Verification of Timed Algorithms: Gurevich Abstract State Machines versus First Order Timed Logic

Danièle Beauquier[1]
Dept. of Informatics,
University Paris-12, France

Anatol Slissenko[2†]
Dept. of Informatics,
University Paris-12, France

Abstract. We give a survey of our recent and current work on the specification and verification of timed algorithms within a rather complete logical framework where timed Gurevich Abstract State Machines (GASM) are used to specify algorithms. The time we use is continuous as it is usual in specifications of control systems and often of protocols (discrete time can be treated within the same framework but it is less intuitive and harder to treat when automating the verification process). The topics addressed in this survey concern semantics of timed GASM, representation of this semantics in a first order logic adapted to verification, heuristic considerations that are implied by the used type of GASM and that help to automate the verification.

1 Introduction

We give a survey of our recent and current work on the specification and verification of timed algorithms within a rather complete logical framework where timed Gurevich Abstract State Machines (GASM) are used to specify algorithms. The time we use is continuous as it is usual in specifications of control systems (and often of protocols) that are our motivation.

Any textbook on software engineering (see e. g. [Som92]), whatever be the models of software development process it considers, distinguishes in its process requirements specification phase, algorithm specification phase and refinement phases. The verification is a method of software validation that checks whether the algorithm specification satisfies the requirements specification. The requirements specification, even if not well distinguished in practice, is usually declarative and close to a natural language. The algorithm specification is usually imperative and is done in terms of some specification language which is easy to understand and to use, and the latter feature constitutes a crucial property of such a language. From the algorithm specification one goes down via consecutive refinements to an executable program. The verification starts from verifying

[1] *Address:* Dept. of Informatics, University Paris-12, 61 Av. du Gén. de Gaulle, 94010, Créteil, France. *E-mail:* beauquier@univ-paris12.fr

[2] *Address:* Dept. of Informatics, University Paris-12, 61 Av. du Gén. de Gaulle, 94010, Créteil, France. *E-mail:* slissenko@univ-paris12.fr

[†] Member of St Petersburg Institute for Informatics and Automation of Russian Academy of Sciences.

whether the initial algorithm specification satisfies the requirements specification, and continues towards ensuring this for the refinements.

A formal verification should produce a *proof* and hence, it demands that the requirements as well as the algorithm be embedded in one logic framework. The following properties are indispensable for persuasive verification: the verification process should be

• **Complete:** The formalized requirements must completely represent the initial ones. The used formalism must permit to represent the executions of the algorithm completely.

• **Direct:** The formalized requirements, and well as the algorithm, must be a *direct* rewriting of the initial ones without any modifications that are not formally justified.

• **Conservative:** The verification of a refinement must preserve the verification of the algorithm itself.

One can hardly believe that these properties are achievable entirely in practice. We state these three properties as an idealized goal and as a basis to compare different approaches to the verification problem.

We are motivated by verification of real-time systems that consist of distributed processes each of them being not too complicated, but their interaction may pose serious problems to solve. A particular feature of this problem is the crucial role of explicit time constraints and time functions involving, for example, arithmetical operations.

Real-time systems of control are usually specified in terms of continuous time. That is why the time chosen in this paper is continuous. Moreover, continuous time is very intuitive — many informal specifications and proofs concerning distributed systems of control use time even if it is not quite necessary.

Notice that continuous time has, in a way, more efficient algorithmics than discrete time, for example, the theory of real addition has smaller complexity than the theory of integer addition (Presburger arithmetics), the theory of real addition and multiplication (Tarski algebra) is decidable and the theory of integer addition and multiplication (formal arithmetics) is even incomplete.

How does look the verification problem from the point of view of logic? Given requirements and an algorithm, the verification consists of proving, first, that the algorithm has a run for every "admissible" input (i. e. satisfying the requirements on the environment) and, second, that every run satisfies the requirements. The first of these two questions (which demands a second order logic to formalize it) is usually not considered, and its general formal treatment remains an open problem. In concrete situations, for well structured algorithms it is usually not hard to prove.

The second question is the most essential point of verification. It can be formalized as establishing the validity of some sentence of the form

$$(\Phi_{Runs} \wedge \Phi_{Env}) \rightarrow \Phi_{Func},$$

where Φ_{Func} describes the requirements on functioning such as safety or liveness, Φ_{Env} describes the environment (constraints on inputs, relations between them etc...) and Φ_{Runs} represents the runs of the algorithm to verify.

To provide a process which is complete, direct and conservative one clearly needs to have rather powerful tools to describe requirements and algorithms. This means that model-checking tools with all their advantages concerning particular questions cannot cover the full-scale verification as they are based on formalisms with bounded expressive power and that are not quite convenient from user's viewpoint, e. g. on temporal or duration logics to specify requirements (see e. g. [Pnu77], [CHR91], [MP92],[Han94], [Var96], [Rab98a]) and, say, on timed automata to specify algorithms [AD94], [LSVW96], [ACH$^+$95]. So we are approaching the verification problem from the same side as, e. g. PVS [PVS], but paying more attention on foundations and algorithmic efficiency. Sure, this approach may profit from model checking algorithms as well.

In our approach to the verification we do not invent new languages, but use universal logical principles to embed this or that particular verification problem in a logical framework trying to remain as much as possible within first order applied logics as the latter are easier to analyze algorithmically and easier to use practically. From this point of view, GASMs represent a quite adequate methodological setting as algorithm specification language for real-time distributed algorithms in particular. The fact that GASMs are syntactically and semantically close to first order logic permits to embed them into such a logical framework working out only questions that are essential indeed.

In the present paper our logic of verification is the First Order Timed Logic (FOTL) with explicit continuous time that we started to study in [BS97b,BS97a]. A concrete FOTL is an extension of a decidable theory of reals by timed predicates and functions. For a large class of problems we can state the following properties of such a logic.

First, concerning the requirements specification, this framework is more complete, direct and conservative than most part of other approaches known to us. It is sufficiently expressive from the user's point of view to rewrite *directly* and *entirely* the requirements specification of the problem under consideration usually given in a language close to the natural one.

For example, the property "two x-events are never separated by exactly 1 time unit" can be directly rewritten as the FOTL formula $\neg\exists\, tt'\,(x(t) \wedge x(t') \wedge |t-t'| = 1)$. Another example, the property "the average value of the clocks $x_1, ..., x_n$ does not exceed d" can be directly rewritten as the formula $\forall t\,(x_1(t) + ... + x_n(t) \leq n \cdot d)$ with t being a time variable, d being a variable or constant and n being a constant. The presence of arithmetics permits to easily specify also such problems as clock synchronization, that is impossible for commonly used temporal logics and timed automata.

Second, FOTL permits to represent rather easily, in fact automatically, the set of runs of timed programs (e. g. the runs of Gurevich Abstract State Machines [BS97a] or timed automata [BS98]).

Third, we can describe decidable classes of the verification problem based on the fact that the underlying theory of reals is decidable [BS99a]. And we have in fact only one logic to consider as compared with numerous temporal logics. (The unifying framework [HR98,HR99] for temporal logics or the version of Büchi's second order monadic logic for continuous time [Tra98,Rab98b,Rab97] neither give sufficient power of expressibility, though preserve the decidability.)

The structure of the paper is as follows. In section 2 we define the syntax and semantics of block GASM's for continuous time, illustrate them by the RailRoad Crossing Problem and discuss open questions which arise with continuous time. Section 3 is devoted to the logic FOTL. We explain here how one can write a formula of this logic which characterizes the set of total runs of a block GASM. The entire specification of the RailRoad Crossing Problem in this logic is given. In the last section, we mention our PVS proof for the RailRoad Crossing Problem and heuristic considerations that can be used in automized search for verification proofs within our setting.

2 Timed Gurevich Abstract State Machines (GASM)

We briefly describe a simple type of GASM namely block ones.

2.1 Syntax and semantics of a block GASM

Syntax
The vocabulary W consists of a finite set of *sorts*, and of a set of *function symbols* (predicates are treated as a particular case of functions). To each sort there is attributed a set of variables, the sets attributed to different sorts are disjoint.

The sorts are classified as *predefined* and *abstract* ones. The *predefined* sorts are those which interpretation is fixed. Here we consider only finite *abstract* sorts.

We limit predefined sorts to the following ones:
• *real numbers* $\mathbb{R}$,
• *time* $\mathcal{T} =_{df} \mathbb{R}_{\geq 0}$ treated as a subsort of $\mathbb{R}$,
• *boolean values Bool*
• finite sets of given cardinality
• finite unions of the sorts mentioned above.

We explicitly mention just *variables for time*: t and τ with indices. For other sorts we will use any letter with explicit indication of its sort. For example, $\forall X \in \mathcal{X}$, where X is a list of n variables and $\mathcal{X}$ is a direct product of n sorts, will mean that ith variable of X is of the ith sort of $\mathcal{X}$.

As usually, each function has its type (profile) which determines also its arity.
The functions are classified as *predefined* and *abstract*.
The *predefined* function symbols have a fixed given interpretation. We take as predefined functions the following ones:
• Boolean constants $true$ and $false$,
• rational numbers $\mathbb{Q}$ each of the type real,
• addition $+$ and subtraction $-$ of reals, and infinite number of unary multiplications by rational numbers,
• usual binary predicates over reals: $=$, $\leq$, $<$.

We suppose that the type of any *abstract* function is of the form $\mathcal{X} \to \mathcal{Z}$ where $\mathcal{X}$ is a finite product of finite sorts. The functions are also classified as *dynamic* or *static*. The dynamic functions are those which interpretation depends on time as it will be explained later. At last, functions are classified as *internal* or *external*. Internal functions are computed by the GASM, and they are, obviously, abstract and dynamic. External functions represent the inputs of the GASM. We denote by V_{Extrn} and V_{Intrn} respectively the sets of external and internal functions , and $V =_{df} V_{Extrn} \cup V_{Intrn}$.
Among external functions, there is a predefined dynamic one that is CT, representing the current time. The type of CT is $\to \mathcal{T}$. Predefined static functions have a fixed interpretation valid for every $t \in \mathcal{T}$. The interpretation of a predefined dynamic function, though changing with time, does not depend on the functioning of the machine.

We assume that the *equality* is defined for each abstract sort.

A vocabulary W being fixed, the notion of *term* and that of *formula* over W are defined in a usual way.

To define a block GASM we need the notion of *rule*:

• Any expression of the form $f(\theta_1, \ldots, \theta_k) := \theta$, where f is an internal function, and $\theta_1, \ldots, \theta_k$, θ are terms of the appropriate types, is an *update* rule (in other words, an assignment). The rule is executed instantaneously. This is the assumption of *instantaneous actions*.

• Any expression of the form $\alpha_1; \ \ldots \ ; \alpha_m$ where $\alpha_1, \ldots, \alpha_m$ are rules is a *block*. If $\alpha_1, \ldots, \alpha_m$ are update rules then the block is an *update block*. Informally, we assume that all rules α_i are executed simultaneously.

• Any expression of the form **If** G **Then** A **EndIf**, where G is a formula (called here a *guard*) and A is a block, is a *conditional* rule. If A is an update block then the conditional rule is a *conditional update rule*.

A *block program* has the form:

```
Repeat
  ForAll $\omega \in \Omega$
    InParallelDo
      If $G_1(\omega)$ Then $A_1(\omega)$ EndIf
      If $G_2(\omega)$ Then $A_2(\omega)$ EndIf
      ...........
      If $G_m(\omega)$ Then $A_m(\omega)$ EndIf
    EndDo
  EndForAll
EndRepeat
```

where ω is a variable of a finite abstract sort Ω, each G_i, $1 \leq i \leq m$, is a guard not having free variables different from ω, and each A_i is an update block.

A *block GASM* over W is a triple of the form $(W, \mathit{Init}, \mathit{Prog})$, where W is a vocabulary, Init is a closed formula over W describing the initial state and Prog is a program of the just described form. The formula Init is presumed to have the property: given an interpretation of abstract sorts and abstract external functions for time 0, there is a unique interpretation of internal functions such that the condition $Init$ is satisfied. This unique interpretation of internal functions defines their value at time 0.

Semantics
One can define semantics of timed algorithms in several ways that are "physically" equivalent but are formally different and either impose some constraints on the guards of the algorithms or not. We consider here a version from [BS99b] that formalizes the intuition given in [GH96], and at the end of this section we will mention other possibilities. Informally, the external **Repeat** means that guards are evaluated permanently, and **InParallelDo** means that all the guards are evaluated in parallel. If some guards become true at some moment, then the assignments related to these guards are done. We give now a formal definition of the semantics.

Consider a block GASM $(W, Init, Prog)$ with the program $Prog$ as given above. Denote by W_k the set of terms that appear to the left of $:=$ in the update block $A_k(\omega)$, and denote by $\theta_{k,v}$ the term of the assignment of $A_k(\omega)$ with the left hand side v. Without loss of generality, we assume that there are no two assignments of the form $v := \theta$ and $v := \theta'$ in $A_k(\omega)$. Thus, the assignments of $A_k(\omega)$ are of the form $v := \theta_{k,v},\ v \in W_k$.

As it was remarked above, informally speaking, all the **If–Then** conditional rules (statements) are executed simultaneously as well as all the assignments in any **Then**-part if the corresponding guard is true. Sure, if the assignments are inconsistent, the execution is interrupted, and the run of the algorithm becomes undefined.

For a given interpretation of abstract sorts we can define the semantics of the program in terms of runs. Informally, given an input, that is an interpretation of external functions for each moment of time, the machine computes a run ρ which is an interpretation of internal functions for each moment of time or at least for an initial segment of $\mathcal{T}$. Notice that the external functions which are classified as static have the same interpretation for every moment of time. The interpretation of function CT at time t is t, for every $t \in \mathcal{T}$.

Now suppose that an interpretation V^*_{Sorts} of abstract sorts is given. Let V^*_{Extrn} (resp. V^*_{Intrn}), be the set of possible interpretations of external (resp. internal) functions.

Consider an input $\mathcal{E} : \mathcal{T} \to V^*_{Extrn}$; $\mathcal{E}(t)$ is the set of values of external functions at the moment t, and we denote by $\rho(t)$ the set of values of internal functions at the moment t. We are going to define $\rho(t)$ by a recursion. Together with $\rho(t)$, we define a sequence of time moments $\{T_i\}_{i>0}$, where some guards of the GASM become valid, and a sequence $\{\rho_i\}_{i>0}$ of updated values of ρ induced by these valid guards (at time 0 there may be no update, so ρ_0 will be defined in a special way). In fact, the new value ρ_i will be assigned to ρ to the right of the moment $\{T_i\}$. Such ρ_i will be used as an *extension of the run to the right of* T_i.

We denote by $Dom(\rho)$ the (initial) segment of $\mathcal{T}$ where ρ is defined.

The input $\mathcal{E}(0)$ and the proposition $Init$ uniquely define $\rho(0)$. To avoid repetitions we will define ρ_0 later.

Set $T_0 = 0$. Suppose that $0 \leq T_1 < \ldots < T_n$ are the first n update times, (T_0 is a particular case which may be an update time or not, we will see it later) and the $\{\rho_i\}$ are defined for $0 \leq i \leq n$. Suppose that run ρ is defined on $[0, T_n]$. Now we introduce some auxiliary total run φ_n extending ρ to the right of T_n as ρ_n:

$$\varphi_n(\tau) = \begin{cases} (\mathcal{E}(\tau), \rho(\tau)), & \text{if } \tau \in [0, T_n], \\ (\mathcal{E}(\tau), \rho_n), & \text{if } \tau > T_n \end{cases} \tag{1}$$

This $\varphi_n(\tau)$ is an interpretation of V. Let τ be fixed. For a formula F or a term θ over W denote by $F[\varphi_n(\tau)]$ and $\theta[\varphi_n(\tau)]$ respectively the sentence and the constant term (that has now some concrete value) obtained from F and θ by replacing all function symbols by their interpretations given by $\varphi_n(\tau)$.

Look for

$$T_{n+1} = \inf\{\tau > T_n : \text{ for some } \omega \text{ and } k \text{ the guard } G_k(\omega)[\varphi_n(\tau)] \text{ is valid}\}. \tag{2}$$

If $T_{n+1} = \infty$ (we assume that $\inf \emptyset = \infty$) then $\rho(\tau) = \rho_n$ for all $\tau > T_n$. Thus $Dom(\rho) = \mathcal{T}$ and the sequence of update times is, by definition, $(T_i)_{i=1,\ldots,n}$(and $T_{n+1} = \infty$).

Suppose $T_{n+1} < \infty$.

If $T_{n+1} = T_n$ (Case 1) or $\forall \omega \bigwedge_k \neg G_k(\omega)[\varphi_n(T_{n+1})]$ (Case 2) then the execution stops at time T_n, $Dom(\rho) = [0, T_n]$ and the sequence of update times is $(T_i)_{i=1,\ldots,n}$. We will give in subsection 2.3 examples of machines for which these cases appear.

Suppose $T_n < T_{n+1}$ and $\exists \omega \bigvee_k G_k(\omega)[\varphi_n(T_{n+1})]$.

Let

$$K =_{df} \{(k, \omega) : G_k(\omega)[\varphi_n(T_{n+1})]\}.$$

Consider an update $f(\theta_1, \ldots, \theta_n) := \theta_0$ from any $A_k(\omega)$, $(k, \omega) \in K$. The interpretation $\varphi_n(T_{n+1})$ gives some value $a_i^* = \theta_i[\varphi_n(T_{n+1})]$ to θ_i. Thus, this update for the interpretation $\varphi_n(T_{n+1})$ means that the value of the function f for the arguments $a_1^*, \ldots, a_n^*$ becomes equal to a_0^* *after* time T_{n+1}. There may be several updates in the set $\{A_k(\omega)[\varphi_n(T_{n+1})]\}_{(k,\omega) \in K}$ concerning f for the same value $a_1^*, \ldots, a_n^*$. If these updates are inconsistent the run is interrupted at T_{n+1}, $Dom(\rho) = [0, T_{n+1}]$, the sequence of update times is $(T_i)_{i=1,\ldots,n+1}$ and $\rho(\tau) = \rho_n$ for $\tau \in (T_n, T_{n+1}]$.

Suppose that updates are consistent. Hence, they determine a new value of internal functions that we denote by ρ_{n+1}. By default, the values not touched by the updates from the mentioned set remain unchanged.

Set $\rho(\tau) = \rho_n$ for $T_n < \tau \leq T_{n+1}$. And thus, the run is defined on $[0, T_{n+1}]$, the sequence of update times on this interval is $(T_i)_{i=1,\ldots,n+1}$ and the value of the extension of ρ at the right of T_{n+1} is ρ_{n+1}.

We have to initialize the recursion.

The run ρ is defined on $[0, T_0]$, and the extension ρ_0 of ρ to the right of T_0 depends on the evaluation of the guards at time 0.

Set $\phi(0) = (\mathcal{E}(0), \rho(0))$.

• If there are no ω and no k such that the guard $G_k(\omega)[\phi_0(0)]$ is valid, then the extension ρ_0 of ρ to the right of T_0 is defined as equal to $\rho(0)$.

• If there exists some ω and some k such that the guard $G_k(\omega)[\phi_0(0)]$ is valid, then the updates are defined as above. If some updates are inconsistent, the run is just defined in 0. If the updates are consistent they define a new value ρ_0 for the extension of ρ at the right of 0.

Hence, the recursive definition of update times is accomplished.

Let $\overline{T} =_{df} \sup_{0 \leq i} T_i$. The domain of ρ is $Dom(\rho) = [0, \overline{T})$, and the run $\rho : Dom(\rho) \to V^*_{Intrn}$ is defined.

If the sequence (T_i) is infinite and $\overline{T}$ is finite then the run is *Zeno*.

If $\overline{T} = \infty$ then the domain of ρ is $\mathcal{T}$ and ρ is a *total* run.

It can be easily proved that for a given input $\mathcal{E} : \mathcal{T} \to V^*_{Extrn}$, a function $\rho : \mathcal{T} \to V^*_{Intrn}$ is a total run for this input iff the interpretation $\varphi = (\mathcal{E}, \rho)$ satisfies the following property:

there exists a strictly increasing sequence $(T_i)_{0 \leq i < N}$ of time moments (N may be infinite) with $T_0 = 0$ such that:

– ρ is equal to some constant ρ_i on $(T_i, T_{i+1}]$ if $T_{i+1} < \infty$ and on (T_i, T_{i+1}) if $T_{i+1} = \infty$ for every $0 \leq i < N$.

– for every $0 < i < N$, there is some guard $G_k(\omega)$ valid at time T_i for the interpretation φ and the value ρ_i is the result of the updates done as above.

– if no guard is valid at time 0 for the interpretation φ then $\rho_0 = \rho(0)$ otherwise ρ_0 is the result of the updates done at time 0.

– for every time moment $t \neq T_i, 1 \leq i < N$, no guard is valid for the interpretation φ at time t. These properties will be used in section 3.2 to write a formula describing the set of total runs.

2.2 An example : the Railroad Crossing Problem

We consider a largely studied problem involving several interacting processes, the Generalized Railroad Crossing Problem[3] introduced in [HL96]. Our analysis of this problem was inspired by [GH96]. We give an algorithm to solve the problem following the algorithm described in the mentioned paper, slightly modifying it to better meet the liveness property. We tried to give a direct and relatively complete study of first-order part of this problem in [BS99b] and before in [BS97a].

The problem is as follows. A railroad crossing has several parallel train tracks and a common gate. Each track admits in each direction two sensors, one at some distance of the crossing in order to detect incoming of a train and another one just after the crossing in order to detect the train is leaving (see Figure 1 where one-directional situation is shown). An automatic controller receives the signals from the sensors and on the basis of these signals, decides to send to the gate a signal *close* or *open*. The correctness requirements to satisfy by the controller (i. e. by the algorithm to construct) are the following ones:

[3] The Generalized Railroad Crossing Problem seems to be a good introductory example and is widely exploited, e. g. see a recent collection of papers [HM96] where it serves as a common example.

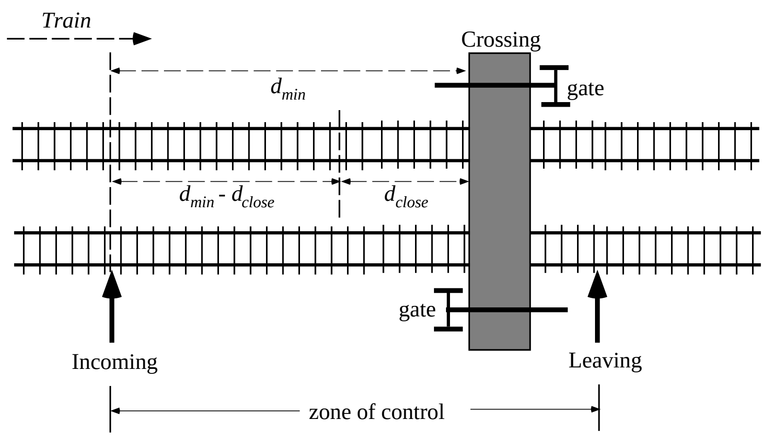

Fig. 1. Railroad Crossing.

Safety. *If a train is in the crossing, the gate is closed.*

Utility (Liveness). *The gate is open as much as possible.*

Notice that Safety alone is easy to satisfy with the gate always closed. On the other hand, the formulation of Utility is clearly vague. It demands to optimize some functioning, clearly, under condition that Safety takes place. If to formalize this directly we are to say that the demand is to construct a controller such that whatever be any other controller satisfying the environment constraints (to discuss below) and Safety and whatever be an input, the time interval when the gate is opened for the constructed controller is not less than this time interval for arbitrary controller. Such a formulation involves second order quantifiers, and the existence of a demanded controller is not evident. We assume that (Utility) is reasonably reformulated in a first order logic as in [BS99b]. The (Utility) in the initial formulation has never been studied and hardly worth to be studied (the solutions of [HL96,GH96] trivially do not satisfy this property).

Some assumptions are usually done. Without loss of generality, the tracks are presumed to be one directional. It is assumed that a train cannot arrive on a track (i. e. in the zone of control) before the previous one has left this track. The situation when a train does not leave the crossing is not formally excluded. It takes at least time d_{min} for a train to reach the crossing after the sensor has detected its incoming. And it takes at most d_{open} (respectively d_{close}) to the gate to be really opened (respectively closed) after the reception of signal to open (respectively, to close) if the opposite signal has not been sent in between. To exclude degenerated case, it is assumed that at least $d_{close} < d_{min}$.

One can directly formalize the requirements in a first order applied logic modulo our remark on (Utility) [BS99b]. This rather short description of requirements on environment and functioning is direct and complete. We do not touch it here as our main subject concerns GASM.

A GASM controller is in Fig. 2, where the following notations are used. We have a finite set $Tracks$ of tracks of unknown cardinality. The *current time* is represented by identifier (nullary function) CT. Predicate $Cmg(x)$ (Cmg for Coming) says that a train on a track x has been detected; $Emp(x) =_{df} \neg Cmg(x)$ (Emp for Empty). This predicate is an input, and thus an external function. To catch the moment when to start to close the gate the Controller uses its internal function $DL(x)$ (DL for Deadline), where $WT =_{df} d_{min} - d_{close}$ (WT for WaitTime). The output signal of control demanded by the specification of functioning is either $DirCl$ or $DirOp =_{df} \neg DirCl$ saying that the gate must be closed or opened. Remark that the specification gives only upper bound on the time to close/open the gate, so it can be, say, opened instantaneously. The decision to open or close the gate is being done on the basis of global condition $SafeToOpen$ which says when it safe to open the gate:
$SafeToOpen =_{df} \forall x \; \big(\; Emp(x) \vee CT < DL(x) \; \big)$. The initial values are defined by the condition $Init =_{df} \; \big(\forall x \, DL(x) = \infty \wedge DirOp \, \big)$ The algorithm is written

```
Repeat
  ForAll $x \in Tracks$ InParallelDo
    If $Cmg(x) \wedge DL(x) = \infty$ Then $DL(x) := CT + WT$ EndIf;
    If $Emp(x) \wedge DL(x) < \infty$ Then $DL(x) := \infty$ EndIf;
    If $DirOp \wedge \neg SafeToOpen$ Then $DirCl := true$ EndIf;
    If $DirCl \wedge SafeToOpen$ Then $DirOp := true$ EndIf;
  EndDo EndForAll
EndRepeat
```

Fig. 2. Railroad Crossing Controller.

in Fig. 2

The *predefined* sorts used are: time $\mathcal{T} =_{df} \mathbb{R}_{\geq}$, its extension $\mathcal{T}^{\infty} =_{df} \mathcal{T} \cup \{\infty\}$, reals $\mathbb{R}$, $\mathbb{R}^{\infty} =_{df} \mathbb{R} \cup \{\infty\}$ with natural inclusion of all these sorts to use arithmetical operations and order relations $<$, $\leq$ without special comments. (Clear, ∞ can be eliminated.) Sure, boolean values $Bool$ are also present. The only *abstract* sort is $Tracks$, and we know only that it is finite with unknown cardinality. The latter point is important: we wish to fulfil a verification without any reference to concrete values of the cardinality of $Tracks$.

In addition to sorts, the vocabulary W contains functions. Some of them are predefined as the just mentioned arithmetical ones, the other ones are abstract. We add to predefined functions ∞, rational numbers $\mathbb{Q}$ and $CT :\to \mathcal{T}$ to represent the current time. The abstract functions consist of *static*, as d_{min}, d_{close}, ∞, and *dynamic* ones, the latter are denoted by V. The external ones, as $Cmg : Tracks \to Bool$ or CT, are *inputs* that cannot be changed by the algorithm. The internal ones can be changed by the algorithm, and we further divide them into *private* functions of the algorithm, as $DL : Tracks \rightarrow \mathcal{T}^{\infty}$, and *output* ones, as $DirCl :\rightarrow Bool$. The outputs are represented in requirements as compared with private functions that are used only inside the algorithm. The functions $DirCl$ or CT, though being a constant from logical point of view, are dynamic. Such internal *nullary* functions will be called here *identifiers*, as well as restrictions of other functions for fixed values of their arguments, as $DL(x)$ for a concrete x.

2.3 Open Questions of Defining Semantics of Timed GASM

In defining semantics for continuous time there are some boring situations that are, on the one hand, of no "physical" significance but, on the other hand, are not desirable to be excluded if we wish not to care about minor details of algorithms presentation.

First example

```
$Algo1$:
**If** $CT = 1$ **Then** $flag := 1$ **EndIf**
**If** $flag = 1 \wedge x = 0$ **Then** $x := 1$ **EndIf**
```

If for some input the associated run is defined up to time 1, then 1 is an update time. In that case, if x is equal to 0 before time 1, the next possible update time is again 1, and the second update, namely $x := 1$ should be done at time 1. The run stops at this time. It corresponds to Case 1 above in the description of the different possibilities to interrupt the run. This simple example underlines the fact that an internal function cannot be used as an input to fire a guard. This is due to the fact that internal functions are constant on intervals which are left-open (it is a clear consequence of the semantics we have chosen). There are several solutions to this problem.
• A first solution is to introduce some small delay *eps* between the moment when the internal function takes a new value, and the moment when this new value can be read. From a practical point of view, this delay seems to be reasonable, but it contradicts the hypothesis of instantaneous actions. This delay can be explicit in the machine. $Algo1$ will be rewritten:

```
$Algo1bis$:
**If** $CT = 1$ **Then** $flag := 1;\ DL := CT + eps$ **EndIf**
**If** $CT = DL \wedge flag = 1 \wedge x = 0$ **Then** $x := 1$ **EndIf**
```

• Another possibility is to introduce delays in the semantics itself. This leads to the notion of asynchronous GASM developed in [CS99]
• A third possibility is to allow the machine to make instantaneously a sequence of updates. One can allow a finite sequence of updates, or also use a fixpoint. If the sequence of updates is infinite and the values of the consecutive updates

converge, then the limit will be the final result of the update. This notion of run can be defined properly again by induction. The shortcoming of this semantics is that such runs cannot be described by a first order formula of FOTL (see the next section for the definition of FOTL).
• At last, to avoid these consecutive updates at the same time, it is often possible to write in place of the initial machine a new one which "skips" the intermediate steps and provides the final result. For $Algo1$ the new machine would be the following one:

```
$Algo1ter$:
If $CT = 1 \wedge x = 0$ Then $flag := 1;\ x := 1$ EndIf
If $CT = 1$ Then $flag := 1$ EndIf
```

Second example

```
$Algo2$:
If $CT > 1 \wedge x = 0$
Then $x := 1$ EndIf
```

Suppose that for some input the run can be defined up to $CT = 1$, x is equal to 0 up to this moment and the next update time is $CT = 1$ because 1 is the minimal of time moments when the guard $CT > 1$ is valid. Nevertheless the guard is not valid at time 1, so the run stops at this time. It corresponds to Case 2 above in the description of the different possibilities to interrupt a run.
One solution is to change the guard $CT > 1$ into $CT \geq 1$. From a practical point of view, this seems to be reasonable, because these two guards cannot be distinguished by a physical process.

3 Verification in First Order Timed Logic (FOTL)

The verification, as it was noticed in the Introduction, means a proof of some properties. Thus, its rigorous treatment implies using some logic (maybe implicitly).

3.1 Syntax and Semantics of FOTL.

We define our logics as some extensions of decidable theories of reals. For a particular specification and verification problem it is reasonable to take a minimal decidable theory that suffices to rewrite directly and completely a given specification. We take here the theory of real addition and unary multiplications by rational constants. But one can take also Tarski algebra or other theories.

The vocabulary of a FOTL is constituted in a similar way as the vocabulary of timed GASM as described above. The main difference concerns the types of dynamic functions: any dynamic function is of the type $\mathcal{T} \times \mathcal{X} \to Z$, where $\mathcal{X}$ is as before a product of abstract finite sorts and Z is any sort.
A vocabulary being fixed, the notion of *term* and that of *formula* over this vocabulary are defined in a usual way.

Semantics of FOTL.

A priori we impose no constraints on the admissible interpretations. Thus, the notions of interpretation, model, satisfiability and validity are treated as in first order predicate logic modulo preinterpreted part of the vocabulary.

3.2 FOTL Representation of Runs of a Block GASM.

Remind that the main goal of representing runs in logic is to give a logical framework for the verification.
Let W be the vocabulary of a block GASM $(W, \mathit{Init}, \mathit{Prog})$ with the program of the form given above. Denote by W° the vocabulary obtained from the vocabulary W by replacing each *dynamic* function symbol $f \in W$ of the type $\mathcal{X} \to Z$ by f° of the type $\mathcal{T} \times \mathcal{X} \to Z$.

Define operation "$\widehat{\ }$" which transforms a term θ over W and $t \in \mathcal{T}$ into a term $\widehat{\theta}(t)$ over W° by the following recursion (a)–(c):

(a) $\widehat{u}(t) = u$ if u is a variable or a static function symbol (constant).

(b) For terms θ over W of the form $f(\theta_1, \dots, \theta_n)$, where f is a static function symbol,

$$\widehat{\theta}(t) = f(\widehat{\theta_1}(t), \dots, \widehat{\theta_n}(t)).$$

(c) For terms θ over W of the form $f(\theta_1, \dots, \theta_n)$, where f is a dynamic function symbol,

$$\widehat{\theta}(t) = f^\circ(t, \widehat{\theta_1}(t), \dots, \widehat{\theta_n}(t)).$$

For a formula F over W we denote by $\widehat{F}(t)$ the formula over V° obtained from F by replacing all terms θ by $\widehat{\theta}(t)$.

In such FOTL one can write a formula Ψ which models are exactly the runs of the GASM. This formula can be produced by a simple algorithm, see [BS99b]. It is a conjunction of several formulas, the initial condition among them. For example, one of these conjuncts says that if no guard is valid at t then no guard is valid in some neighborhood of t, and this can be written for $t > 0$ as :

$$\big(\, \forall \omega \bigwedge_k \neg \widehat{G_k}(\omega)(t) \,\big) \to \exists\, t_1 t_2 \,\big(\, t_1 < t < t_2 \wedge \forall\, \tau \in (t_1, t_2) \forall \omega \bigwedge_k \neg \widehat{G_k}(\omega)(\tau) \,\big)\, .$$

In a similar way we write formulas saying

– if a guard is valid at t then no guard is valid in some neighborhood of t except t itself,

– for any interval where no guard is valid, all the internal functions preserve their values,

– the values of internal functions at t are equal to their values before t in some left neighborhood of t,

– the updates made at t hold on some (t, t_1),

– the values of internal functions that are not updated at the moment t remain unchanged on some $[t, t_1)$.

3.3 FOTL-Vocabulary of the Generalized Railroad Crossing Problem

Let W° be the vocabulary obtained from the vocabulary W given in subsection 2.2. We add to W° some abstract auxiliary predicates needed to describe the environment:

• $GtClsd^\circ : \mathcal{T} \to Bool$ says that the gate is closed at a given time moment.

• $GtOpnd^\circ : \mathcal{T} \to Bool$ says that the gate is opened at a given time moment.
($GtOpnd^\circ$ is not the negation of $GtClsd^\circ$ as we know only that the gate cannot be opened and closed at the same time.)

Requirement Specifications of the Railroad Crossing Problem.

We have no formal notion of train within the given syntax. We assume that for a given track a new train reaches the sensor launching Cmg° only after the previous one has left the crossing making the track status Emp°. The alternation $Emp^\circ/\ Cmg^\circ/\ Emp^\circ \dots$ corresponds to appearance of successive trains on a given track. But we cannot express this property directly as a specification formula, and use it only indirectly to justify the control requirements below. More safe treatment of train detection is in terms of instantaneous predicates In° and Out°, that represent pointwise signals of incoming and going out of a train. Though being more difficult to treat, such representation gives more reliable specification because, in order to provide a proof of correctness, we are to state explicitly the mentioned alternation of arrivals and departures of the trains.

Notations:

• If f is a function of type $\mathcal{T} \times \mathcal{X} \to Z$

$f(t^-, X) = r$ is an abbreviation for $\exists\, \varepsilon > 0\, \forall\, \tau\, (t - \varepsilon \le \tau < t \to f(\tau, X) = r)$.

$f(t^+, X) = r$ is an abbreviation for $\exists\, \varepsilon > 0\, \forall\, \tau\, (t < \tau \le t + \varepsilon \to f(\tau, X) = r)$.

• A notion describing when the controller may open the gate is stated as follows:

$SafeToOpenSp(t) =_{df}$

$$\forall\, x\ \big[\ Emp^\circ(t, x) \vee \forall \tau \le t\ \big(\forall \tau' \in [\tau, t)\, Cmg^\circ(\tau', x) \to t < \tau + WaitTime\ \big)\ \big]\,.$$

As compared with [HL96,GH96] we do not pretend that we can prove the Utility in the initial formulation. Instead of that we take some reasonable formulation in the chosen vocabulary. And our formulation will give an algorithm with strictly better Utility than that of the mentioned papers.

The specifications that follow consist of two parts, namely, of specification of the environment and that of functioning (or control).

Generalized Railroad Crossing Problem: Specification of the Environment.

(TrStInit) $\forall\, x\, Emp^\circ(0, x)$

(At the initial moment there are no trains on any track.)

(GtStInit) $GtOpnd^\circ(0)$

(At the initial moment the gate is opened.)

(GtSt) $\forall t\, \neg(GtOpnd^{\circ}(t) \;\wedge\; GtClsd^{\circ}(t))$
(The gate cannot be closed and opened at the same time, but it can be neither opened nor closed .)

(DirInit) $DirOp^{\circ}(0)$
(At the initial moment the signal controlling the gate is opened.)

(CrCm) $\forall t\, \big(\, InCr^{\circ}(t) \;\rightarrow\; \big(\, t \geq d_{min} \;\wedge\; \exists\, x\, \forall\, \tau \in [t - d_{min},\, t]\, Cmg^{\circ}(\tau, x)\, \big)\, \big)$
(If a train is in the crossing it had been detected on one of the tracks at least d_{min} time before the current moment.)

(OpnOpnd) $\forall\, t\, \big(\, \big(\, t \geq d_{open} \wedge \forall\, \tau \in (t - d_{open}, t]\, DirOp^{\circ}(\tau)\, \big) \;\rightarrow\; GtOpnd^{\circ}(t)\, \big)$
(If at time t the command has been *open* for at least a duration d_{open} then the gate is opened at time t.)

(ClsClsd) $\forall t\, \big(\, \big(\, t \geq d_{close} \;\wedge\; \forall \tau \in (t - d_{close}, t]\, DirCl^{\circ}(\tau)\, \big) \;\rightarrow\; GtClsd^{\circ}(t)\, \big)$
(If at time t the command has been *close* for at least a duration d_{close} then the gate is closed at time t.)

(Cmg)
$$\forall x\, \forall t\, \big[\, Cmg^{\circ}(t, x) \rightarrow \exists\, t_0\, \big(\, 0 < t_0 \leq t \;\wedge\; \forall \tau \in [t_0, t]\, Cmg^{\circ}(\tau, x) \;\wedge\; Emp^{\circ}(t_0^{-}, x))\, \big)\, \big]$$

(Emp)
$$\forall x\, \forall t\, \big[\, Emp^{\circ}(t, x) \rightarrow \big[\, \forall \tau\, \big(\, 0 \leq \tau \leq t \;\rightarrow\; Emp^{\circ}(\tau, x)\, \big) \;\vee\; \exists\, t_0\, \big(\, 0 < t_0 \leq t \;\wedge\; \forall \tau \in [t_0, t]\, Emp^{\circ}(\tau, x) \;\wedge\; Cmg^{\circ}(t_0^{-}, x))\, \big)\, \big]\, \big]$$

(The two last properties express that the predicate Cmg° is true on intervals closed on the left and opened on the right and that the set of points where the value changes has no accumulation points.)

(dIneq) $0 < d_{close} < d_{min}$
(This is trivial constraint on the durations involved, the time for closing is smaller than the minimum time of reaching the crossing by any train detected as coming.)

Generalized Railroad Crossing: Specification of the Control.
These specifications concern requirements on the functioning.
Safety: $\forall t\, (InCr^{\circ}(t) \rightarrow GtClsd^{\circ}(t))$.
(When a train is in the crossing, the gate is closed).

Utility: $\forall t\, (SafeToOpenSp(t) \;\rightarrow\; DirOp^{\circ}(t))$.
(If the zone of control is safe to open at time t then the control signal must be to open the gate).

One can notice that the formalization of Safety directly follows the informal specification. On the contrary, we renounce to reformulate straightforwardly the informal description of Utility because the difficulty of proving it (the other authors [GH96,HL96] neither prove it in the initial generality). Instead of direct reformulation we take a formulation that seems natural and can be expressed in

FOTL which vocabulary corresponds to that of the informal description of the problem.

4 PVS Verification of the Generalized RailRoad Crossing Problem and Heuristics Considerations

In our formalism, a formal verification of Safety of the Generalized RailRoad Crossing Problem is rather short (about three pages, not taking in account the initial specification that takes about 2 pages) and is relatively easy to understand. Its simplicity and readability are incomparable with 73 pages paper [AH98] treating the case of one track. Moreover, the formalization of [AH98] deviates from the original formulation of the problem, without proving that this modification is correct. Recently we have checked this proof with PVS. For this purpose, we have expressed in PVS the semantics of block GASM starting from the FOTL formula describing runs that is given in [BS99b]. However, to make the description of GASM runs more practical, we have developed more detailed and efficient properties of GASM runs. The entire specification in PVS of the requirements and controller of the Generalized RailRoad Crossing Problem takes less than four pages.

Heuristics as Pattern Based Strategy.
Our current work on feasible algorithms for the verification of timed systems (with participation of our colleagues A. Durand, T. Crolard, and student F. Belloni) concerns, in particular, development of strategies for proof search oriented on the verification problems under discussion. It was noticed in [Sli99,Sli91] that a basic language to express human considerations about where and what rule to apply to advance in proof search, is a language of patterns that one can detect in the current proof search situation. This idea was implemented in [Tar96] for a particular term rewriting system related to symbolic computations and showed very encouraging results.

The analysis of our proof search and the obtained proof permits to formulate a pattern based strategy for verification of real-time block GASM's. Together with many known considerations concerning the proof search, this heuristics relies on the particular role of internal functions. The main part of this heuristics describes how to process elementary formulas containing internal functions of the GASM under consideration.

Conclusion.

Our study of the verification problem for timed algorithms, though rather limited, shows that a logic approach based in particular on GASM ideology, gives a good basis for developing practical and theoretical tools. However, the fact that even for this bounded domain there cannot be universal feasible languages to write requirements and algorithms, implies that one needs some "methodology" of producing particular tools for particular situations. To build particular tools rapidly, one needs a collection of well developed components. As an example of such building blocks we may mention a block GASM (maybe divided into some subclasses), its semantics, its logic, particular rules of inference for verification proofs, and a collection of patterns and basic strategies to construct more specific and efficient strategies of proof search. We believe that there could be found some efficient algorithm of verification if to study carefully the model checking approach and to generalize it on a first order timed logic.

References

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[AH98] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. Technical Report 5546-98-8180, Naval Research Laboratory, Washington, 1998. NRL Memorandum Report.

[BS97a] D. Beauquier and A. Slissenko. On semantics of algorithms with continuous time. Technical Report 97–15, Revised version., University Paris 12, Department of Informatics, 1997. Available at http://www.eecs.umich.edu/gasm/ and at http://www.univ-paris12.fr/lacl/.

[BS97b] D. Beauquier and A. Slissenko. The railroad crossing problem: Towards semantics of timed algorithms and their model-checking in high-level languages. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development*, pages 201–212. Springer Verlag, 1997. Lect. Notes in Comput. Sci., vol. 1214.

[BS98] D. Beauquier and A. Slissenko. Decidable verification for reducible timed automata specified in a first order logic with time. Technical Report 98–16, University Paris 12, Department of Informatics, 1998. Available at http://www.univ-paris12.fr/lacl/.

[BS99a] D. Beauquier and A. Slissenko. Decidable classes of the verification problem in a timed predicate logic. In *Proc. of the 12th Intern. Symp. on Fundamentals of Computation Theory (FCT'99), Iasi, Rumania, August 30 – September 3, 1999, Lect. Notes in Comput. Sci, vol. 1684*, pages 100–111. Springer-Verlag, 1999.

[BS99b] D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms: Basic properties and a decidable class. Submitted., 1999.

[CHR91] Z. Chaochen, C. A. R. Hoare, and A. Ravn. A calculus of duration. *Inform. Proc. Lett.*, 40(5):269–279, 1991.

[CS99] J. Cohen and A. Slissenko. On verification of refinements of asynchronous timed distributed algorithms. Manuscript., October 1999. 15 p. Submitted.

[GH96] Y. Gurevich and J. Huggins. The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In H. K. Buening, editor, *Computer Science Logics, Selected papers from CSL'95*, pages 266–290. Springer-Verlag, 1996. Lect. Notes in Comput. Sci., vol. 1092.

[Han94] H. A. Hansson. *Time and Probability in Formal Design of Distributed Systems.* Elsevier, 1994. Series: "Real Time Safety Critical System", vol. 1. Series Editor: H. Zedan.

[HL96] C. Heitmeyer and N. Lynch. Formal verification of real-time systems using timed automata. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, pages 83–106. John Wiley & Sons, 1996. In series: "Trends in Software", vol. 5, Series Editor: B. Krishnamurthy.

[HM96] C. Heitmeyer and D. Mandrioli, editors. *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*. John Wiley & Sons, 1996. Series Editor: B. Krishnamurthy.

[HR98] Y. Hirshfeld and A. Rabinovich. Quantitative temporal logic. Manuscript, 11 p., 1998.

[HR99] Y. Hirshfeld and A. Rabinovich. A framework for decidable metrical logics. In *Proc. of ICALP'99*. Springer-Verlag, 1999. Lect. Notes in Comput. Sci.. To appear.

[LSVW96] N. Lynch, R. Segala, F. Vaandrager, and H. Weinberg. Hybrid i/o aotomata. In *Proc. of DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems (Hybrid Systems III: Verification and Control), New Brunswick, New Jersey, October 1995, Lect. Notes in Comput. Sci, vol. 1066*, pages 496–510. Springer-Verlag, 1996.

[MP92] Z. Manna and A. Pnueli. *Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. IEEE 18th Annu. Symp. on Found. Comput. Sci.*, pages 46–57, New York, 1977. IEEE.

[PVS] PVS. WWW site of PVS papers. http://www.csl.sri.com/sri-csl-fm.html.

[Rab97] A. Rabinovich. Decidability in monadic logic of order over finitely variable signals. Manuscript, 15 p., 1997.

[Rab98a] A. Rabinovich. Expressive completeness of duration calculus. Manuscript, 33 p., 1998.

[Rab98b] A. Rabinovich. On the decidability of continuous time specification formalisms. *J. of Logic and Computation*, 8(5):669–678, 1998.

[Sli91] A. Slissenko. On measures of information quality of knowledge processing systems. *Information Sciences: An International Journal*, 57–58:389–402, 1991.

[Sli99] A. Slissenko. Minimizing entropy of knowledge representaion. In *Proc. of the 2nd International Conf. on Computer Science and Information Technologies, August 17–22, 1999, Yerevan, Armenia*, pages 2–6. National Academy of Sciences of Armenia, 1999.

[Som92] I. Sommerville. *Software Engineering.* Addison-Wesley, 4th edition, 1992.

[Tar96] V. V. Tarasov. *Inference search control in expert systems based on explicit meta-rules of inference search.* PhD thesis, St. Petersgurg Institute for Informatics and automation, Russian Academy of Sciences, 1996. (Russian.).

[Tra98] B. Trakhtenbrot. Automata and hybrid systems. Lecture Notes 153, Uppsala University, Computing Science Department, 1998. Edited by F. Moller and B. Trakhtenbrot.

[Var96] M. Vardi. An automata-theoretic approach to linear temporal logic. In F. Moller and G. Birtwistle, editors, *Logic for Concurrency. Structure versus Automata*, pages 238–266. Springer-Verlag, 1996. Series: "Lecture notes in Computer Science (Tutorial)", Vol. 1043.