On Implementations of Instantaneous Actions Real-Time ASM by ASM with Delays

J. Cohen[1] **A. Slissenko**[1♯]
E-mail: {j.cohen,slissenko}@univ-paris12.fr
Laboratory for Algorithmics, Complexity and Logic, University Paris 12, France

Abstract. We define a notion of implementation for real-time Abstract State Machines (ASM) with instantaneous actions by machines with delayed actions. The time is continuous, and time constraints are metric. It is clear that not every machine of the first type can be implemented by a machine of the second type. We describe sufficient conditions on ASM that permit to construct such an implementation in a rather straightforward way. The problem of properties preservation, when implementing ASM within this framework, is also briefly discussed.

1 Introduction

We study the problem of implementation of real-time Abstract State Machines (ASM) with instantaneous actions, as can be found in [GH96,BS02]. The time we consider is continuous and is modeled by non negative real numbers. The time constraints are metric and may involve arithmetical operations. We consider only very basic ASM, that are however largely used in theoretical and applied papers on ASM. ASM with instantaneous actions are a convenient abstraction that permit not to think about many secondary details of implementation. However, one must be careful when specifying a program in terms of such machines to be sure that this abstract specification is implementable correctly. We study the implementation of such ASM with instantaneous actions by ASM with delayed actions. More precisely, we consider one type of implementation that is easy to construct and that seems to be the first that comes to mind. The machines used for the implementation are in some sense traditional while-programs with parallelism. It is clear that not every ASM of the first type is implementable by an machine of the second type. For example, if the input values can change within intervals of time that are smaller than delays, it is impossible to guarantee that all the inputs will be processed. We have found that there are many other subtleties that may make an implementation impossible. Roughly speaking, if we wish to implement in a more or less straightforward way an ASM with instantaneous actions by a machine with delayed action, the former must be very robust.

[1] *Address:* Dept. of Informatics, University Paris 12, 61 Av. du Gén. de Gaulle, 94010, Créteil, France.
[♯] Member of St.Petersburg Institute of Informatics and Automation, Russian Academy of Sciences.

2 Real-Time Abstract State Machines with instantaneous and delayed actions

To illustrate the types of ASM we deal with and the whole problem of implementation consider the following ASM:

> **ForAll** $x \in Tracks$
> **If** $Cmg(x) \wedge DL(x) = \infty$ **Then** $DL(x) := CT + WT$
> **If** $\neg\, Cmg(x) \wedge DL(x) < \infty$ **Then** $DL(x) := \infty$
> **If** $DirOp \wedge \neg\, SafeToOpen$ **Then** $DirCl := true$
> **If** $\neg\, DirOp \wedge SafeToOpen$ **Then** $DirOp := true$

This ASM, that we comment just below, is a controller for the notorious Generalized Railroad Crossing Problem taken from [BS02], and that is very close to that of [GH96]. It controls a finite number of tracks that constitute the set $Tracks$. The controller analyzes simultaneously the input signals of sensors $Cmg(x)$, each one saying whether there is a train in the zone of control. $DL(x)$ is an internal function of the controller with values in $\mathcal{T} \cup \{\infty\}$, where $\mathcal{T}$ is time treated as non negative reals. The value ∞ adds to $DL(x)$ a functionality of a flag that catches the first instant of arrival or leaving of a train. So when a train arrives on a track x the controller memorizes the time of arrival, given by the current time CT, plus time WT during which there is no need to send the signal to close the gate if to take into account the situation only on this track x. The difference $WT = d_{min} - d_{close}$, where d_{min} is the minimal time of reaching the crossing by a fastest train (counting from the instant of its detection), and d_{close} is an upper bound on the time of closing of the gate counting from the instant when the signal $DirOp$ becomes $true$.

So the first two lines of the ASM above describes a way to detect the instant of arrival of a train on a track; in fact the lines are parameterized by x, thus their 'executable' number is twice the number of tracks. The next two lines describe the closing and opening of the gate. The condition $SafeToOpen$ says that for all track x either x is empty (no train) or the train is yet far from the crossing, i.e., $CT < DL(x)$.

This ASM consists of lines of **If-Then**-operators that are executed simultaneously and instantaneously, as well as the assignments after **Then**.

Generally an ASM is defined by a vocabulary that consists of sorts, and functions (and predicates). Equality relation for each sort is always in the vocabulary. Sorts and functions may be pre-interpreted or abstract. As pre-interpreted sorts we presume real numbers, time $\mathcal{T}$ (non negative reals), Boolean values. As pre-interpreted functions each vocabulary we consider contains the addition of reals, unary multiplications of reals by rational numbers, order relations over reals, rational constants, Boolean values $true$ and $false$. All the mentioned functions are static, i.e., they do not change during executions. One particular pre-interpreted function is dynamic — it is CT, Current Time, that is interpreted as identity over time. Abstract functions in our context are dynamic. Some of them are inputs —

they cannot be changed by machine. Other our internal, i.e., they are computed by machine. among internal functions we distinguish output (functions that are used in requirements specifications), and proper functions of machine. In our example, Cmg is an input, $DirOp$ is an output and DL is a proper function of the machine. One can formulate the safety property using only inputs and outputs; this property say that if a train is in the crossing then the gate is closed (this formulation is not exactly in our vocabulary, but it can be described in an appropriate way).

The ASM above is parametric, its parameters are $Tracks$ and real constants d_{min} and d_{close}. An implementation cannot have parameters, so we consider ASM without parameters where all functions are nullary ("variables" of programming). Without loss of generality we suppose that all functions are either predicates or real-valued (functions with a fixed number of values can be replaced by predicates).

The general form of such basic ASM is the following :

If G_1 **Then** A_1
$\vdots$
If G_k **Then** A_k
ASM $\mathcal{A}$.

where G_i are guards that we assume to be conjunctions of literals, and A_i are assignments (*updates*) that we assume to be of the form $f_i := \theta_i$ with f_i being an internal function and θ_i being a function or arithmetical term (in fact, a sum of real-valued functions with rational coefficients).

The both constraints on the form of the machine do not diminish the generality — each basic machine can be put into an equivalent machine of such form by a simple syntactic transformation.

We do not go into details of semantics. A *run* for a given input is defined uniquely in the following way. In an infinite loop, we check simultaneously all the guards, and for those that are true we execute, again simultaneously, the corresponding updates. These ASM will be referred to as IA-machines (*IA* stands for *Instantaneous Actions*).

We consider a particular implementation $\mathfrak{I}$, clearly practically well motivated, that transforms an IA-machine $\mathcal{A}$ into machine $\mathfrak{I}(\mathcal{A})$ that uses more rich language. We do not give a formal description of this language. The notations will be self-explanatory.

Description of $\mathfrak{I}$. Let $\mathcal{A}$ be an IA-machine (see above), and let V be its vocabulary of abstract dynamic functions. Denote by $\widetilde{V}$ a vocabulary of new functions such that each $z \in V$ has a respective function $\widetilde{z} \in \widetilde{V}$. Denote by $\widetilde{CT}$ a new function that will be used to save CT.

The machine $\mathfrak{I}(\mathcal{A})$ is shown in Fig. 1 and illustrated in Fig. 2. This machine has non deterministic delays between actions. More precisely each assignment and each evaluation of a guard takes some positive bounded time. Machine $\mathfrak{I}(\mathcal{A})$ works in two phases: Backup and Update, executed sequentially.

During the first phase the machine $\mathfrak{I}(\mathcal{A})$ saves the state of $\mathcal{A}$ in the new variables marked by tilde. This phase is executed in parallel, however because of delays, the backup may be a mixture of consecutive and simultaneous actions. The

Repeat
  Seq
    ForAll $z \in V$
      Par
        $[z := \widetilde{z},\ \widetilde{CT} := CT]$
      EndPar;
    ForAll $l = 1 \cdots k$
      Par
        If $\widetilde{G_l}$ **Then** $f_l := \widetilde{\theta_l}$ **EndIf**
    EndPar
  EndSeq
EndRepeat

Fig. 1. implementation $\mathcal{A}_1$

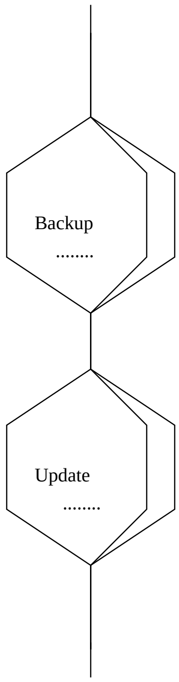

Fig. 2.

second phase simulates, in a similar parallel manner, the work of $\mathcal{A}$ using the saved values. The guard $\widetilde{G_l}$ and the term $\widetilde{\theta_l}$ are obtained from respectively G_l and θ_l by replacing $z \in V$ by the respective function $\widetilde{z} \in \widetilde{V}$.

We can describe these machine with delays in a general manner,(e.g., see []), but do not need it here. However, we will refer to such machines as DA-machines (*DA* stands for *Delayed Actions*).

The only important parameter of the implementation $\mathfrak{I}(\mathcal{A})$ is the total time of one loop. Below it will be denoted by ξ.

Clearly, if an input may change its value within a duration not greater that ξ the machine $\mathfrak{I}(\mathcal{A})$ can miss some of input changes, and consequently, can not to do the corresponding update. Or it may make an update that does not correspond to any update in $\mathcal{A}$. In fact, the situation is more subtle, and $\mathcal{A}$ should be very robust to admit an implementation that we have just described.

3 Approximate Implementations

Now we make precise the type of implementations we study. In this section $\mathcal{A}_0$ is a IA-machine and $\mathcal{A}_1$ is a DA-machine (not necessarily the implementation mentioned above). We define what is an approximate implementation of $\mathcal{A}_0$ by $\mathcal{A}_1$. Because of delays we cannot have exact correspondence between update instants of two machines, neither between the value of real-valued functions. So we may demand only approximate correspondence between runs of two machines, the approximation for the instants of updates will de denoted by ε and the approximation values — by η.

Denote by V the vocabulary of abstract dynamic functions of $\mathcal{A}_0$. We assume that V is also a part of the vocabulary of $\mathcal{A}_1$. These are observable functions that we are interested in when implementing $\mathcal{A}_0$ by $\mathcal{A}_1$.

We suppose that each occurrence of an update in $\mathcal{A}_0$ has its unique name. Below, when we speak about an update, we tacitly refer to its name.

A *separating sequence* of a run is a sequence of open intervals, each one containing only one time instant with updates of functions from V. If σ is such a sequence then it may be finite or infinite, and its domain $dom(\sigma)$ is a prefix of $\mathbb{N}$. We assume that for any $i, (i+1) \in dom(\sigma)$ the *right end* $\sigma(i)^{(r)}$ of interval $\sigma(i)$ is strictly smaller then the *left end* $\sigma(i+1)^{(l)}$ of the next one: $\sigma(i)^{(r)} < \sigma(i+1)^{(l)}$, i.e., $\sigma(i)$ is *strictly monotone*.

A pair (ρ, σ), where ρ is a run and σ is one of its separating sequences is a *run with separated updates*.

If the length of each interval of σ is less than $\varepsilon \in \mathbb{R}_{>0}$ then we say that σ is *ε-separating sequence* for this interpretation.

Remark that an interval of a separating sequence may have updates of several functions, but all of them take place at the same time instant. The set of functions whose values are updated in $\sigma(i)$ in a run φ will be denoted by $\Theta[\varphi, \sigma](i)$, and the instant where the functions from $\Theta[\varphi, \sigma](i)$ are updated in $\sigma(i)$ will be denoted by $\tau[\varphi, \sigma](i)$ (some arguments may be omitted if they are clear from the context). The value of a function or of a formula F on a run φ at instant t will be denoted by $F[\varphi](t)$.

Consider two pairs (φ_0, σ_0) and (φ_1, σ_1), where σ_0 and σ_1 are ε-separating sequences for timed interpretations φ_0 and φ_1 of functions in V. Denote $\Theta[\varphi_k, \sigma_k](i)$ and $\tau[\varphi_k, \sigma_k](i)$ by respectively $\Theta_k(i)$ and $\tau_k(i)$, $k = 0, 1$.

Definition 1 *A partition π_1 of $dom(\sigma_1)$ is* admissible *for a separating sequence σ_0 if*

- *$dom(\pi_1) = dom(\sigma_0)$,*
- *each $\pi_1(j)$ is a non empty set of consecutive integers from $dom(\sigma_1)$,*
- *if $\pi_1(j) = \{m, m+1, \ldots, m+s_j\}$, $s_j \geq 0$, the intervals $\sigma_1(i)$, $i \in \pi_1(j)$ are inside $\sigma_0(j)$, but the ends are the same: $\sigma_1(m)^{(l)} = \sigma_0(j)^{(l)}$ and $\sigma_1(m+s_j)^{(r)} = \sigma_0(j)^{(r)}$.*

We suppose that below ε, $\eta \in (0, 1)$. For predicates η-closeness means that the values are equal, and for real valued functions η-closeness means that absolute value of the difference of values is less than η.

Definition 2 *Pairs (φ_1, σ_1) and (φ_0, σ_0) are (ε, η)-close if σ_0 is an ε-separating sequence and there exists an admissible partition π_1 of $dom(\sigma_1)$ for σ_0, such that*

– for each $j \in dom(\sigma_0)$, if $\pi_1(j) = \{m, m+1, \ldots, m+s_j\}$, $s_j \geq 0$, then the functions $\Theta_0(j)$ are distributed in intervals $\sigma_1(i), i \in \pi_1(j)$, i. e. $\Theta_0(j) =$

$\bigcup_{i\in\pi_1(j)} \Theta_1(i)$ as well as their instants of updates, and the values of respective functions are η-close in $\sigma_0(j)$ at $\tau_0(j)$, and at the respective $\sigma_1(i)$ at $\tau_1(i)$ (respective means the identity of names of updates — see the beginning of the section);

– for each interval $\sigma_1(k)$ of σ_1 with $k \in \pi_1(j)$, there holds $\Theta_1(k) \subseteq \Theta_0(j)$ and the values of respective functions are η-close in $\sigma_0(j)$ at $\tau_0(j)$ and at the respective $\sigma_1(k)$ at $\tau_1(k)$.

Remark 1 Notice that since σ_0 is an *ε-separating* sequence for φ_0 then σ_1 is also an *ε-separating* sequence for φ_1. Moreover the time instants of the respective updates are ε-close.

By $\mathrm{proj}_W(\varphi)$ we denote *projection* of an interpretation (in particular, of a run) φ over some vocabulary onto its sub-vocabulary W. Projection, as usually, means that we delete from φ all the functions that are not in W. Clearly, if (φ_1, σ_1) and (φ_0, σ_0) are (ε, η)-close over a vocabulary W_0 then the same takes place over vocabulary $W \subseteq W_0$ for $(\mathrm{proj}_W(\varphi_1), \sigma_1)$ and $(\mathrm{proj}_W(\varphi_0), \sigma_0)$.

Let $\mathcal{A}_0$ and $\mathcal{A}_1$ be 2 machines, the vocabulary of dynamic abstract functions of $\mathcal{A}_0$ be V, let it be a sub-vocabulary $\mathcal{A}_1$. We suppose that the inputs of two machines are the same, as well as the initial values of functions of V. We suppose that $\mathcal{A}_0$ is a IA-machine, and thus, given an input, its run is determined uniquely. But this is not the case for $\mathcal{A}_1$ because of non deterministic delays.

Definition 3 *A DA-machine $\mathcal{A}_1$ is a (ε, η)-implementation of an IA-machine $\mathcal{A}_0$ if for every inputs and initial values, for every run ρ_1 of $\mathcal{A}_1$ with these inputs and initial values, there exist a separating sequence σ_1 for $\mathrm{proj}_V(\rho_1)$, a separating sequence σ_0 for the run ρ_0 of algorithm $\mathcal{A}_0$, uniquely defined by the given inputs and initial values, such that (ρ_0, σ_0) and $(\mathrm{proj}_V(\rho_1), \sigma_1)$ are (ε, η)-close.*

4 Robust Real Time Machines

For a given IA-machine $\mathcal{A}_0$ and $\varepsilon, \eta \in (0, 1)$ we take as $\mathcal{A}_1$ the DA-machine described in section 2 as presumable implementation. To ensure that our construction of $\mathcal{A}_1$ is an (ε, η)-implementation of $\mathcal{A}_0$ we impose some constraints on $\mathcal{A}_0$, and define an upper bound on ξ that characterized the speed of $\mathcal{A}_1$. These constraints are formulated in terms of $\mathcal{A}_0$. We suppose for simplicity that input functions are predicates.

The updates of$\mathcal{A}_0$ are of the form $\alpha := \beta$ where α, β are boolean valued functions or of the form

$$g := c_0 \cdot CT + c_1 \cdot f_1 + \cdots + c_q \cdot f_q \qquad (*)$$

where $c_i, i = 0 \ldots q$ are rational numbers and $(f_i)_{(i=1\ldots q)}$ are real valued functions.

A literal constituting a guard G_i is either a predicate or its negation or an inequality, maybe involving CT:

$$b_0 \, \omega_0 \, CT + a_1 h_1 + \cdots + a_r h_r \, \omega_1 \, b_1 \ (**)$$

where $\omega_0, \omega_1 \in \{<, \leq\}$, a_i, b_j are rational numbers, $i = 1 \ldots r, j = 0, 1$, and $(h_i)_{(i=1\ldots r)}$ are real valued functions (without loss of generality, the coefficient of CT is 1).

Denote by p be the maximal number of real valued functions involved in the guards of the form (**) and the updates of the form (*), and by K be the maximum of absolute values of a_i, c_i and 1.

In the initial algorithm $\mathcal{A}_0$ a real valued function is updated with computed real value involving current time CT. In the implemented algorithm $\mathcal{A}_1$ it is updated with some value $\widetilde{CT}$ of current time caught during the the first phase of the external loop of $\mathcal{A}_1$. In order to avoid an accumulation of time shifts implied by the difference between the value of CT of $\mathcal{A}_0$ and the corresponding value $\widetilde{CT}$ of $\mathcal{A}_1$, we suppose that the real-valued functions are regularly set to a default value in a synchronized way. For simplicity, we give below a formulation that is too restrictive. We need only that only 'dependent' functions have this property. In the controller $\mathfrak{C}_0$ all real-valued functions are 'independent'.

A time instant t is a *default values instant* if all the functions have their default value at t. We suppose that there is a natural number ν with the following property:

Assumption 1 *After any time instant t_0 that is either the initial instant* 0 *or a default values instant there can be not more than ν updates before the next default values instant.*

Other constraints of $\mathcal{A}_0$ concern separation of instants value changes and independence of updates done simultaneously.

Definition 4. For a formula or term F, a run ρ and a time instant t we define (ρ, t)*-interval* is the interval $\{\tau : F[\rho](\tau) = F[\rho](t)\}$. In other words, it is the maximal interval containing t where the value of F remains unchanged. For a guard G we define (ρ, t)*-interval* as intersection (ρ, t)-intervals of literals of G.

The parameters K, p and ν permits to choose the value ξ of delay. We choose a positive ξ such that (here C is a positive constant that we do not make precise)

$$\boxed{\begin{array}{l} \xi < \dfrac{\eta}{C(\nu-1)\cdot p^{(\nu-1)}K^{2(\nu-1)}} \\ \xi < \dfrac{\varepsilon}{2\cdot CKp\cdot[(\nu-1)\cdot p^{(\nu-1)}K^{2(\nu-1)}]} \end{array}} \qquad (\nabla)$$

Assumption 2. (*Assumptions on the runs of* $\mathcal{A}_0$) For any run ρ of $\mathcal{A}_0$:

— For any two inputs predicates F_1 and F_2 which occur in one guard and for any time instants t_1 and t_2, such that $|t_1 - t_2| < \xi$, where they are respectively true in ρ, the length of the intersection of (ρ, t_1)-interval for F_1 and that of (ρ, t_2)-interval for F_2 is greater than 2ε.

— For any t and any guard G, the length of (ρ, t)-interval of G is at least 2ε.

— For any interval (t, t'), where a guard G is false in ρ, if the set E of (ρ, t)-intervals of literals of G contain at least two intervals σ such that: (i) a literal of G is true in σ,

(ii) σ lies to the right of t,

then among the intervals of E there are two that are 2ε-separated.

— Two different time instants where two guards are true are 2ε-separated (in fact we need this separation only for 'dependent' guards, but we take a simplified version).

— For any time moment t the set $\mathcal{G}$ of guards that are true at t is such that for any sub-set $\mathcal{G}'$ of $\mathcal{G}$ the values obtained after updates fired by guards from $\mathcal{G}'$

(a) does not affect the evaluation of guards from $\overline{\mathcal{G}'} = \mathcal{G} \setminus \mathcal{G}'$ that remain true on the interval $[t, t + 2\varepsilon]$,

(b) the guards from $\mathfrak{G}'$ (evaluated with the obtained values) are false on $(t, t + 2\varepsilon]$.

— For any time moment t the set $\mathcal{G}$ of guards that are true at t is such that no function appears in the right and left side of two different updates fired by guards of $\mathcal{G}$.

Though Assumption 2 looks clumsy, if we omit any constraint, the implementation construction becomes incorrect.

Definition 5. A IA-machine $\mathcal{A}_0$ is (ε, η)-*robust* if it satisfies Assumptions 1, 2.

Theorem 1 *For every IA-machine $\mathcal{A}_0$ that is (ε, η)-robust the construction of $\mathcal{A}_1$ of section 2 gives an (ε, η)-implementation if the delay ξ is chosen as described above by* (∇).

5 Preservation of Properties and Other Questions

It is clear that an (ε, η)-implementation $\mathcal{A}_1$ (here we speak about arbitrary implementations), in the general case, does not verify the requirements for the initial machine $\mathcal{A}_0$. To preserve the requirements under (ε, η)-implementations, the requirements should be in some sense redundant. We illustrate it as follows. Denote by V° a timed version of V, i.e. the vocabulary constituted by functions $f^\circ : \mathcal{T} \to Z$, where $f^\circ :\to Z$ is from V.

Consider a property of the form $\forall T \bigvee_i \mathcal{K}_i(T)$, where T is a list of time variables, and $\mathcal{K}_i$ is a conjunction of literals of the timed version V° of the vocabulary V and of inequalities involving real-valued function. A typical safety property is of this form. For a given run ρ_0 of $\mathcal{A}_0$ and T_0, a connected component of $\{T : \mathcal{K}_i(T)\}$ that contains T_0 is a polyhedron. These polyhedra covers the space of T's. If we take a run ρ_1 of $\mathcal{A}_1$ the respective polyhedra can deflate by some value defined by ε, η) and other parameters, like in the previous section. Thus, to preserve the property for any (ε, η)-implementation such a deflation should give polyhedrons that again cover the whole space of T. So to preserve a property verified by $\mathcal{A}_0$ we can inflate the polyhedra. Sometimes this can be done by a syntactical transformation of the initial conjunctions $\mathcal{K}_i(T)$. The main difficulty comes from the real-valued functions.

The notion of robustness that we described needs, on the one hand, an elaboration to make it less restrictive and more compact, and on the other hand, to make it more practical, sufficient syntactical conditions of robustness would be useful.

References

[BS02] D. Beauquier and A. Slissenko. 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, 2002.

[GH96] Y. Gurevich and J. Huggins. The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In H. K. Buening, editor, *Lecture Notes in Computer Science, Computer Science Logics, Selected papers from CSL'95*, volume 1092, pages 266–29. Springer-Verlag, 1996.