \Rightarrow Pr(e',v) = 0$

The preceding definition for the timed automata semantic unit does not conform to the precise definition of the timed automata formalism [5,6]. This is not the consequence of the formalism's lack of expressiveness or applicability; instead, the semantic unit has to minimally adjust and extend it to fit the requirements imposed by simulation environments and its general applicability to common timed automata languages and tools present in the literature [9,12]. Still, concern for keeping the semantic unit definition as general and precise as possible has been and will continually be a key design principle for all ongoing semantic anchoring work.

In the following, we will use Microsoft Research's Abstract State Machine Language (AsmL) [1] for the specification of the semantic unit. The semantic unit consists of an Abstract Data Model (ADM) and a set of model interpreter rules and methods. The ADM provides the data structures representative of the timed automata Model of Computation (MoC), and the model interpreter methods provide the operational semantics that describe the evolution of an instantiated model fitting the ADM data structures. Using the AsmL tools currently available from Microsoft Research [1] enables the immediate execution of instantiated data models.

2.2 Abstract Data Model

Events and *Clocks* are enumerated stores for the defined events ($e \in \Sigma$) and clock variables ($c \in C$) of an automaton. The predefined elements of *Events* and *Clocks*, *time_ev* and *t*, respectively, are the event generated upon the taking the time transition and the non-resetting system clock.

The *TimedAutomaton* class holds the read-only objects that define an instance of a timed automaton. The *id* field here and in the following class definitions provides a naming construct to identify objects, i.e., it has no semantic implications. The *initial* field holds the initial location of the automaton, and it must be a pre-defined location of the system or an error will occur. *transitions* holds the set of all defined location-switching transitions in the *TimedAutomaton* instance. Concerning time-related entities, *local_clocks* holds the clocks variables of the automaton, and the variable *v* is the valuation of the clock variables in *local_clocks*. The valuation function specifies a mapping from the clock domain, *C,* to the set of non-negative integers, $\mathbb{N}_0$. Integers

were chosen instead of the domain of the positive reals to allow integer domain math functions (such as the modulus operator) to be used on the clock valuations. The variable field *cur* is a 2-tuple holding the *Location* and set of active *Events* in the current configuration of the *TimedAutomaton* during execution.

```
enum Events
time_ev
enum Clocks
t
class TimedAutomaton
const id           as String
const locations    as Set of Location
const initial      as Location
const transitions  as Set of Transition
const local_clocks as Set of Clocks
var v              as Map of Clocks to Integer  = {->}
var cur            as (Location, Set of Events) = (null,{})
```

The *Location* class contains the read-only *id* of the location. Invariant conditions and their mapping to each location are provided via methods in the *TimedAutomaton* class (see Section 2.3). *Transition* captures a move from a location *src* to a location *dst* given the appropriate input event set, *trigger*, and emits the set of events *output*. A transition also specifies the set of clocks to be reset to zero, *resets*, upon taking the transition and the variable *time_guard*, the guard condition $\varphi(c)$, responsible for restricting the enabling of transitions. *time_guard* maps the clock valuations to Booleans. Obviously, since the valuation v of clocks is variable over time, the guard *time_guard* must be a variable over time as well. The progression of time is modeled as a transition, the *time* transition. If the *time* transition is enabled and taken, all clocks in the timed automaton will be incremented by some value $\varepsilon \in \mathbb{N}$ that must be globally defined in the model and the output event *time_ev* is produced.

```
class Location
  const id as String
class Transition
  const id       as String
  const src      as Location
  const dst      as Location
  const trigger  as Set of Events
  const output   as Set of Events
  const resets   as Set of Clocks
  var time_guard as Map of (Map of Clocks to Integer) to Boolean

const time = new Transition("time",null,null,{},{time_ev},{},{->})
```

2.3 Operational Semantics

The operational semantics specification is provided as a set of AsmL rules that execute over input models instantiated using the timed automaton ADM data structures. The *TimedAutomaton* class methods, the global input events function, and an example *Main* rule are provided.

The global function *InputEvents* provides an interface to continue the execution of a model within the AsmL tools. This method takes a *TimedAutomaton* object as input and should return a possibly empty set of *Events* to drive the model execution.

The *InitializeTA* method initializes the *cur* variable to the initial location and an empty active event set and sets the valuation of all local clock variables to zero. The *EnabledTransitionsTA* method returns the set of all enabled transitions that originate from the current location and whose triggering events are currently present given the satisfaction of their guard conditions, *time_guard*(v), and the invariant conditions for all destination locations. The *time* transition is included as an enabled transition if the current location still satisfies its invariant condition following the progression of time.

The *EvolveTA* method fires the transition passed to it, *tr*, by updating the current configuration to be the destination location of the transition and the generated output events of the transition and resetting all clocks specified in the *resets* field of *tr* to zero. The partial update to the mapping v also maintains the current valuations of all other clocks not in *resets, local_clocks – (local_clocks intersect resets)*. The *time* transition is fired by invoking the *TimeProgressTA* method from the *Main* rule. This method updates all clock valuations by the defined integer value *epsilon* and sets the active event set to the output of the time transition, *time_ev*.

The *EvaluateInvariant* method determines if a transition will be enabled by evaluating if the invariant condition of a location is satisfied following the transition. A location switching transition can only be enabled if its destined location's invariant condition is satisfied after the update to the clock valuations, $v[\lambda := 0]$. The *time* transition is enabled only if the current location's invariant condition is satisfied following the clock valuation update $v+\varepsilon$. The *UpdateEvents* and *GetEvents* methods are self explanatory.

The *PriorityTA*, *UpdateTimeGaurdTA*, and *Inv* methods are not predefined functions since they are model dependent. Each must be appropriately specified when simulating an instance model generated from a DSML.

```
InputEvents(ta as TimedAutomaton) as Set of Events
class TimedAutomaton
  InitializeTA()
    require initial in locations
    cur := (initial,{})
    v := {clki -> 0 | clki in local_clocks}

  EnabledTransitionsTA() as Set of Transition
    return {ic | ic in ({et | et in transitions where (et.src =
                         cur.First) and (et.trigger <= cur.Second)
                         and et.time_guard(v)} union {time}) where
                         EvaluateInvariant(ic)}
```

```
EvolveTA(tr as Transition)
  require tr in transitions
  cur := (tr.dst, tr.output)
  v := { clki -> 0 | clki in (tr.resets intersect local_clocks)}
                   union{clki -> v(clki) | clki in local_clocks -
                  (tr.resets intersect local_clocks )}

TimeProgressTA()
  v := {clki -> v(clki) + epsilon | clki in local_clocks}
  cur := (cur.First, time.output)

EvaluateInvariant(tr as Transition) as Boolean
  if tr = time then
    return Inv(cur.First.id,{c -> v(c) + epsilon | c in local_clocks})
  else
    return Inv(cur.First.id,{c -> 0 | c in (tr.resets intersect
                            local_clocks)} union { c -> v(c) | c in
                            local_clocks - (tr.resets intersect
                            local_clocks )})

UpdateEvents(ev as Set of Events)
  cur := (cur.First, ev)

GetEvents() as Set of Events
  return cur.Second

PriorityTA(tr as Transition) as Integer
UpdateTimeGuardTA()
Inv(cur_loc as String, vprime as Map of Clocks to Integer) as Boolean
```

The *PriorityTA* method returns a non-negative integer value for each transition in the system with the lowest priority being 0, that of the *time* transition. The example below shows that all transitions defined in the system, the "_" pattern [1], have a priority of 0, including the *time* transition.

The *UpdateTimeGuardTA* method revaluates the time guard condition of every transition given the current valuation of all clocks within the timed automaton. The example below indicates that transition "t1" will satisfy its guard once the valuation of clock x, $v(x)$, is greater than or equal to 1, transition "t2" is enabled when $v(x)$ is greater than or equal to 2, and transition "t3" is enabled when $v(y)$ is equal to 3. The *time* transition's guard is never evaluated since it is never restricted by a time guard.

The *Inv* method evaluates the invariant condition of each location, given as the location's *id*, over the updated valuation of all clocks, either $v' = v[\lambda := 0]$ or $v' = v+\varepsilon$. In this example, the invariant condition for location "on" is only satisfied when $v'(y)$ is less than or equal to 3, and the invariant conditions for all other locations are always satisfied.

```
class TimedAutomaton
  PriorityTA(tr as Transition) as Integer
    match tr.id
      _  : return 0
  UpdateTimeGuardTA()
    let vt = v
    forall tr in transitions
      match tr.id
        "t1" : tr.time_guard := { vt -> v(x) >= 1 }
        "t2" : tr.time_guard := { vt -> v(x) >= 2 }
        "t3" : tr.time_guard := { vt -> v(y) = 3 }
        _    : skip
  Inv(cur_loc as String, vprime as Map of Clocks to Integer) as Boolean
    match cur_loc
      "off" : return true
      "on"  : return vprime(y) <= 3
      _     : return true
```

Below is a sample *Main* rule for executing a timed automaton instance, *sw1*, defined using the timed automaton ADM. Along with updating the active event set at the beginning of the loop, the time guards for all transitions must be reevaluated. Also, instead of just arbitrarily choosing an enabled transition, the *Main* loop must non-deterministically choose one with the highest priority. If there are no enabled transitions, including the *time* transition, due to restrictions from the invariant conditions, the system is blocked and the execution is stopped.

```
Main()
  step sw1.InitializeTA()
  step while true
    step
      WriteLine("Location: " + sw1.cur.First.id + ", Clocks: " + sw1.v)
      sw1.UpdateEvents(sw1.GetEvents() union InputEvents(sw1))
      sw1.UpdateTimeGuardTA()
    step
      let eT  = sw1.EnabledTransitionsTA()
      if eT = {} then
        error("No enabled transitions, location invariant failure")
      else
        let eT2 = (any h | h in eT where sw1.PriorityTA(h) =
                  (max sw1.PriorityTA(tp) | tp in eT))
        if eT2 = time then
            sw1.TimeProgressTA()
        else
          sw1.EvolveTA(eT2)
```

2.4 Timed Automata Modeling Language

Within the MIC semantic anchoring framework, a model transformation between the metamodel of a given DSML and the timed automata semantic unit (TASU) specifies the anchoring of the DSML to the TASU. In order to connect the AsmL and the MIC tool suites, we must represent the TASU ADM as a MIC metamodel (expressed in MetaGME) and must implement a translator that translates the TASU MIC model instances into AsmL model instances fitting the TASU ADM. Figure 1 shows the metamodel of the TASU specified in the MetaGME metamodeling language [2] of the MIC tool GME.

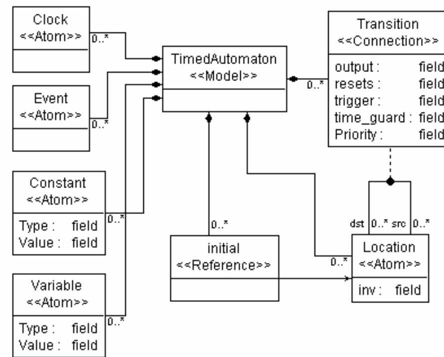

Figure 1 – TASU MetaGME Metamodel

The metamodel in Figure 1 captures the Abstract Data Model of the TASU presented in section 2.2. Note the metamodel includes *Constant* and *Variable* objects for defining model dependent data. The remaining piece of the TASU is a model translator that generates the AsmL representation of data models from timed automata models created in the MIC tool suite. A simple XML parser is created for the semantic unit that takes the XML representation of the TASU instance models from GME and generates the corresponding AsmL ADM representation. The resulting AsmL data model code is then executed in the AsmL tools for comparing against observed behavior from a deployed implementation or testing a design prior to fixing the semantic mapping of a DSML model to an implementation.

Figure 2 shows an example timed automaton for a train approaching a controlled gate commonly used throughout timed automata literature [7]. Location “Far” is the initial location, and inequalities labeling locations are invariant conditions, e.g., $x \leq 5$. Transitions are labeled with a name, “approach”, time guards (if present), $x > 2$, and clock resets, $x := 0$. The automaton indicates that a train may remain “far” from a gate indefinitely; however, the train must proceed completely through a gate and be sufficiently “far” away from the gate within 5 time units once it approaches the gate. The model in Figure 3 is the TASU modeling language representation of the same automaton in GME. This modeling language is defined by the metamodel in Figure 1. The washed-out automaton labeled “Train” is an icon for a timed automaton class in the TASU modeling language. The other box represents the train automaton as it

appears in the modeling language. Single circles are locations, the double circle with the centered "I" indicates the initial location of the automaton, "Far" in this example, the clock icon labeled "x" is a clock variable defined for the automaton, and directed arrows between locations are transitions. The invariant conditions of locations and attributes of transitions like the time guards are provided as attribute code in GME and are visible in a text window not shown here.

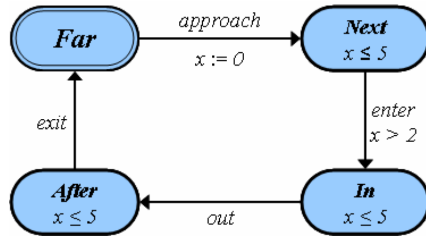

Figure 2 – Example Train Timed Automaton

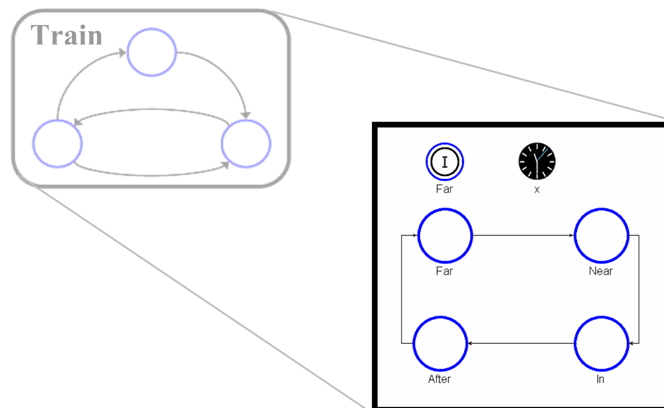

Figure 3 –TASU GME Representation of Train Timed Automaton

The corresponding AsmL data model representation for the train automaton of Figure 3 is provided in Figure 4. This is the direct output of the XML parser previously mentioned. Figure 5 shows a generated segment (from the AsmL tools) of a possible execution trace for the data model representation in Figure 4. The right command prompt window is the continuation of the execution trace started in the left window. The trace shows that the train took the "approach" transition at time 0 and time 6, and it safely proceeded through the gate and achieved the "far" location within 5 and 4 time units, respectively. In both cases, the train also waited 3 time units before taking the "enter" transition due to the restriction of this transition's time guard.

```
const epsilon as Integer = 1
enum Clocks
  x

After = new Location("After")
In    = new Location("In")
Near  = new Location("Near")
Far   = new Location("Far")

enter_tr    = new Transition("enter_tr", Near, In,{},{},{},{->})
approach_tr = new Transition("approach_tr", Far, Near,{},{},{x},{->})
exit_tr     = new Transition("exit_tr", After, Far,{},{},{},{->})
out_tr      = new Transition("out_tr", In, After,{},{},{},{->})

train = new TimedAutomaton("train", {After,In,Near,Far}, Far,
                          {enter_tr,approach_tr,exit_tr,out_tr}, {x,t})
class TimedAutomaton
  PriorityTA(tr as Transition) as Integer
    match tr.id
      "enter_tr"    : return 0
      "approach_tr" : return 0
      "exit_tr"     : return 0
      "out_tr"      : return 0
      _             : return 0
  UpdateTimeGuardTA()
    let vt = (v union g v)
    forall tr in me.transitions
      match tr.id
        "enter_tr"    : tr.time_guard := { vt -> v(x) > 2 }
        "approach_tr" : tr.time_guard := { vt -> true }
        "exit_tr"     : tr.time_guard := { vt -> true }
        "out_tr"      : tr.time_guard := { vt -> true }
        _             : skip
  Inv(cur_loc as String, vprime as Map of Clocks to Integer) as Boolean
    match cur_loc
      "After" : return vprime(x) <= 5
      "In"    : return vprime(x) <= 5
      "Near"  : return vprime(x) <= 5
      "Far"   : return true
      _       : return true
```

Figure 4 – TASU AsmL Representation of Train Timed Automaton

```
C:\Program Files\Spec Explorer\bin\sem...
Starting Execution
Location: {Far}, Clocks: {t->0}{{x->0}}
Firing: (train1,approach_tr), Far -> Near

Location: {Near}, Clocks: {t->0}{{x->0}}

Location: {Near}, Clocks: {t->1}{{x->1}}

Location: {Near}, Clocks: {t->2}{{x->2}}

Location: {Near}, Clocks: {t->3}{{x->3}}
Firing: (train1,enter_tr), Near -> In

Location: {In}, Clocks: {t->3}{{x->3}}

Location: {In}, Clocks: {t->4}{{x->4}}

Location: {In}, Clocks: {t->5}{{x->5}}
Firing: (train1,out_tr), In -> After

Location: {After}, Clocks: {t->5}{{x->5}}
Firing: (train1,exit_tr), After -> Far
```

```
C:\Program Files\Spec Explorer\bin\sem...
Location: {Far}, Clocks: {t->5}{{x->5}}

Location: {Far}, Clocks: {t->6}{{x->6}}
Firing: (train1,approach_tr), Far -> Near

Location: {Near}, Clocks: {t->6}{{x->0}}

Location: {Near}, Clocks: {t->7}{{x->1}}

Location: {Near}, Clocks: {t->8}{{x->2}}

Location: {Near}, Clocks: {t->9}{{x->3}}

Location: {Near}, Clocks: {t->10}{{x->4}}
Firing: (train1,enter_tr), Near -> In

Location: {In}, Clocks: {t->10}{{x->4}}
Firing: (train1,out_tr), In -> After

Location: {After}, Clocks: {t->10}{{x->4}}
Firing: (train1,exit_tr), After -> Far
```

Figure 5 – TASU AsmL Execution Output of Train Automaton

2.5 Composition of Automata

Concerning ASMs, modeling concurrency is approached using multi-agent systems where each agent executes its own ASM [8]. The execution semantics of multi-agent ASMs are not explicitly defined, but synchronous and asynchronous approaches have been studied and documented. In the semantic anchoring work, a similar approach is being investigated as a preliminary step towards a unified framework for investigating and modeling concurrent systems. Since timed automata are most readily used for modeling real-time systems that typically consist of many parallel-acting components, applying multi-agent concepts to the TASU presented above is necessary for specifying industrial-scale systems.

To capture concurrent automata in the TASU, a new system-level class is required for holding all of the objects created using the *TimedAutomaton* class. Within this composed system class, new methods are required to call the simple affecting methods of the *TimedAutomaton* class in parallel for all of the defined automata in the system. This is fairly straightforward for all of the methods except for those that manipulate clocks and the *EvolveTA* methods. Since the automata of the TASU are defined with a set of local clocks scoped to each automaton, the new composed system concept begs the question of whether or not a set of global clocks should be defined. If a set of global clocks is added, methods such as *TimeProgressTA* and *EvolveTA* need to be modified or supplemented in the composed system class methods to update and handle the global clocks, e.g., resetting a global clock specified by a transition of a automaton.

Another major challenge for composing automata in the TASU is how to handle the input and output events of automata. Should all events be global to all automata or should locally-scoped events be allowed? The simplest approach is to allow only global events; therefore, a global event set must be maintained by the composed system class. This event set would be the union of the constituent active event sets of the composed automata denoted as *cur.Second* in the *TimedAutomaton* class of the TASU. Following this addition, the global event set would need to be updated after

taking a transition of a constituent automaton and calling the *InputEvents* method (would also need to be extended to take the set of *TimedAutomaton* objects as input), and each automaton of the composed system would need its active event set equated to the global event set prior to determining its enabled transitions.

Following these additions and all of the necessary modifications to the TASU for handling global clocks, the remaining issue surrounding the composition of automata is same question concerning multi-agent ASMs. Which form of execution semantics to use, synchronous or asynchronous? In order to execute the composed automata in parallel, the composed system class would require a method responsible for globally enacting the steps previously presented in the example *Main* rule for a single automaton of the TASU. Now, the variations in execution semantics arise when choosing how to take transitions of automata in the TASU. In the synchronous case, all automata of the composed system take an enabled transition if possible, and all automata without an enabled transition remain in the same location. Under the same semantics, time is only advanced when all automata agree to take the *time* transition. Asynchronous or interleaving semantics restrict transitions to be enacted for only a single automaton in each execution step. Still, the time is advanced globally in the composed system when all automata agree. This is fairly straightforward; however, the updating of the global event set presents some of its own quandaries under the varying execution semantics. Under both execution semantics, should the global event set should be identical for all automata before any number of transitions is taken in one execution step, and how should the active event set of non-evolving automata should be treated after each transition? One possible implementation would clear the active event set of non-active automata after each step, and another possible implementation could allow the active event set of each non-active automaton to continually persist until it fires one of its respective transitions in some future step.

When composing automata or other agents to model concurrency in the semantic anchoring framework, it currently does not seem appropriate to restrict a semantic unit to exactly one form of execution semantics. Obviously, this is the case since not all real-time systems operate under the same rules and assumptions, and the desired large-scale application of such a modeling formalism requires adaptability. Still, the proper use of the semantic anchoring framework is dictated by the ability of users to grasp possible variations of behaviors and apply them appropriately.

3 Conclusion

Developing an extensive library of the common semantic units coupled with composition techniques required for understanding behaviors and interactions of concurrent components is necessary for industrial-scale applicability of specifying operational semantics for domain specific modeling languages. We see the development of the semantic anchoring framework from the Abstract State Machine formalism as an enabling means of constructing unambiguous yet expressive behavioral specifications like the timed automata semantic unit presented in this work, that will only further advance model-based software development methods for exploring design spaces and alternatives efficiently.

References

1 The Abstract State Machine Language: http://www.research.microsoft.com/fse/asml.

2 The Generic Modeling Environment: http://www.isis.vanderbilt.edu/Projects/gme/.

3 MIC Tool Suite. http://www.escherinstitute.org/Plone/tools/suites/mic

4 Semantic Anchoring Framework: http://www.isis.vanderbilt.edu/SAT.

5 Alur, R. 1999. Timed Automata. In *Proceedings of the 11th international Conference on Computer Aided Verification* (July 06 - 10, 1999). N. Halbwachs and D. Peled, Eds. Lecture Notes In Computer Science, vol. 1633. Springer-Verlag, London, 8-22.

6 Alur, R. and Dill, D. L. 1994. A theory of timed automata. *Theor. Comput. Sci.* 126, 2 (Apr. 1994), 183-235.

7 Beffara, E., Bournez, O., Kacem, H. and Kirchner, C. Verification of timed automata using rewrite rules and strategies, in: N. Dershowitz, A. Frank (Eds.), Proc. BISFAI 2001, Tel Aviv (Israel), June 2001.

8 Borger, E. and Stark, R. F. 2003. *Abstract State Machines: a Method for High-Level System Design and Analysis*. Springer-Verlag New York, Inc.

9 Bozga, M., Graf, S., Ober, I. and Sifakis, J. 2004. Tools and applications II: The IF toolset. In Proceedings of SFM'04, LNCS, volume 3185. Springer-Verlag, 2004.

10 Chen, K., Sztipanovits, J., and Neema, S. 2005. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM international Conference on Embedded Software* (Jersey City, NJ, USA, September 18 - 22, 2005). EMSOFT '05. ACM Press, New York, NY, 35-43.

11 Clements, P. C. 1996. A Survey of Architecture Description Languages. In *Proceedings of the 8th international Workshop on Software Specification and Design* (March 22 - 23, 1996). International Workshop on Software Specifications & Design. IEEE Computer Society, Washington, DC, 16.

12 Larsen, K., Pettersson, P. and Yi, W. 1997. UPPAAL in a nutshell. Int. Journal on Software Tools for Technology Transfer 1(1-2):134-152.