On Refinements of Real-Time Distributed Abstract State Machines

Cohen, J. and Slissenko, A.

Technical Report TR-2002-03

Laboratory of Algorithms, Complexity and Logic
University of Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
F-94010 CRÊTEIL Cedex – FRANCE
Tel: +33 (0)1 45 17 16 47
Fax: +33 (0)1 45 17 66 01
On Refinements
of Real-Time Distributed Abstract State Machines

J. Cohen\(^1\) A. Slissenko\(^1\,\)\(^2\)
E-mail: \{j.cohen,slissenko\}@univ-paris12.fr

Laboratory for Algorithmics, Complexity and Logic,
University Paris-12, France

Abstract. We introduce notions of refinement of distributed real-time algorithms. The goal is to find realistic models of refinement that guarantee preservation of requirements properties under refinement process. In this paper we study this question from semantical point of view - an algorithm is considered as a set of runs of an abstract state machine. We try to outline possible approaches to defining refinements of distributed real-time algorithms. Two notions of refinement are considered the first one, stable refinement, can capture some 'light' real-time constraints and the second one, shifted refinement, permits to capture 'hard' real-time constraints. In conclusion we discuss open questions.

1 Introduction

Program development usually starts at a high level of abstraction of requirements and of the first program specification. This high level of abstraction, in particular, simplifies proving requirements properties, i.e. simplifies the verification. Though the practical development process is not 'monotone', its resulting phase gives a hierarchy of more and more concrete programs. This process of concretization is described by words like "refinement", "implementation" and maybe some others. Not entering into terminology discussions we consider only simple type of what can be called "refinement". This process presumes that we go from an initial algorithm to a refined one by only adding functions to the vocabulary. So we suppose that while refining the sorts and the types of functions of the initial, non refined program remain unchanged. It is a simple case — we do not consider concretizations where, say, "real" in mathematical sense becomes "float" of a certain kind.

We wish to preserve the proven requirements properties when making refinements. It is a usual demand. Refinements of sequential algorithms without time were studied in numerous papers, for example in [Wir90], [Ost99], [AL88], [GHLZ00] — we commented their relation to our problem in [CS00]. On the whole refinements of real-time algorithms were not yet seriously studied.

\(^1\) Address: Dept. of Informatics, University Paris 12, 61 Av. du Gén. de Gaulle, 94010, Créteil, France.
\(^2\) Member of St-Petersburg Institute for Informatics and Automation, Russian Academy of Sciences.
We consider a simple type of refinements where observables are inputs and outputs, sorts and types do not change in refinement process and runs, as well as requirements properties, are considered semantically, that is as sets of models. Implicitly we use the notion of Gurevich Abstract State Machine (ASM) [Gur95,Gu00] – our runs will be evolving algebras (our motivations and examples are also ASM based). We concentrate on problems of real-time, in particular for reactive systems. This case is difficult as the runs are infinite in the general case. Taking into consideration that reactive systems are usually controllers whose initial specification is done in terms of continuous time we treat time as non-negative reals. Below we make some precisions of this setting.

An algorithm will be in our context identified with its set of runs. An algorithm in ‘syntactical’ sense deals with functions without time argument. It can see time as the value of some input function $CT$ (Current Time). But to describe its runs we add to all dynamic functions of the algorithm a time argument to describe our vision of evolution of their values. So algorithm in ‘semantical’ sense is a set of interpretations of some vocabulary containing also timed functions. We will assume that syntactical algorithms have dynamic functions of the type $f : \mathcal{X} \rightarrow \mathcal{Z}$, where $\mathcal{X}$ is a direct product of finite (maybe abstract) sorts. The timed image of $f$ is $f^\tau : T \times \mathcal{X} \rightarrow \mathcal{Z}$, and this image will be in vocabularies we consider to speak about algorithms.

Let $f^\tau : T \times \mathcal{X}^\tau \rightarrow \mathcal{Z}^\tau$ be an interpretation of $f^\tau$, where $\mathcal{X}^\tau$ and $\mathcal{Z}^\tau$ are interpretations of, respectively, $\mathcal{X}$ and $\mathcal{Z}$. If we fix $X \in \mathcal{X}^\tau$ we get a function $f^\tau_X = \lambda \tau. f^\tau(\tau, X)$ of one time argument that will be called identifier. It represents a nullary dynamic function of syntactical algorithm. Each identifier is piecewise finitely defined as an algorithm can change its values only in some discrete time moments – for simplicity we assume that all dynamic functions, in particular inputs, are piecewise constant (more general cases are described in [BS00,Bs99]).

In each run an identifier may change its value at or after certain time moments. Each value change is defined by some term from a fixed set – one may think about an assignment in some program. The only property of value change essential to us is that we can speak about the same value changes in two runs – not to go into formal discourse imagine that we speak about the same assignment, the “same” from all points of view. One more point is to be clarified: whether a value change takes place at a moment $t$, i.e. for all $\tau \in [t, t')$, where $t'$ stands for the moment of the next value change, or after $t$, i.e. for all $\tau \in (t, t']$. Physically we cannot distinguish these two possibilities, but for formal treatment they are to be fixed. To simplify intuitive understanding we assume that the first type of value change takes place (and inputs and outputs are of appropriate types), though the second type is in some way better – see [BS00,Sb01] for details. So we assume that intervals where values of our functions do not change are of the form $[a, b)$, $a < b$, and there are no Zeno effects in runs.

In a run we have value changes that are attached to some time moments. Taken together with a time moment a value change will be called an event.

We will use the notation $\text{Runs}(\mathcal{A}, \mathcal{I})$ to denote the set of runs of an ‘algorithm’ $\mathcal{A}$ for input $\mathcal{I}$; however, in this notation $\mathcal{A}$ is simply a symbol that in fact makes
a reference to a set of interpretations. Below we will denote inputs by $I$ and outputs by $O$ and, as it was mentioned above, that will be our observables. We could consider more general situation but will not do it as our main concern is time.

It is intuitively clear that relation between time moments of corresponding event is the main question to study in refinements of timed algorithms. We touched this question in [CS00] where a rather tough type of refinements was studied – refinements where input and output events are exactly the same in an initial and in a refined algorithms. We constructed a refinement of interprocess communications of a real-time version of Lamport's Bakery and gave some arguments that it cannot be a straightforward syntactical substitution (though there is no proof of such impossibility). This notion of refinement does not work for hard real-time when a (more realistic) refinement is slower than a (more abstract) initial algorithm. In this paper we introduce two more general notions – one (stable refinement) for 'light' real-time where time is mainly viewed as an ordered (separable) topological space and delays between actions can be diminished when refining an initial algorithm, and the other one (shifted refinement) for 'hard' real-time with metric time constraints and delays that augment when refining. To preserve the requirements we, clearly, must demand some robustness of algorithms and of requirements, and, maybe, even slightly weaken the requirements.

Another way to illustrate our motivation is given by the two scenarios below.

Scenario 1: From the very beginning we consider algorithms with some delay between actions, so while refining our algorithms we demand that, though the number of actions to accomplish augments, these actions must respect the constraints on essential global delays. In other words, we assume that our more concrete algorithm has more fast 'instructions' that its more abstract ancestor.

Scenario 2: For the initial high level abstraction we suppose that there are no delays between actions or the delays are 'very small'. In this situation we cannot guarantee the initial time constraints when passing to a more realistic algorithm with 'more delayed' actions. So we are to modify some metric time constraints in the requirements. The latter does not means that we cannot meet the required properties. It means that we represent the time constraints as some parameters, and we impose additional constraints on time in terms of new parameters that describe the functioning of our more refined algorithm. And while refining we recalculate the time constraints in requirements in terms of all the parameters that have appeared.

For each of two notion of refinement we give conditions under which the requirements properties, maybe modified, are preserved. The technical problem is to find reasonable notions – proving theorems about preservation of requirements properties is not hard. In conclusion we mention open questions and make a
remark on compositionality issues that become difficult to insure for real-time refinements if the corresponding time moments can drift.

Notational conventions.
Let $A_0$ and $A_1$ be two algorithms over vocabularies $V_0$ and $V_1$ respectively. We assume that input functions $I$ and output functions $O$ of these vocabularies are the same, and we denote the vocabulary of these common input and output functions by $V IO$. Below $A_1$ will be a refinement of $A_0$.

Concerning the requirements property $\Phi$, we assume that $\Phi$ is over the vocabulary $V IO$ of inputs/outputs only. In practice this is not always the case. The property $\Phi$ is often an implication from a property $Env$ describing the environment of functioning of our algorithm to properties of functioning like safety and liveness. Usually $Env$, in particular, imposes constraints on admissible runs. We tacitly assume that $Env$ is taken into consideration in the definition of runs (usually it is quite feasible) and thus, $\Phi$ represents only properties of functioning. Without taking into consideration this remark our treatment of shifted refinements may seem not well justified from practical point of view.

Denote by $Runs[A, I]$ the set of all runs of $A$ for a given input $I$. This set is presumed to be non-empty. It can contain many elements if we admit some non-deterministic delays between actions. For real-time situations these delays are presumed to be bounded from above by constants (that can be different for different types of delays).

For any notion of refinement the following property, that will be referred to as Requirements Preservation Property (RPP), is highly desirable in the context of verification:

$$\forall I \left( \left( Runs[A_0, I] \models \Phi \right) \iff \left( Runs[A_1, I] \models \Phi \right) \right), \quad \text{(RPP)}$$

where the notation of the form $SetOfInterpretations \models Formula$ means that that any interpretation of $SetOfInterpretations$ is a model of $Formula$. Why we do not demand in (RPP) just implication "$\Rightarrow$"? The real software development process is not a one-directional descent from high-level specifications to lower level ones; it involves moving up and down, and making more precise programs as well as requirements.

2 Stable Refinements

To model 'light' real-time (the first scenario) we introduce a notion of stable refinement.

First we define a notion of separating sequence that will describe neighborhoods of sequences of events of a run. We consider sequences of time intervals whose type corresponds to the type of value changes (in our case of the form $[a, b)$, $a < b$) that are strictly monotone in the following usual sense. If $\sigma$ is such a sequence then it may be finite or infinite and for this reason its domain $dom(\sigma)$ that is a prefix of the set $\mathbb{N}$ of natural numbers, will appear in the definitions. We assume that for any $i, (i + 1) \in dom(\sigma)$ the right end $\sigma(i)^{r}$ of interval $\sigma(i)$ is strictly smaller than the left end $\sigma(i + 1)^{l}$ of the next one: $\sigma(i)^{r} < \sigma(i + 1)^{l}$. 
Given an arbitrary input $I$ and a run $\rho$ of an algorithm $A$ for this input, a sequence $\sigma$ of time intervals is \textit{separating} for $\rho$ if any of its intervals contains only one moment of change of values of identifiers of $A$ and if no event takes place outside intervals of $\sigma$. Remark that an interval of a separating sequence may have value changes of several identifiers, but all of them take place at the same time moment. The set of identifiers that change their values in $\sigma(i)$ will be denoted by $\Theta[\rho, \sigma](i)$, and the time moment where the identifiers from $\Theta[\rho, \sigma](i)$ change their values in $\sigma(i)$ will be denoted by $\tau[\rho, \sigma][i]$. 

Consider two pairs $(\rho_0, \sigma_0)$ and $(\rho_1, \sigma_1)$, where $\sigma_0$ and $\sigma_1$ are separating sequences for runs $\rho_0$ and $\rho_1$ of algorithms $A_0$ and $A_1$ respectively. Denote for brevity $\Theta[\rho_k, \sigma_k][i]$ and $\tau[\rho_k, \sigma_k][i]$ by $\Theta_k(i)$ and $\tau_k(i)$ respectively, $k = 0, 1$.

Now we wish to describe a property that says that two runs are ‘close’. To do it we supply each run with a separating sequence and say that, first, these sequences define, in a way, the same closeness, and, second, there is a ‘quasbijection’ between the changes of values taking place in both sequences modulo gluing or ungluing of consecutive events within intervals of separating sequences. Each interval gives a precision with which we treat a run – we can displace the moment of value change inside the interval and have again a run that will be considered as close to the initial one.

More precisely, we say that $(\rho_0, \sigma_0)$ is a \textit{modification} of $(\rho_1, \sigma_1)$ if there exists a partition $\pi_k$ of $\text{dom}(\sigma_k)$, $k = 0, 1$, such that

- $\#\text{dom}(\pi_0) = \#\text{dom}(\pi_1)$;
- each $\pi_k(j)$ is a non empty set of consecutive integers from $\text{dom}(\sigma_k)$, $j \in \text{dom}(\pi_k)$;
- if each of $\sigma_0(j)$ and $\sigma_1(j)$ contains one element, say respectively $i_0$ and $i_1$, then $\sigma_0(i_0) = \sigma_1(i_1)$, $\Theta_0(i_0) = \Theta_0(i_1)$ and the values changes of $\Theta_0(i_0)$ at $\tau_0(i_0)$ are identical to value changes of the respective identifiers from $\Theta_1(i_1)$ at $\tau_0(i_1)$.
- if $\pi_k(j)$ contains more than one element then $\pi_{1-k}(j)$ is a singleton (several elements in $\pi_k(j)$ describe a gluing of the events in intervals $\sigma_k(i)$, $i \in \pi_k(j)$, of run $\rho_k$ onto one time moment in $\sigma_{1-k}(\pi_{1-k}(j))$; in other direction, from $\rho_{1-k}$ to $\pi_k(j)$ this will be ungluing – see the item below);
- if $\pi_k(j)$ contains more than one element, say $\pi_k(j) = \{m, m + 1, \ldots, m + s_j\}$, $s_j \geq 1$, then the intervals $\sigma_k(i)$, $i \in \pi_k(j)$ are inside $\sigma_{1-k}(\nu)$, where $\nu$ denotes the only element of $\pi_{1-k}(j)$, but the ends are the same: $\sigma_k(m)^{[i]} = \sigma_{1-k}(\nu)^{[i]}$ and $\sigma_k(m + s_j)^{[i]} = \sigma_{1-k}(\nu)^{[i]}$, identifiers $\Theta_{1-k}(\nu)$ are partitioned between $\sigma_k(i)$, i.e. $\Theta_{1-k}(\nu) = \bigcup_{i \in \pi_k(j)} \Theta_k(i)$ as well as their moments of value change, and the changes of respective identifiers are the same in $\sigma_{1-k}(\nu)$ at $\tau_{1-k}(\nu)$ and at the respective $\sigma_k(i)$ at $\tau_k(i)$. (See Figure 1.)

Below for the \textit{projection} of an interpretation (in particular, of a run) $\rho$ over an alphabet $V$ onto its sub-alphabet $W$ we use the notation $\text{proj}_W(\rho)$: the alphabet $V$ will be clear from the context; the projection, as usually, means that we delete from $\rho$ everything that is related to $V \setminus W$.

An algorithm $A_1$ is a \textit{stable refinement} of an algorithm $A_0$ (remind that for their alphabets we have $V_0 \subseteq V_1$) if
(A) for every run \( \rho_0 \) of \( A_0 \) there exist a separating sequence \( \sigma_0 \) for \( \rho_0 \), a run \( \rho_1 \) of \( A_1 \) and a separating sequence \( \sigma_1 \) for \( \text{proj}_{V_0} (\rho_1) \) such that \( (\text{proj}_{V_0} (\rho_1), \sigma_1) \) is a modification of \( (\rho_0, \sigma_0) \) (remind that by definition of modification \( \text{proj}_{V_0} (\rho_1) \) is a run of \( A_0 \));

(B) for every run \( \rho_1 \) of \( A_1 \) there exist a separating sequence \( \sigma_1 \) for \( \text{proj}_{V_0} (\rho_1) \), a run \( \rho_0 \) of \( A_0 \) and a separating sequence \( \sigma_0 \) such that \( (\rho_0, \sigma_0) \) is a modification of \( (\text{proj}_{V_0} (\rho_1), \sigma_1) \) (again remark that \( \text{proj}_{V_0} (\rho_1) \) is a run of \( A_0 \)).

A property \( \Phi \) is robust for a class of algorithms if for any run \( \rho \) of any algorithm of this class, there is a separating sequence \( \sigma \) such that if \( \rho \models \Phi \) then for any modification \( (\rho', \sigma') \) of \( (\rho, \sigma) \) we have \( \rho' \models \Phi \). (Remind that we consider properties over the alphabet of inputs/outputs that is common for all algorithms under consideration).

**Theorem 1** If \( A_1 \) is a refinement of \( A_0 \) and \( \Phi \) is a robust property for \( A_0 \) then \((RPP)\) holds: \( \forall I \left( (\text{Runs}[A_0, I] \models \Phi) \iff (\text{Runs}[A_1, I] \models \Phi) \right) \).

**Proof.** Let \( I \) be an input of algorithms \( A_0 \) and \( A_1 \).

\( \Rightarrow \). Let \( \rho_1 \) be a run of \( A_1 \). By definition of stable refinement there exist a separating sequence \( \sigma_1 \) for \( \text{proj}_{V_0} (\rho_1) \), a run \( \rho_0 \) of \( A_0 \) and a separating sequence \( \sigma_0 \) for \( \rho_0 \) such that \( (\rho_0, \sigma_0) \) is a modification of \( (\text{proj}_{V_0} (\rho_1), \sigma_1) \). Therefore \( \text{proj}_{V_0} (\rho_1) \) is a run of \( A_0 \), and it verifies \( \Phi \) by assumption: \( \text{proj}_{V_0} (\rho_1) \models \Phi \). Remains to notice that this implies \( \rho_1 \models \Phi \) as \( \Phi \) is over alphabet of inputs/outputs \( V_{\text{ZO}} \) and \( V_{\text{ZO}} \subseteq V_0 \subseteq V_1 \).

\( \Leftarrow \). Let \( \rho_0 \) be a run of \( A_0 \). There exist a separating sequence \( \sigma_0 \) for \( \rho_0 \), a run \( \rho_1 \) of \( A_1 \) and a separating sequence \( \sigma_1 \) for \( \text{proj}_{V_0} (\rho_1) \) such that \( (\rho_0, \sigma_0) \) is a modification of \( (\text{proj}_{V_0} (\rho_1), \sigma_1) \). The definition of modification being symmetric, \( (\rho_0, \sigma_0) \) is also a modification of \( (\text{proj}_{V_0} (\rho_1), \sigma_1) \). By assumption \( \rho_1 \models \Phi \). As \( \Phi \) is over alphabet of inputs/outputs \( V_{\text{ZO}} \subseteq V_0 \subseteq V_1 \) then \( \text{proj}_{V_0} (\rho_1) \models \Phi \). But \( \text{proj}_{V_0} (\rho_1) \) and \( \rho_0 \) are runs of \( A_0 \), and \( \Phi \) is robust for \( A_0 \). Hence \( \rho_0 \models \Phi \).

To illustrate the notion of stable refinement consider Lampert’s Bakery [Lam74] as shown on Fig. 2 for two semantics: one with unbounded positive delays between actions, and another one with bounded (from above by a fixed constant) positive delays between actions (for a rigorous treatment of the semantics see [CS00], however the notations are self-explanatory). Remind the idea of this al-
Algorithm. A process \( p \) shows its desire to enter its Critical Section (CS) by setting \( \text{number}_p = 1 \) (one may assume that the delay before the beginning of line 1 is always unbounded). In line 3 it calculates its ticket and in line 4 assigns it to \( \text{number}_p \) to make it visible to other processes. Then in line 5 it waits until its ticket becomes a minimal one and after that it enters its CS in line 6 and leaves it in line 7. To show to other processes that it has left its CS process \( p \) sets its \( \text{number}_p \) to 0 in line 8. The order \((a,b) < (c,d)\) means that \( a < c \), or \( a = c \) and \( b < d \).

Two usual requirements demand that

(Safety) \( \Phi_S : \forall p \forall t \ ( p \neq q \rightarrow \neg (\text{CS}_p(t) \land \text{CS}_q(t)) ) \)

- no no two different processes are in the CS at the same time;

(Liveness for unbounded delays)
\( \Phi_L : \forall p \forall t \ ( \text{number}_p(t) > 0 \rightarrow \exists t' \ ( t < t' \land \text{CS}_p(t')) ) \)

- each process wishing to enter the Critical Section eventually enters there;

(LivenessB for bounded delays)
\( \Phi_{LB} : \forall p \forall t \ ( \text{number}_p(t) > 0 \rightarrow \exists t' \ ( t < t' < t + \gamma \land \text{CS}_p(t')) ) \)

(where \( \gamma \) is some positive constant that depends on the number of processes and on the upper bounds on delays)
- each process wishing to enter the CS enters there after waiting not more than γ time.

One can prove, following [Lam74], that algorithm BakeryAsyn verifies the requirements. This proof permits to establish robustness of the requirements for BakeryAsyn. Take any run ρ and consider (Safety) - formula ΦS. By the definition of runs the events of entering CS can be separated, i.e. there exists a separating sequence σ for ρ where each interval contains at most one entering CS event and no leaving CS events, or at most one leaving CS event and no entering CS events (remark that there is a delay between each of lines 6-8). Any modification of (ρ, σ) (which is again a run) will be safe as no modification of ρ can execute entering and leaving CS for one process p in the same time, consequently entering CS for processes p and q cannot be simultaneously executed in any modification of ρ.

Now let us consider (Liveness). It is clearly robust for unbounded delays due to similar arguments as for safety. For bounded delay we can argue as follows. Take any run ρ. It has a separating sequence, say σ, by the definition of runs. Now take any process p and consider two time moments, first, t₀ where ρ demanded entering its CS by setting numberρ to 1 and, second, the nearest next moment t₁ when ρ entered its CS. Each of these moments is in some interval of σ, suppose tₖ ∈ σ(iₖ), k = 0, 1. We know that t₀ < t₁ < t₀ + γ. Set ε = \( \frac{γ - t₀}{3} \). Clearly, ε > 0. Construct a sequence σ' by replacing (for all p and all pairs (t₀, t₁)) σ(i₀) by an interval where its left end is (maybe) shifted to the right to t₀ - ε and by replacing σ(i₁) by an interval where its right end is (maybe) shifted to the left by t₁ + ε: σ'(i₀) = max{σ(i₀)(l), t₀ - ε}, σ'(i₁) = min{σ(i₁)(l), t₁ + ε}. Now if we modify (ρ, σ') we will still have (LivenessB). Thus both ΦL and ΦLB are robust for BakeryAsyn.

In [CS00] we gave a refinement of BakeryAsyn that is a stable refinement; we refined interprocess communications as message exchange via sufficiently large number of channels that permit to send and receive different types of messages to/from different processes independently. We also outlined a proof of a property that implies that this refinement is stable, though this term was not used in that paper.

3 Shifted Refinements

For the 2nd scenario we are to treat time shifts. We start with an example to illustrate the situation. Consider the controller on Fig. 3 for the Generalized Railroad Crossing Problem (we take the controller from [BS00] that is similar to that of [GH96] but with better liveness).

This is a simple Gurevich ASM that executes all its If-Then operators simultaneously and whose actions are instantaneous. Here the following notations are used. There is a finite set of tracks denoted Tracks, letter x being used as variable for tracks. Input function (predicate) Cmg(x) says there is a train on track x, and Emp(x) is its negation. Notation WT (Wait Time) stands for \( d_{\text{min}} - d_{\text{start}} \), where \( d_{\text{min}} \) is a lower bound on the time of arrival of a detected
Repeat
ForAll x ∈ Tracks
InParallelDo
  If Cm(x) and DL(x) = ∞ Then DL(x) := CT + WT EndIf
  If Emp(x) and DL(x) < ∞ Then DL(x) := ∞ EndIf
  If DirOp and ¬SfOp Then DirCl := true EndIf
  If DirCl and SfOp Then DirOp := true EndIf
EndInParallelDo
EndForAll
EndRepeat

Fig. 3. Railroad Crossing Controller.

train in the crossing, and dcl is an upper bound on the time to close the
gate if the signal to close that is denoted by DirCl is maintained. This abstract
constant WT give an interval of time during which a detected train remains far
enough from the crossing so one can not to close the gate. The signal opposite
to DirCl, i.e. a signal to open the gate, is DirOp (the negation of DirCl). It is
clear that DirCl is the output of the controller to construct. Internal function
DL expresses a deadline for not to close the gate from the point of view of the
situation on one track. A global condition saying when it is safe to open the gate
is given by:

\[ SfOp = \forall x (Emp(x) \lor CT < DL(x)) \]

where CT denotes current time – it is an external function. To finish the de-
scription we state the initial condition:

\[ Init = \forall x (DL(x) = ∞ \land DirOp) \]

The goal of the controller is to close the gate to insure safety and to open
the gate to insure liveness (even an optimal one):
(Safety): \( \forall t (InCr^o(t) \rightarrow GtClsd^o(t)) \).
(When a train is in the crossing, the gate is closed).
(Liveness): \( \forall t (SfOp^o(t) \rightarrow DirOp^o(t)) \).
(If the zone of control is safe to open at time \( t \) then the control signal must be
to open the gate).

Here we use auxiliary functions:

- \( InCr^o \) (\( ^o \) indicates that the function depends on time; it is important not to
  mix \( DirOp \) of the controller and its timed image \( DirOp^o \) used in requirements)
  that says that there is a train in the crossing. This function can be expressed in
  terms of inputs/outputs, i.e. in terms of \( Cm^o \) and \( DirOp^o \).
- \( GtClsd^o \) says that the gate is closed. This function can be also expressed
  in terms of inputs/outputs.
• $SfOp^*$ is a ‘logic’ analogue of safe to open condition that is slightly complicated (we explain it below):

\[ \forall x \left[ \text{Emp}^x(t, x) \land \forall \tau \leq t \left( \forall \tau' \in [\tau, t) \text{Cmg}^x(\tau', x) \rightarrow t < \tau + WT \right) \right]. \]

In terms of inputs/outputs (Safety) says that if at least on one track a train approaches the crossing at a distance that can be covered during time $d_{\text{close}}$ the controller must send a signal to close the gate. So this property depends on the distance between moments of arrival of trains and the corresponding moment to close the gate.

As for (Liveness) it simply says that if all the trains (if any) are far enough from the crossing (again in terms of $d_{\text{close}}$) either to open the gate or not to close it. Again we have a similar dependence on the distance between moments of arrival of trains and the corresponding moment to close the gate, as just above for (Safety).

We omit the description of environment that can be found in [BS00] as our discourse is illustrative. One can prove that the controller verifies the requirements (for a formal PVS checked proof see [BCS00]).

Now suppose that we wish to implement this controller as a one processor C program in a straightforward way. It is reasonable because the train speed is much smaller than the speed of computer. So we execute the guard checks and assignments as a round-robin loop. On a real computer the executions are not instantaneous. For a simple program as the just mentioned one we can calculate practically exactly all the delays involved. So the output of the control signal will be computed with a delay, say, $\delta > 0$ (to simplify the presentation we suppose that communications with sensors and effectors are instantaneous). We assume that $\delta$ is much smaller than constants that were mentioned above, namely $d_{\text{close}}$ and $WT$.

In this setting we cannot use the guards of the controller as they are as by the moment we calculate the value of control signal ($\text{DirOpr}$ or $\text{DirCl}$) we will be $\delta$ time late. So to close the gate and to insure safety we are to shift this moment by $\delta$ to a more early time. But (Safety) will not see this shift – the necessary distance between train appearance and $\text{DirCl}$ will be respected. As for liveness, we cannot do a similar thing – after the moment when one can send a signal to open the gate and real calculation of this value of signal there is a delay $\delta$, so we are to weaken the requirement itself and impose a liveness that demands a smaller accessibility to the crossing, i.e. a smaller time when the gate is open.

Remark that for the initial algorithm as well as for the refined algorithm the input events are exactly the same; the algorithms can play only with outputs, in our example with $\text{DirCl}$.

Now we will try to arrange the considerations outlined above into a precise framework.

Consider an algorithm $A$ over an alphabet $V$ with inputs $I$ and outputs $O$. Let a run $\rho$ of $A$ be given, and let $\theta$ be the interpretation of an identifier of $A$ in this run. Remind that identifiers are piecewise constant. Let $U$ be a subset of the set of all possible values of $\theta$. This set determines a sequence $\zeta$ of time
intervals on which the value of $\theta$ belongs to $U$. For a real $\delta > 0$ denote by $\theta[\delta, U]$ a function of time that is obtained from $\theta$ by shrinking all $\zeta(i)$ (excluding the first one if $\zeta(0)[i] = 0$) from the left to the right by $\delta$. More formally, let $\alpha$ be an interval adjacent to $\zeta(i)$ from the left, i.e. such that $\alpha^{(r)} = \zeta(i)^{[l]}$, where the value of $\theta$, that we denote by $\theta(\alpha)$, is not in $U$. Then we set $\theta[\delta, U]^{(r)} = \theta(\alpha)$ for $r \in [\zeta(i)^{[l]}, \min(\zeta(i)^{[l]} + \delta, \zeta(i)^{[l]}))$, and do it for all $i$ modulo the mentioned exception. For all other time moments the value of $\theta[\delta, U]$ coincides with the value of $\theta$.

This $\theta[\delta, U]$ will be called left $(\delta, U)$-shrinking of $\theta$. In a similar way one can define right shrinking or two-sided shrinking. We will deal only with left shrinkings.

We will consider only non-degenerating shrinkings: we suppose that $\delta$ is smaller than the length of intervals being shrunk, i.e. in terms of notations in the definition of shrinking we assume that $\zeta(i)^{[l]} + \delta < \zeta(i)^{[r]}$, and hence $\min(\zeta(i)^{[l]} + \delta, \zeta(i)^{[r]} = \zeta(i)^{[l]} + \delta$. With this assumption, given a shrinking we can uniquely restore its pre-image.

As an example consider $DirOp$ of the controller and $U = \{true\}$. Suppose that $\delta$ is exactly the delay that is implied by the implementation of the controller as a sequential program. If we take a run of the controller and make $(\delta, U)$-shrinking of $DirOp$, the obtained interpretation will satisfy (Safety) - it is clear intuitively, as the moments of sending signal to close the gate are the same, and moments to open the gate are delayed.

Now suppose that $\mathcal{F}$ is a subset of the vocabulary of output functions. An interpretation of inputs/outputs being fixed, this set determines a set of identifiers: a function $f : T \times X \to Z$ gives identifiers $f_X$ for each value $X$ of the interpretation $X^*$. Fixing for each $f \in \mathcal{F}$ a set of values $U_f$ we get a set $\mathcal{U} = \{U_f\}_{f \in \mathcal{F}}$ of sets of values for identifiers of the form $f_X$. Denote $\Delta = (\delta, \mathcal{U}, \mathcal{F})$. We can define $\Delta$-shrinking of an interpretation as $(\delta, \mathcal{U})$-shrinkings defined for each identifier $f_X$ of this interpretation. One can easily imagine more general notions.

We say that a property $\Phi$ is robust for left $\Delta$-shrinkings of interpretations of a given class if for any model $\mathcal{M}$ of $\Phi$ of this class its left $\Delta$-shrinking is again a model of $\Phi$.

If to take (Safety) as $\Phi$, function $DirOp$ as $\mathcal{F}$ and $U = \{true\}$ as $\mathcal{U}$ we can prove that (Safety) is robust for interpretations satisfying the environment properties. Intuitive argument were given just above: (Safety) says that if at least one train is close enough to the crossing the control signal must be $DirCl$. If we extend $DirCl$ to the right (that is shrink $DirOp$ from the left) the property remains true (to prove it rigorously one needs a complete description of the requirements specifications).

For (Liveness) we have no robustness like for (Safety). To understand what we can do to insure a reasonable analogue of (Liveness) we come to defining our shifted refinements.

Let $\Delta = (\delta, \mathcal{U}, \mathcal{F})$ be fixed with the meaning defined above; remark that types of $\mathcal{U}$ and $\mathcal{F}$ depend only on the alphabet of outputs that is common for all the algorithms under consideration.
Not to be too general, as a notion of shifted refinement (the term does seem to be good, this refinement has a flavor of implementation) we introduce the following one.

An algorithm \( A_1 \) over alphabet \( V_1 \) is a left \( \Delta \)-shrinking of an algorithm \( A_0 \) over alphabet \( V_0 \) if

(a) for every run \( \rho_0 \) of \( A_0 \) there exists a run \( \rho_1 \) of \( A_1 \) such that \( \text{proj}_{V_0}(\rho_1) \) is a left \( \Delta \)-shrinking of \( \rho_0 \);

(b) for every run \( \rho_1 \) of \( A_1 \) there exists a run \( \rho_0 \) of \( A_0 \) such that \( \text{proj}_{V_0}(\rho_1) \) is a left \( \Delta \)-shrinking of \( \rho_0 \).

Using the notion of shrinking of algorithms we can summarize our previous discussions on the Railroad Crossing (Safety) as

**Theorem 2** If \( \Phi \) is robust for left \( \Delta \)-shrinkings of runs of \( A_0 \) then for any left \( \Delta \)-shrinking \( A_1 \) of \( A_0 \):

\[
\forall I \left( (\text{Runs}[A_0, I] \models \Phi) \Rightarrow (\text{Runs}[A_1, I] \models \Phi) \right).
\]

**Proof.** Let \( I \) be an input of algorithms \( A_0 \) and \( A_1 \).

Let \( \rho_1 \) be a run of \( A_1 \). Then there exists a run \( \rho_0 \) of \( A_0 \) such that \( \text{proj}_{V_0}(\rho_1) \) is a left \( \Delta \)-shrinking of \( \rho_0 \). By assumption \( \rho_0 \) is a model of \( \Phi \). Any its left \( \Delta \)-shrinking is also a model of \( \Phi \) as \( \Phi \) is robust. In particular, the left \( \Delta \)-shrinking \( \text{proj}_{V_0}(\rho_1) \) of \( \rho_0 \) is a model of \( \Phi \), and taking into consideration that \( \Phi \) is over \( V_0 \ell \), run \( \rho_1 \) is a model of \( \Phi \).

Simple, we cannot have (RPP) equivalence in this situation without additional constraints on \( \Phi \).

For (Liveness) we can obtain some analogue of (RPP) by modifying the property for ‘shrunk’ algorithms.

Denote by \( \Phi_{\Delta} \) the set of left \( \Delta \)-shrinkings of models of \( \Phi \). It may be expressible as a formula of the same logic language as \( \Phi \) or not — we never mentioned any syntax for our properties in spite of practical importance of this question. This property \( \Phi_{\Delta} \) may be of interest or not — it is for us to decide. However, we have the following useful observation:

**Theorem 3** If \( A_1 \) is a left \( \Delta \)-shrinking of \( A_0 \) then

\[
\forall I \left( (\text{Runs}[A_0, I] \models \Phi) \iff (\text{Runs}[A_1, I] \models \Phi_{\Delta}) \right).
\]

**Proof.** Let \( I \) be an input of algorithms \( A_0 \) and \( A_1 \).

\( \Rightarrow \). Let \( \rho_1 \) be a run of \( A_1 \). Its projection onto \( V_0 \) is a left \( \Delta \)-shrinking of some run \( \rho_0 \) of \( A_0 \) as \( A_1 \) is a left \( \Delta \)-shrinking of \( A_0 \). By assumption \( \rho_0 \models \Phi \). By definition of \( \Phi_{\Delta} \) any left \( \Delta \)-shrinking of \( \rho_0 \) is a model of \( \Phi_{\Delta} \), in particular, \( \text{proj}_{V_0}(\rho_1) \) is a model. Thus \( \rho_1 \models \Phi_{\Delta} \).

\( \Leftarrow \). Let \( \rho_0 \) be a run of \( A_0 \). As \( A_1 \) is a left \( \Delta \)-shrinking of \( A_0 \) there exists a run \( \rho_1 \) of \( A_1 \) such that \( \text{proj}_{V_0}(\rho_1) \) is a left \( \Delta \)-shrinking of \( \rho_0 \). By assumption \( \text{proj}_{V_0}(\rho_1) \) is a model of \( \Phi_{\Delta} \). But any such model is, by definition of \( \Phi_{\Delta} \), a left
\(\Delta\)-shrinking of a model of \(\Phi\). But this left \(\Delta\)-shrinking is defined uniquely, so it is a left \(\Delta\)-shrinking of \(\rho_0\). Thus \(\rho_0\) is a model of \(\Phi\).

\section*{Conclusion}

This paper is an attempt to advance a theory of refinements of real-time algorithms. The main obstacle for us was to find some systems of notions to start a discussion. It is quite normal that proofs of preservation of requirements properties are simple — the major problems are in proving that one program is a refinement of another. To facilitate this task syntactical transformations giving refinements are to be studied. However, in practical program development there are passages from program to a more concrete one that are hard to analyze — for example, results of optimizing compilers. So verification that one program is a refinement of another is a question of interest (the problem is clearly undecidable in general). Compositional properties constitute a useful tool to insure being a refinement. For our stable refinement some such properties can be proven. For shifted refinement exact compositionality is impossible but one may have an 'approximate' one. Taking into consideration that real-time algorithms are usually of simple structure even such approximate compositionality may be useful.

\section*{References}


