Hierarchical Timed Abstract State Machines for WCET Estimation

Vladimir-Alexandru Paun, Bruno Monsuez, Philippe Baufreton

Hierarchical Timed Abstract State Machines for WCET Estimation

Vladimir-Alexandru Paun
UIIS
ENSTA ParisTech
828, Boulevard des Maréchaux,
91762 Palaiseau Cedex France
paun@ensta-paristech.fr

Bruno Monsuez
UIIS
ENSTA ParisTech
828, Boulevard des Maréchaux,
91762 Palaiseau Cedex France
monsuez@ensta-paristech.fr

Philippe Baufreton
Sagem - SAFRAN Electronics
Etablissement F. Hussenot - R&D
100 avenue de Paris
91344 MASSY Cedex France

In this paper we present an extension of the Abstract State Machines suited for the modelling of complex processors in the context of system verification. Hard real-time systems use evermore elaborate processors in an environment where certification rules are getting tighter and more explicit regarding the verification of software. The goal of our model is to provide a base for worst-case execution time estimation, providing abstraction capabilities that enable the scaling of analysis. The core difference between our model and the others is that we define time as a mean to enable time accurate runs and components at different abstraction levels that can be dynamically chosen during the execution while staying the closest possible to the original mathematical foundation. The model is able to choose the best suited component definition in order to respond to factors like information on data values. The time extension takes into account the fact that actions are not instantaneous which is essential for real-time systems. Adaptable precision and separation of the analysis from the model of the processor, make our model suited for worst-case execution time.

WCET, ASM, hard real-time systems

1. INTRODUCTION

Certification standards, like the ones that can be found in avionics, give precise recommendation regarding the confidence level that functional and non-functional properties must provide. Regarding the non-functional aspects, distinct focus is granted to the bounding of resource consumption. Of particular interest is the timing aspect or the ability to estimate a tight worst-case execution time (WCET) of tasks on a given system. Nevertheless, modern processors have evolved in order to maximise the maximum performance throughput with little to no regard to the determinism of their components. Such modern features influence the instruction timings that can be context or history dependent. Therefore the local worst case no longer suffice in the estimation of the global worst case execution time. In order to safely and precisely estimate the WCET of a processor we need a versatile model that can take into account all the possible component interaction and offer the means to confine and control the inherent state space explosion of exploring all the execution scenarios.

Abstract State Machines (ASMs) have been used with success in processor modelling and verification Huggins and Campenhout (1997), and are a good candidate to describe the underlying architecture for worst-case execution time estimation. Despite the formal background which makes it suited for proofs, the ASM model can be seen as a simple language and used accordingly with a minimum time to take in hand.

Many approaches exist for integrating time into ASMs, however they are either focused on the verification of the correctness of the specification or on the flexibility of design of embedded systems. Based on the richness of ASMs, we create a model better suited for WCET estimation, where a permanent tradeoff between precision and state space explosion can to be made through the selection of component abstraction levels. Our approach is different from others as we use the time information in conjunction with the notion of dynamic turbo-jumps that can vary the duration of the transition from one state to the next one. The temporal model serves as a base to introduce hierarchical levels similar to refinements, extended to run-time refinements, dynamically controlled by the language in order to optimise the aforementioned tradeoff.

Proceedings of . . .

We choose to represent the delay as a semantical information that increments a special location used to store the current time associated to a state. We prefer simplicity and specialisation opposed to versatility as we only target a subset of real-time systems, like processors and closely connected parts of the environment. Special attention is given to preserving the mathematical foundation of the original basic ASM model.

1.1. Related Work

Numerous approaches to integrate time into the ASMs exist in the literature, all for a relatively different purpose. The main directions in the related works are focused around the notion of time as a durative action or as an instantaneous action. Timed ASM with instantaneous actions were first introduced in Gurevich and Huggins (1996). Both paradigms are further developed in Beauquier and Slissenko (2002), Cohen and Slissenko (2008) and Ouimet and Lundqvist (2008) with semantics oriented on verification. In Ouimet and Lundqvist (2008) the Timed ASM is presented as the moves of agents synchronised using a system clock. Their concurrency semantics is based on synchronous multi-agent ASMs. Moves can take time and are associated to durative actions. Time is used to specify the duration of a step and it is chosen non-deterministically from the specified interval. The parallel with the analysis of the system design for worst-case and best-case behavior, such WCET is also made. A detailed presentation of the use of ASMs for precise WCET computation can be found in Benhamamouch and Monsuez (2009). The works in Artëmov et al. (2010) deal with continuos and discrete time, introduce time constraints as linear inequalities, instantaneous actions and delayed actions with the delay chosen non-deterministically. The model is tailored in order to enable first order timed logic (FOTL) to automatically describe runs of ASM. This is achieved mainly for instantaneous actions in a form apt for formal analysis like model checking.

The previous approaches deal with time as an information needed primarily for system verification which imposes certain choices regarding the semantics of the ASM. The approach presented in this work incorporates concepts from previous approaches from the ASM community however it represents the time in the simplest useful manner, close to the basic ASM but on the other hand adapted to generate runs that can reduce the number of processed information. The model is intended to give a clear vision of semantics of timed algorithms and enables easy abstraction of external or internal components.

To the best of our knowledge, no other attempts were made to integrate a dynamically adaptable level of abstraction of the ASM.

1.2. Structure of the paper

In Section 2 we give a short description of the ASMs in general. We further introduce the temporal extension of ASM sin Section 3, its formalisation and the proof of some useful temporal properties. The idea behind the hierarchical ASM is presented in Section 4. Details and formalisation of the HTASM model are given in Section 5 together with the outline of the correction proof of the processor model and the abstract processor execution.

2. ABSTRACT STATE MACHINES

The ASM thesis was introduced by Yuri Gurevich as a reaction to the Turing thesis. One of the most general notion of states and dynamic state changes modern mathematics can offer is (static) algebras as states and guarded destructive assignments for abstract functions as basic dynamic operations. In the science of universal algebra, first-order structures with only functional vocabularies are called algebras. An algebra is a Tarski structure without the relations. It is known from the vast mathematical logic experience that any static mathematic reality can be faithfully represented by a first-order structure.

2.1. ASMs in a nutshell

Abstract State Machines are a computational model based on mathematical structures. The choice for the mathematical structure is static algebra which can be seen as a state. The model of the run are groups of finite updates that make transition from one state to another. The run proceeds either until a final state is reached or for an infinite number of steps. The evolution of states are achieved through the notion of dynamic algebras by the use of dynamic functions that change the content at certain locations. All the function names form a set called the *vocabulary*. A special function defined on this set is in charge with the interpretation of function names. For example, every vocabulary has the Boolean set $Bool = \{true, false\}$ as well as the natural boolean relation names as static functions. The interpretation of static functions is fixed throughout the run, and translates for the Boolean set as the natural boolean interpretation. Locations can be defined as functions applied to the argument. The interpretation of the functions is the content of the locations. Changing (or defining, if there is none) the value location (represented by the functions at the given parameters)

$$f(t_1, ..., t_n) := t$$

is done through *updates*. The ASM is a finite set of guarded updates: `if Condition then Update`.

ASMs can describe algorithms at the appropriate abstraction level. System design can be performed incrementally with the help of simultaneous updates which help avoiding an explicit description of the intermediate storage.

3. A TIMED ASM

The notion of time is important to model real-time systems. Hard real-time systems emphasis the notion of safety with regard to the respect of deadlines. Determining the WCET is essential for this kind of systems and therefore the notion of time in the hardware model used for the estimation. The most natural extension is to start from the basic ASM, expanding the notion of instantaneous updates that generate update sets in parallel with simultaneous effect, if they are consistent. The effect of the updates is to change the interpretation of the updated functions at the specified location. The consistency of the updates implies that parallel updates do not affect the interpretation of the same function at the same location and must be ensured by the user.

3.1. Adding time in basic ASMs

ASM is a state-based model, characterised by an explicit notion of state as static algebras. The state is a first class citizen while the events are secondary level citizens in the form of moves between states. The basic ASM has runs that consist in moves from on state to another. A move applies the update set generated by all the update rules with valid guards in that particular state. All updates are instantaneous and applied in parallel. This influences the way the system is designed, in our case the processor. In order to describe it's exact behaviour, we must give a *step by step* definition of all the interacting components.

Even though there is no explicit notion of ttime in the basic ASM, there is a clear notion of sequentiality that can be very easily applied to the model of a processor where all *changes* are governed by a central clock and applied in the same time after a clock tick. Therefore the basic ASM model can be seen as a transition system that gives a picture of the state before and after the clock tick. By adding a simple counter we can represent the notion of clock tick.

We introduce a simple way to abstract certain components that take several basic ASM moves to complete by adding the time information to the final group of updates. Therefore we introduce *delayed* updates that model a whole set of rules in a *blackbox* style. The execution of delayed rules implies the use of a global time scale that will be discussed in the following.

We use the notation δ to refer to a location holding a term defined on $\mathcal{T}$, $(l, v, \delta) \in U$, where v is the value of the delay. We can now introduce the $delay : U \rightarrow \mathcal{T}$ function that applies to the update set associated to an update rule in U and extracts the time information of that rule. If the rule has no duration information, then it returns 1.

$$delay(U, l) = \begin{cases} \delta & , \text{ if } \exists (l, v, \delta) \in U \\ & \quad \text{with } \delta = [\![t]\!]^{\mathfrak{A}}_{\zeta} \text{ and } v \in \mathcal{T} \\ 1 & , \text{ otherwise.} \end{cases}$$

Note that in this case v has the normal interpretation of terms in $\mathfrak{A}$ under ζ and can be any term that evaluates to a time value in $\mathcal{T}$.

Time also enables the notion of time-accurate transitions. The idea is to compact the run by only making moves that change the locations of the state besides the current time. In this case we obtain a conjunction of update sets that makes a move to an equivalent state (the first that is *different*) with the same equivalent duration $\delta = \sum \delta_i$.

If every rule can have an associated duration, then we can have in the same state update sets associated to different durations and the following scenarios.

3.1.1. No timed updates

The run is therefore equivalent to the one of a cycle-accurate model.

$$\mathfrak{B}_0, \mathfrak{B}_1 = \mathfrak{B}_0 + U_0, \ldots, \mathfrak{B}_{i+1} = \mathfrak{B}_i + U_i, \ldots$$

If no time update functions are present, the CT function is by default incremented by 1 after all the updates are effective.

3.1.2. Single timed update

Only one timed update occurs and no other guards of any rules are valid.

$\mathfrak{B}_0, \mathfrak{B}_1 = \mathfrak{B}_0 + U_0, \ldots, \mathfrak{B}_{i+1} = \mathfrak{B}_i + U_i$ with $U_{j \in [0,i]} = (l, v, \delta)$.

This is the equivalent of a time accurate run, as we make directly the move to a state that has *significant* updates.

3.1.3. Mixed updates

Both timed and untimed updates, or timed updates with different durations are specified.

$\mathfrak{B}_0, \mathfrak{B}_1 = \mathfrak{B}_0 + \{(l_0, v_0, \delta_0), \ldots, (l'_0, u'_0, \delta'_0)\}, \ldots$, with $\delta_0 \neq \delta'_0$.

In this case we first apply the update set associated with the smallest $\delta_{min} = min(\overline{\delta_i})$ and subtract δ_{min} from all the other durations. After each update, a move is made to a new state, meaning that all the rules are evaluated again and new update sets are added to the update set list. This is a way to encapsulate rules in a single black-box rule or in other words to replace a rule definition with a less refined version that hides the inner actions. When all the updates in a state take one cycle, we can say that we have a precise definition of all the units. Updates that take several cycles hide away some inner updates and look like less detailed definitions of the respective unit. Certain constraints apply to the use of such *delayed* versions, like the absence of *critical locations* with regards to update sets that must be applied earlier.

We first give the detailed mathematical definition of our model and then prove some temporal properties of the timed state transit system.

3.1.4. Detailed definition

The semantics of the delayed updates can be completely simulated with basic ASMs constructs and the introduction of the pre-interpreted sort $Time$ and the external pre-interpreted function CT that gives the time associated to the current state.

```
executedStatus := true
...
if C then
  if executedStatus = true then
    CD := CT + delay
    executedStatus := undef
  endif
  if CT = CD then
    delayedUpdateRule
  endif
endif
```

The semantics in the above listing imposes that the guard C remains valid throughout the whole delay period, which might be a necessary constraint. Eitherwise, we can use a definition similar to the *control state ASMs* to simulate the delayed application of the rule like in the listing below, where `new(CTD(ruleName))` is a function that generates a new location to store a delay for the rule `ruleName`.

```
if Cr then
  if ctl_state = 1 then
    CTD(ruleName) := CT + delay
    ctl_state := 0
  endif
endif
if CT = CTD(ruleName) then
  delayedUpdateRule
  ctl_state := 1
endif
```

Other than the verbosity of this approach it would be furthermore difficult to express the delay as an interval, where the delay can take all the values from the set $\{\delta_{min}, \ldots, \delta_{max}\}$.

3.1.5. Abstract States and Update Sets

For a processor modelling, a discrete interpretation of time suffices as all time informations of the description are multiples of a cycle. Therefore we can only react to external actions after the next cycle tick, which we consider sufficient in our WCET estimation context.

We extend the definition of the Signature from Borger and Stark (2003), Section 2.4, in order to handle time.

Definition 1 *(Signature): A signature Σ is a finite collection of function names. Each function name f has an arity, a non-negative integer. Every ASM signature contains the static constants undef, true, false and also the external dynamic pre-interpreted function CT that gives the value of the current time from the pre-interpreted sort $\mathcal{T}$ of time.*

The definitions of the state and location remain unchanged and are presented like in Borger and Stark (2003) from which we also adopt the notations.

Definition 2 *(State). A state A for the signature Σ is a non-empty set X, the superuniverse of $\mathfrak{A}$, together with interpretations of the function names of Σ.*

Definition 3 *(Location). A location of $\mathfrak{A}$ is a pair $(f, (a_1, \ldots, a_n))$, where f is an n-ary function name and $a_1, \ldots a_n$ are elements of $|\mathfrak{A}|$. The value $f^{\mathfrak{A}}(a_1, \ldots, a_n)$ is called the content of the location in $\mathfrak{A}$.*

We write $\mathfrak{A}(l)$ for the content of the location l in $\mathfrak{A}$ and also $\mathfrak{A}(\delta) = [\![\delta]\!]^{\mathfrak{A}}$ for the value of the delay in the state $\mathfrak{A}$. Depending on the interpretation of the duration δ that we allow we can introduce a first order temporal logic for our ASMs. This can be accomplished by allowing the delays to be terms and associate the same formulas like for the other locations.

The definition of updated set is modified in order to allow delayed updates.

Definition 4 *(Update and update set). An update for $\mathfrak{A}$ is a pair (l, v, δ), where l is a location of $\mathfrak{A}$, v is an element of $|\mathfrak{A}|$ and δ is the delay with regards to the current time, CT. The delay δ cannot be infinitely small and is multiple of a smallest time interval, that can be associated to a system clock tick, for*

example. The location l keeps its old value for δ moves starting from the moment the update was fired. The update is trivial, if v is the content of l in $\mathfrak{A}$. An update set groups all updates that must be fired with regard to the current state.

Delays associated to updates of locations imply that the update set of a state is not always empty after the move to the next state. However adding the notion of update set more closely in the state of the ASM would modify its structure. We prefer to consider that the update set is *consumed* in the current state and *passed to the next state*. In other words, the next state can already have a non-empty update set before starting the evaluation of its rules.

Due to the parallelism of the updates, we must avoid the update *clash*, when a function is updated at the same arguments in the same time.

We therefore modify the definition of the *consisted* update set.

Definition 5 *(Consistent update set). An update set U is called consistent, if it has no clashing updates, i.e. if for any location l, all elements v_1, v_2 and δ_1, δ_2 it is true that if $(l, v_1, \delta_1) \in U$ and $(l, v_2, \delta_2) \in U$, then either $v_1 = v_2$ either if $v_1 \neq v_2$ then $\delta_1 \neq \delta_2$.*

If an update set U is consistent in a given state, then it can be fired. In the generated new state the dynamic functions, that had associated updates with delay values equal to the time step of the move, are changed according to U.

Definition 6 *(Firing of updates). The result of firing a consistent update set U in a state $\mathfrak{A}$ is a new state $\mathfrak{A} + U$ with the same superuniverse as $\mathfrak{A}$ such that for every location l of $\mathfrak{A}$:*

$$(\mathfrak{A} + U)(l) = \begin{cases} v & , \text{ if } (l, v, \delta) \in U.\delta = 1; \\ \mathfrak{A}(l) & , \text{ if } (l, v, \delta) \in U.\delta > 1 \\ \mathfrak{A}(l) & , \text{ otherwise.} \end{cases}$$

The state $\mathfrak{A}+U$ is called the *sequel* of $\mathfrak{A}$ with respect to U, therefore $(\mathfrak{A}+U)(l)$ is the content of the location l after the firing of the updates in U.

Since U is consistent, the state $\mathfrak{A} + U$ is still determined in a unique way, and those locations have a new content in $\mathfrak{A} + U$ with respect to $\mathfrak{A}$ because if we do not have *immediate* the updates with $\delta > 1$ make a move directly after the minimum delay.

Due to the notion of update sets that can be *inherited* from one state to another, we must also change the definition of the difference between two states. Two cases arise for the definition of the difference: the cycle-transition where at least one update is *delayed* until the next *cycle* and the δ-transition for which we must modify the difference as the locations and all their associated delay decremented by δ_{min}. We must also ensure that the current time after a δ-transition will return the correct value, i.e. incremented by the right amount of steps.

Definition 7 *(Difference). Let $\mathfrak{A}$ and $\mathfrak{B}$ be two states with the same superuniverse. Then*

$$\mathfrak{A}-\mathfrak{B} = \begin{cases} \{(l, \mathfrak{B}(l), 1) \mid \mathfrak{B}(l) \neq \mathfrak{A}(l)\} \cup \\ \{(l, v, \mathfrak{A}(\delta) - 1)\} \cup \\ \{\mathfrak{A}(CT) + 1, 1)\} & , \text{ if } \exists \delta \in U.\delta = 1; \\ \{(l, v, \mathfrak{A}(\delta) - \delta_{min})\} \cup \\ \{(CT, \mathfrak{A}(CT) + \delta_{min}, 1)\} & , \text{ otherwise,} \end{cases}$$

where $\delta_{min} = min\{\delta_i \mid (l_i, v_i, \delta_i) \in U\} + 1$, and l_{CT} is the location of CT, the current time function.

The original ASM lemma still holds.

Lemma 1 $\mathfrak{A} + (\mathfrak{A} - \mathfrak{B}) = \mathfrak{B}$.

The passage of time is represented by the update of a special location therefore, we can look at the update set that moves a state to another state as the union of the update set of regular locations and the update of the time location.

Let $\mathfrak{A}$, $\mathfrak{B}$ and $\mathfrak{C}$ be states such that $\mathfrak{A} + U = \mathfrak{B}$ and $\mathfrak{B} + V = \mathfrak{C}$. We define $U = U_l + U_{l_{CT}}$.

The composition of update sets, which corresponds to the sequential application of the updates, is defined in the following way for basic ASMs,

$U \oplus V = V \cup \{(l, v) \mid$ there is no w with $(l, w) \in V\}$.

In other words, the composition of the update sets is the set of all the updates in V, some that override locations in U, others that are generated by U and all the updates unique to U and V.

The composition of delayed update sets U and V is the set $U \oplus V$ and follows a similar principle, containing:

- the updates that were generated by firing the rules in U, which guaranties that the application of the composition is equivalent to the sequential application;
- the updates that should have fired in U and are not overridden in V at the next state, therefore after a $[\![\delta_{min}]\!]^{\mathfrak{A}+V}$ delay;
- all the updates that are unique to V, with the delay incremented by $[\![\delta_{min}]\!]^{\mathfrak{A}+U}$.

All updates were incremented with $[\![\delta_{min}]\!]^{\mathfrak{A}+U}$ in order to maintain the consistency of the count time

function ct, that has to be equal after the firing of the two rules to the sum of their minimum delays.

Definition 8 *(Composition of update sets).*

$U \oplus V =$
$\{(l, v, [\![\delta_{min}]\!]^{\mathfrak{A}+U} + [\![\delta_{min}]\!]^{\mathfrak{A}+V}) \mid (l, v, [\![\delta_{min}]\!]^{\mathfrak{A}+U}) \in U$ *and there is no* w *with* $(l, w, [\![\delta_{min}]\!]^{\mathfrak{A}+V}) \in V\} \cup$
$\{(l, v, \delta + [\![\delta_{min}]\!]^{\mathfrak{A}+U}) \mid (l, v, \delta) \in V\}$

The composite of update sets still works as long as we can compute the value of the current time in the intermediate state (in our case $\mathfrak{B}$).

Lemma 2 *Let U, V and W be update sets.*

1. $(U \oplus V) \oplus W = U \oplus (V \oplus W)$
   The equality is obvious because of the associativity of the union operation on sets.
2. If U and V are consistent, then $U \oplus V$ is consistent.
   The consistency of the composition is ensured by definition and it verifies the Definition 5.
3. If U and V are consistent, then $\mathfrak{A} + (U \oplus V) = (\mathfrak{A} + U) + V$.

Definition 9 *(Move of an ASM). We say that an ASM M makes a move from a state $\mathfrak{A}$ to another state $\mathfrak{B}$ (written $\mathfrak{A} \overset{M}{\Longrightarrow} \mathfrak{B}$) when the main rule yields a consistent update set U in state $\mathfrak{A}$ and $\mathfrak{B} = \mathfrak{A} + U$.*

3.2. Equivalence with the basic ASM

Adding the delay δ to the updates, (l, v, δ) can be explained through classical locations. If we introduce a mapping function $\zeta_\delta : \Sigma_\delta \to \Sigma \setminus \Sigma_\delta$ where $\delta : \mathcal{T}$ are location names from a subset of the ASM vocabulary Σ we can associate to any location from $\Sigma \setminus \Sigma_\delta$ a duration δ. In order to represent the advancement of the current time we can use a rule that will be fired at each step that computes the minimal delay δ_{min} and adds it to a controlled function ct.

3.3. Timed ASM defined by a set of Axioms

In this section we prove the respect of the time properties presented in Graf and Prinz (2007).

Each consistent update set has at least a location that stores the current time. Let U_i be un update set such that $\mathfrak{A}+U_i = \mathfrak{B}$ and $\mathcal{U}$ be the set of such update sets associated with the run of the ASM. We have

$\forall U_i \subseteq \mathcal{U}, \exists l, (l, v, \delta) \in U_i.\delta = \delta_{min}.$

Therefore we can define a function that gives the time of a move for every move of the ASM. A move takes place at the moment when at least one guard of a rule is satisfied. Therefore the value of the function when is the current time after the move to the new state, which is the current time plus the minimum delay of the updates from the start state.

Property 1 *The move from a state to another takes place at a point in time defined by the function $when$:*

$when : M \to Time$

Let $\mathfrak{A} \overset{m_1}{\Longrightarrow} \mathfrak{B}$ be a move from state $\mathfrak{A}$ to $\mathfrak{B}$, we have

$when(m_1) = \mathfrak{B}(CT).$

Property 2 *Global Time: all time stamps of a run are totally ordered, i.e. the partial order of the moves is extended to a total preorder for their occurrence times, i.e.*

$\forall m_1, m_2 \in M.when(m_1) \leq when(m_2) \vee when(m_2) \leq when(m_1)$

For the moves between state, the property 2 is too strong if we look at the moves as the firing of all updates in the update set at the next delay. This only works for rules that are indeed in either executed in parallel or one after the other. In the general sense $\forall m_1, m_2 \in M.when(m_1) \leq when(m_2) \vee when(m_2) < when(m_1)$.

Property 3 *Strict time progress along causal chains: whenever two moves are causally ordered, their occurrence times are strictly ordered, i.e.*

$\forall m_1, m_2 \in M.m_1 < m_2 \Longrightarrow when(m_1) < when(m_2).$

According to Definition 7, a move takes at least one unit of time, therefore the property above is trivial.

Property 4 *Timed order is not stronger than causal order: causally non related events are not comparable in the timed order, i.e.*

$\forall m_1, m_2 \in M.m_1 \leq m_2 \iff when(m_1) \leq when(m_2).$

For the time being, we forbid conditions on the delays, therefore the time order is induced only by the casual order.

Property 5 *Absence of Zeno computations: In any infinite run, there is no upper bound of the time values attached with moves, i.e.* $\forall R\ isInfinite(R) \Longrightarrow \forall t \in Time\ \exists m \in R.when(m) > t.$

The Property 5 is satisfied because if the number of moves is infinite we will always have a new move with the current time, $when(m)$ superior at least with one time unit then the previous one.

Property 6 *Minimal time distance: There is a lower bound of the time differences between non-simultaneous causally ordered moves, i.e.* $\exists \delta \in Duration\, \forall R\, \forall m_1, m_2 \in R. m_1 < m_2 \implies when(m_2) - when(m_1) > \delta$.

The minimal duration allowed in our update semiotics is one, therefore such a δ exists.

Property 7 *Events at discrete steps: Any two moves occur either at the same instant or the time differences between their occurrence times are a multiple of a given value,*

$\exists \delta \in Duration\, \forall R\, \forall m_1, m_2 \in R \exists k \in N. when(m_1) - when(m_2) = k * \delta$.

According to our definition, the value of the minimal delay is 1. The value of the other delays δ_{min} is a multiple of the minimal delay hence $k = 1$.

Property 8 *Local urgency: The time of each state change of each run is minimal.*

Property 9 *Global Urgency: The earliest state change amongst all distributed agents is taken.*

Updates sets are composed of updates and different associated delays. According to Definition 7, the time progress is defined as the minimal delay in the updates list.

Time
The time is an integrated part in the ASM. We associate time to an update set in the following way. If no time information is present, the time associated to the update set is equal to one as in one cycle. We can hide away details of the implementation of a rule by associating to an update set a time superior to one. We use the term *timed* rule or update to distinguish the last one from the regular one-cycle case. The interpretation is that we see the more complex rule that would have executed in several cycles as a *turbo* rule where the only thing that matters is the result, or the final update set. Several conditions must be satisfied in order to be able to use a timed rule. From the refinement point of view, the one-cycle update corresponds to the most refined version of the ASM which is sufficient inout case to model the processor behavior. We will now take a look at the timing properties of our state transition system.

Duration as interval
An interesting extension is the possibility to add duration intervals for certain actions. There is a difference between our notion of duration interval and the one presented in Artëmov et al. (2010) where the delay is chosen non-deterministically from the interval. We will use the notion of interval in order to simulate traces of abstract runs. When such a duration interval is associated to an update, the pipeline of the processor will generate different time results for different availability times from the interval. A problem for the WCET estimation occurs when the resulting durations are not monotonic with respect to the availability values from the time interval. This is called a timing anomaly, because it generates unexpected results like a cache miss being optimistic than a cache hit with regards to the execution time.

Being able to use imprecise intervals for a certain action comes in hand when using the model in pair with a value analyser that gives imprecise information. In this case a parallel execution of all the different scenario must be made.

Our interest, with regard to the WCET estimation, is to execute in parallel only until a *merging* point is reached, where it would be safe to return to a single execution flow that will generate the highest global execution time.

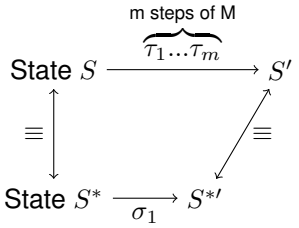
Therefore we must introduce the notion of *stores* and *parallel stores*. We can thus handle the multiple generated runs, each time such an interval is encountered. The problem with this type of execution is the inherent state-space explosion, hence we introduce the notion of *reference store* and at every execution point that deals with an duration interval we will keep a history of all the update sets that must be applied in order to get into that respective state of that particular run. This is equivalent to keeping a list of all the locations that have been modified since the splitting point if we merge all the updates. The merging consists of keeping only the final value for the same location. To even further compact the update sets, we can compare the locations to the ones from the original store and eliminate the information if it's redundant.

After reducing the update sets several scenarios arise.

- an update set becomes empty. This means that except for the count time, we arrive in the same state as the one where the execution split. In this case we can fusion it with the original run, keeping the greater count time.
- an update set becomes identical to the one of another run, we can thus fusion and eliminate it, preserving the maximal count time.
- un update set becomes similar to one of another run. The notion of similarity is to be further discussed. The intuition is that we can reach two states that have differences only in

non-interfering locations, so the two can be merged.

States of ASMs are static algebras, therefore we can use this fact to create equivalence classes between different states of the parallel runs in order to reduce the state space explosion when searching for similar states.



4. A HIERARCHICAL ASM (DYNAMIC CHOICE OF ASM REFINEMENTS)

The hierarchical notion of ASMs in already present in the ASM literature as a basis for the incremental design by refinements. Besides the compactness and good readability of specifications, ASMs offer a homogenous formalism for all levels of abstraction. We believe that the notion is not fully exploited as the level of abstraction of ASM is fixed at the beginning of the modelling phase. We introduce the concept of dynamic choice of the refinement granularity in what we cal Hierarchical ASMs. Therefore, the model itself, is able to chose on the fly the appropriate abstraction level for a given rule among several, user-defined, definitions. This can be particularly useful in the case of a processor design as the precision on the values of manipulated data, given by a value analyser for example, is not always on pair with the level of detail of the processor model. The main idea is to adapt the level of abstraction of the processor description to the precision given by the value analysis in order to master or reduce the state-space explosion inherent to the WCET analysis.

Correctness of the hierarchical ASM
A Hierarchical ASM has multiple definitions for different sub-ASMs (we will use this term for the time being with no regards to the Structured ASMs) that correspond to different levels of refinement that we call abstractions. As during the same run, either of the available hierarchy levels can be selected, we must ensure that the run in correct with regard to the semantics of the processor and also with regards to the time estimation. Informally, the correctness of the model is granted by ensuring all the hierarchical level of the sub-ASM have the same input and output (signatures), that their initial and final states are equivalent and the intermediate runs produce also equivalent states, in the same order.

4.1. Cycle-accurate vs time-accurate model

The time-accurate ASM can be seen as a simple case of ”hierarchy” as the refinement is correct by definition because in the $(m, 1)$ refinement all the intermediate states are identical (the reason to use the time-accurate model in the first place). The top edge of the commutative diagram represents the cycle-accurate model and the bottom part the time-accurate model. The equivalence notion $\equiv$ between locations of interest in corresponding states is trivial to prove and follows from the definition of the time-accurate model. Except for the time function, the pair of start and end states are identical so we have $S = S^*$ and $S' = S^{*\prime}$.

5. HIERARCHICAL TIMED ABSTRACT STATE MACHINES

5.1. Preambule

ASMs have the nice property of being able to describe any algorithm at its right abstraction level. One major difference in the concept of ASMs with regards to other state transition systems is that the values of the support set remain the same while the *transfer* functions change after an update. The relation between functions and data is reversed, instead of having mutable data structures with immutable functions, we have immutable data which is operated on by mutable and immutable functions. A data selector cane be seen as a single argument function over the selected data domain and the other way around. The assignment of an expression to a variable field f of a record x can be re-interpreted as an update of the mutable function f at the position x. In other words instead of changing values, the interpretation of functions, for the right arguments gets changed, associating the function name at the respective arguments to a new value from the superuniverse. We have thus only immutable data serving as index structures for possibly mutable functions. The abstract state machines approach leads naturally to components which are adaptable.

As an extension to this logic, we introduce HTASMs not only as transition systems that change the interpretation of functions according to the rules but also the interpretation of the interpretation of functions. This is achieved through the extension of the model, with an *oracle* that will modify the interpretation of function names, making them depend on different set of rules.

The decisions of the Oracle can be modelled either based on the internal state of the HTASM (watching certain locations can trigger a decision to use a

more abstract state when suited - for example when no timing anomalies can occur) or on external, monitored locations (for example when we have insufficient precision on certain values we can switch to a definition of an unit that can work with the larger - more imprecise - domain).

5.2. Mathematical foundation of HTASM

According to the abstract-state postulate an algorithm does not distinguish among isomorphic types. A state is just a certain implementation of its isomorphism type. Looking at the way elements of the base set are accessed, allows us to identify a good manner to introduce multiple *views* of the same *operation*. Base set elements are accessed through ground terms that contain functions from the vocabulary of the state. By allowing multiple definitions for a function we access different ground terms leading to an equivalent operation with regarding to the semantics of the processor and its temporality for example.

The classic ASM refinement techniques provides us with the ability to build an abstract model with an equivalence notion between data in locations of interest in corresponding states. We want to be able to use multiple refinements levels during the same execution of the algorithm that ensure a correct result with regard to the target property.

Contrary to the refinement concept we want to make moves in both senses, from the refined version to the more abstract and vice-versa. This would not be possible in the general case, nevertheless thanks to the target property (the temporal over-approximation) we can make jumps in both ways.

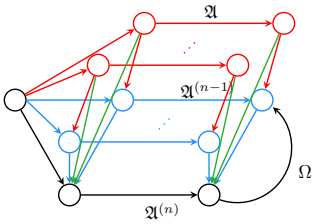


Figure 1: *Dynamic HTASM abstraction level switch*

We hereby extend Definition 2 of the *state* to take into account the different levels of abstraction.

Definition 10 *(HTASM State). A state A for the signature Σ is a non-empty domain $X = X_{Set} \cup X_{\alpha}$, the superuniverse of $\mathfrak{A}$, together with interpretations of the function names of Σ in one or more domains.*

For each rule name we can give a definition for each universe of X. Therefore the values of terms, functions and their arguments, can be *imprecise*.

Definition 11 *(Critical rule). A rule is called critical if it changes the interpretation of terms used in the guard formula of another rule, hence a rule dependency exists.*

In the case of a function term, we speak of critical locations.

Definition 12 *(Critical location). A location is called critical if it is involved in a critical rule.*

Let r_i be a non trivial update rule (l_i, v) at location $l_i = (f, (a_1, ..., a_n))$ and let R be a guarded update if φ then r where φ is a function formula depending on a location location l. If $l = l_i$ then l is a critical location.

$$cloc(r, \mathfrak{A}) = \{l_i \in U : \exists l_j \in \varphi . l_i = l_j\}$$

In the case of a variable term, we speak of critical variables.

Definition 13 *(Critical variable). A variable is called critical if it's involved in a critical rule.*

Let r_i be a non trivial update rule (l_i, v) at location $l_i = (f, (a_1, ..., a_n))$ and let R be a guarded update if φ then r where φ is a function formula depending on a location l. If $l = l_i$ then l is a critical location.

To describe the behaviour of a precision guided HTASM abstraction level choice, we now introduce the ***abs***-construct which combines simultaneous atomic updates of basic TASMs in a global state with a choice of rules to apply.

We denote the abstraction level choice of two HTASM rules P, Q by P ***abs*** Q and define its semantics as the effect either executing P in the given state $\mathfrak{A}$ or Q in the same state $\mathfrak{A}$, depending on which domain critical locations belong too, where U is the set $[\![P]\!]^{\mathfrak{A}}$ of updates produced by P in $\mathfrak{A}$.

Definition 14 *Let P and Q be HTASM rules.*

$$[\![P \textbf{ abs } Q]\!]^{\mathfrak{A}} = [\![P]\!]^{\mathfrak{A}} \oslash [\![Q]\!]^{\mathfrak{A}}$$

$$U \oslash V = \begin{cases} \{(l,t) \mid (l,t) \in U\} & , \text{ if } g \in guard(P, \mathfrak{A}) \\ \implies Dom(t) \in Dom(g)\}; & \\ V & , \text{ otherwise.} \end{cases}$$

where $guard(P, \mathfrak{A})$ denotes the function that returns the last values of the locations used in the guards of P and $Dom(t)$ the functions that returns the type of domain of the super universe of t.

5.3. Correctness proof outline

Safety-critical systems require a certain level of confidence regarding the respect of some

constraints. In order to ensure a level of confidence to the system, tools are used to verify the respect of functional and non-functional properties. One of the advantages of using a formal model for the processor description is the foundation to build proofs of correctness. We provide in the following the intuition on how this can be achieved.

We distinguish two cases that both benefit from the formal support of the model. Firstly, we need to prove that the processor model is correct with regards to the real processor that will used in the actual system and secondly that all the abstraction levels of the processor are correct refinements of HTASMs. ASMs provide a stepwise refinement method that allows the designer of the system to start with a high-level description of the system that is refined step by step into the final version which can be proven correct with regard to the initial one. If a processor SystemC description is available, we can automatically generate a correct HTASM model using techniques described in Muller et al. (2004) with minor modifications because all the additions to our language preserve the nice properties of ASMs.

In order to prove that the different abstraction levels of the HTASM are correct we can use similar techniques used to prove the correctness of compilers, Goos and Zimmermann (2000). Program transformation used in compilers consists in transforming the control and data-flow graphs. In other words, the observable behaviour of the program must be preserved. In our case, we use as an input for the processor model the compiled code of the program. The transformation consists in two steps: data mapping and operation mapping. When compiling a source code, the initial graph depends on values that are known only at compilation time, and the stack and heap are being mapped into the internal register system of the target processor. Similarly we have a high level vision of the architecture in the form of the binary (after the value analysis step which provides information about loop counter, variable interval values, addresses, etc.) based on instructions from the ISA of the processor and a low level vision based on microcode operations that describe the exact behaviour at byte level of a particular ISA implementation (the actual processor).

In Figure 2, the top graph represents the most abstract model (the binary code) and the bottom one the most concrete model (the processor). Proving the correctness of the two models comes to independently prove the correctness of the data mapping and conditional graph rewrite rules. The data mapping assigns a new semantics (by means of an HTASM $\mathfrak{A}'_{\mu P}$) to the binary code using the

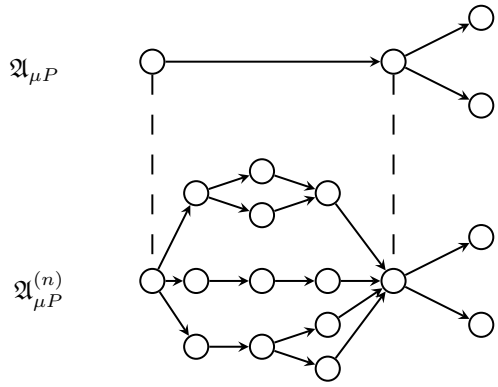


Figure 2: *HTASM refinement*

concepts of the data part of the target language. The behavioural part is kept, therefore the corectenns of the mapping can be shown by proving that $\mathfrak{A}'_{\mu P}$ 1-1-refines $\mathfrak{A}_{\mu P}$.

ASMs are transition systems which transfer static algebras. The abstract state machines make use of the following abstraction principle, while limited to the notion of evolving algebras:

5.4. Abstract processor execution

Analysing all reachable states of a processor makes the WCET estimation safe. Nevertheless, because of the state space explosion we must eliminate as much individual state handling as possible.

The HTASM model is custom tailored to confine the state space explosion of the undergoing analysis. After a number of safe, abstract steps the analysis goes back to a concrete state that corresponds to the global worst case. If the information regarding that state is lost or it is decided to be computationally expensive, the state is safely over-approximated by choosing a more pessimistic one.

We provide in the following a schematic view of the use of the model in the WCET estimation.

1. Complete the value analysis of the binary. We obtain information on the CFG, instructions, loop counters, register values, addresses, etc.
2. Start the conjoint symbolic execution (SE) on the HTASM model of the processor.
3. (a) It the value is exact $\rightarrow$ use the concrete HTASM $\mathfrak{A}^{(0)}_{\mu P}$.
   (b) If the value is imprecise (set, interval, etc.) $\rightarrow$ use abstract HTASM $\mathfrak{A}^{(i)}_{\mu P}$. We deal with symbolic values $\rightarrow$ all or some if the parameters of the functions (locations of the units functions) are intervals.
   What is the type of dependency?

- unit type
- functional (an instruction needs or depends on a certain value) → split the domain set (the universe of the HTASM) in different sub-domains that satisfy the constraint.

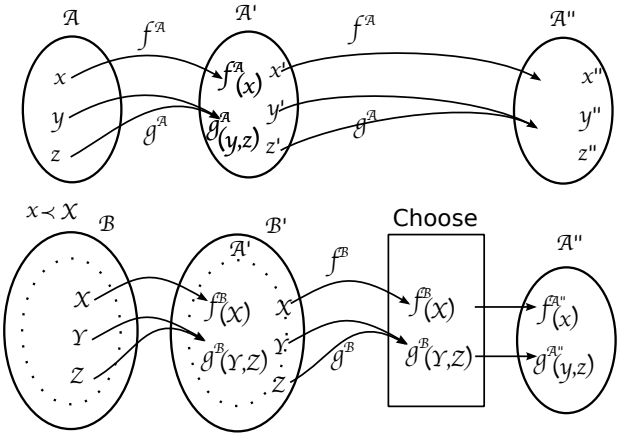


Figure 3: *HTASM abstract execution*

5.5. Handling Timing Anomalies

The evolution of processor architecture made the timing analysis more complicated. Among the consequences of modern processor features, timing anomalies (TA) have an important impact on the WCET estimation, by braking the compositionally of the analysis. Timing anomalies in the context of WCET analysis were first described by Lundqvist and Stenström (1999). Hardware acceleration mechanism produce interferences that lead to timing anomalies, i.e., a local timing change causes an either larger or inverse change of the global timing.

The abstract execution problem comes to:

- being able to handle many potential states;
- at any moment either we have precise information about values → next step or we don't and we must identify the worst case. If we don't know if the worst case can occur, we must suppose that it will.

Usually, the WCET analysis is confronted, at each execution step, to either a great overestimation of the WCET or a complex analysis of the precise worst case path.

Our idea is to use the timing anomalies identification techniques, presented in Wenzel et al. (2005) and Eisinger et al. (2006), to partition the analysis space in several categories corresponding to the presence or absence of the TA.

Wenzel et al. (2005) proposes a criterion that provides a necessary but not sufficient condition for timing anomalies to occur. Our model is based on relation between locations that are stored in functions and guarded updates, capturing by definition the timing anomalies. These relations are exploited to identify the necessary condition for the TA to occur. We can safely assume that if the necessary conditions are not satisfied, TA will not occur and we can use the *divide and conquer* analysis approach.

Therefore we create functions that evaluate the timing anomalies that can occur, obtaining the following cases:

- no TA are possible → chose the worst case;
- some TA might occur → analyse all the cases or suss abstraction techniques.

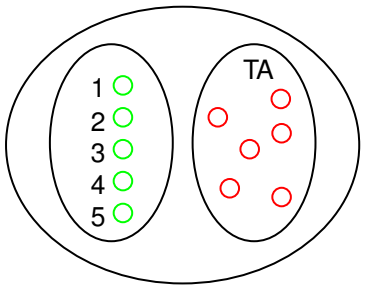


Figure 4: *Timing anomalies partitioning*

In the category where no TA can occur, we have an order on the states defined by a distance on abstract states based on the temporal impact and further relations on locations. This can also be used for defining similarities between states and perform merging based on techniques presented in Benhamamouch and Monsuez (2009).

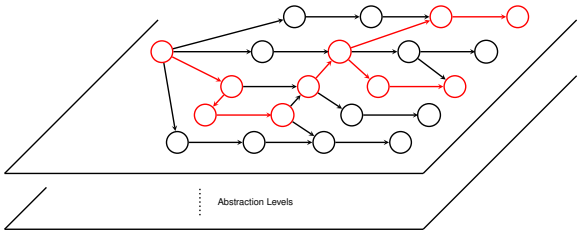


Figure 5: *Timing anomalies identified paths through relations between locations*

6. CONCLUSION

We have proposed an extension of the ASM model that handles time and dynamic abstraction in a simple manner. The possibility to make delayed transition is presented as a support for abstracting the processor components in order to achieve a more compact simulation. Some temporal properties of the system were enumerated and discussed. We

have also introduced a model that can dynamically refine the components of the processor, preparing a framework where run-time abstraction can be made in both ways between the concrete and the more abstract definition. The adaptability of the analysis, given by the separation of the processor model and the analysis, the ease of use, the preservation of the formal background after the model extensions, the adaptability to imprecise value analysis, the state space explosion confinement techniques through abstractions and fusions and the built-in capturing of timing anomalies makes our model adaptable for the WCET estimation. An WCET tool based on this model is under development.

REFERENCES

Anlauff, M. (2000). XASM- An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (Eds.), Abstract State Machines - Theory and Applications, Volume 1912 of Lecture Notes in Computer Science, pp. 69–90. Springer Berlin Heidelberg.

Artëmov, S. N., Y. Matiyasevich, G. Mints, and A. Slissenko (2010). Simulation of Timed Abstract State Machines with Predicate Logic Model-Checking. Ann. Pure Appl. Logic 162(3), 173–174.

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

Benhamamouch, B. and B. Monsuez (2009). Computing worst case execution time (WCET) by Symbolically Executing a time-accurate Hardware Model. In International MultiConference of Engineers and Computer Scientists, Volume II, pp. 3–8.

Borger, E. and R. F. Stark (2003, June). Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag New York, Inc.

Cohen, J. and A. Slissenko (2008). Implementation of Sturdy Real-Time Abstract State Machines by Machines with Delays. In Proc. of the 6th Intern. Conf. on Computer Science and Information Technology (CSIT'2007), September 24–28, 2007, Yerevan, Armenia. Organized by National Academy of Science of Armenia in cooperation with Test Technology Technical Council of IEEE Computer Society. National Academy of Science of Armenia.

Eisinger, J., I. Polian, B. Becker, A. Metzner, S. Thesing, and R. Wilhelm (2006). Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In M. S. Reorda, O. Novk, B. Straube, H. Kubatova, Z. Kotsek, P. Kubalk, R. Ubar, and J. Bucek (Eds.), DDECS, pp. 15–20. IEEE Computer Society.

Gaul, T. (1995). An Abstract State Machine specification of the DEC-Alpha Processor Family. Technical report, University of Karlsruhe.

Goos, G. and W. Zimmermann (2000). Verifying compilers and asms or asms for uniform description of multistep transformations.

Graf, S. and A. Prinz (2007, May). Time in State Machines. Fundamenta Informaticae 77(1-2), 143–174.

Gurevich, Y. and J. Huggins (1996). The railroad crossing problem: An experiment with instantaneous actions and immediate reactions. In H. Kleine Bning (Ed.), Computer Science Logic, Volume 1092 of Lecture Notes in Computer Science, pp. 266–290. Springer Berlin Heidelberg.

Huggins, J. K. and D. V. Campenhout (1997). Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. ACM Transactions on Design Automation of Electronic Systems 3, 563–580.

Lundqvist, T. and P. Stenström (1999). Timing anomalies in dynamically scheduled microprocessors. In Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS '99, Washington, DC, USA, pp. 12–. IEEE Computer Society.

Muller, W., J. Ruf, and W. Rosenstiel (2004). An asm based systemc simulation semantics. In W. Muller, W. Rosenstiel, and J. Ruf (Eds.), SystemC, pp. 97–126. Springer US.

Ouimet, M. and K. Lundqvist (2008, June). The Timed Abstract State Machine Language: Abstract State Machines for Real-Time System Engineering. Journal of Universal Computer Science 14(12), 2007–2033.

Wenzel, I., R. Kirner, P. Puschner, and B. Rieder (2005). Principles of timing anomalies in superscalar processors. In Proceedings of the Fifth International Conference on Quality Software, QSIC '05, Washington, DC, USA, pp. 295–306. IEEE Computer Society.