Université PARIS 12 - Val de MARNE
Faculté des INFORMATIQUE SCIENCES et TECHNOLOGIE

Laboratoire d'Algorithmique,
Complexité et Logique (LACL)
Université Paris 12 — Val de Marne
Faculté des Sciences et Technologie
61, Av. du Gén. de Gaulle
94010 CRETEIL cedex
FRANCE
Tel: (33 1) 45 17 16 47, Fax: (33 1) 45 17 16 49

A Predicate Logic Framework for Mechanical Verification of Real-Time Abstract State Machines: A Case Study with PVS.

Danièle Beauquier **Tristan Crolard** **Anatol Slissenko**

October 2000

TR–00–25

INFORMATIQUE
Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle, 94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 16 49, Telex: UPVMINT 264 167 F

Laboratory of Algorithmics, Complexity and Logic
University Paris 12

Technical Report **00–25**

D. Beauquier, T. Crolard A. Slissenko.

A predicate logic framework for mechanical verification of real-time Gurevich Abstract State Machines: A case study with PVS.

October, 2000.

A Predicate Logic Framework for Mechanical Verification of Real-time Gurevich Abstract State Machines: A case study with PVS

Danièle Beauquier
E-mail: beauquier@univ-paris12.fr

Tristan Crolard
E-mail: crolard@univ-paris12.fr

Anatol Slissenko[♯]
E-mail: slissenko@univ-paris12.fr

Laboratory of Algorithmics, Complexity and Logic,
Department of Informatics, University Paris-12, France[1]

Abstract

We describe general principles and a tool that permit to get a rather short and clear PVS proof of verification of the Generalized Railroad Crossing Problem. The problem is treated completely following the initial specification practically without any modifications. Such a direct, complete and easy to understand formal treatment of the problem has never been done before. The framework we use is rather general and we believe it can be used for verification of other real-time systems.

1 Introduction

The paper describes a methodology of verification of timed abstract state machines with the help of theorem provers within a predicate logic framework. This approach is oriented onto using tools like PVS [PVS], Isabelle [Pau94] and towards development of more advanced ones that take into consideration the experience accumulated in the domain of logic based verification.

Verification presumes an interaction of two specifications: a specification of requirements (briefly, requirements) and a specification of algorithm (briefly, algorithm). Requirements, generally, consist of two parts: description of the environment and demands to the functioning. Suppose that these specifications are represented by logic formulas Φ_{Env} (environment), Φ_{Func} (functioning) and Φ_{Runs} (executions or runs of the algorithm) respectively. Thus, the verification in this situation is to prove $\big((\Phi_{Env} \wedge \Phi_{Runs}) \rightarrow \Phi_{Func} \big)$.

Logic approaches to verification have their advantages and disadvantages, say, with respect to model-checking. Their advantages: (1) they use (relatively) easily comprehensible languages to embed user's specifications without imposing on the latter new languages, (2) logic languages are very expressive, that permits to represent the initial specification directly, completely and without excessive efforts, (3) while refining the specifications the user changes his/her specification languages but the verification environment remains the same. The main disadvantage of the approach is that in its general setting the logics used are undecidable, and the search of a verification proof is not fully automatic in the general case. Our goal is to develop methods to augment the automated part of the proof search. To do it we pursue a more theoretical way of looking for decidable classes (e. g. [BS00]) and a more practical way of seeking heuristics for efficient verification. Here we present a work in the latter direction. With respect to model-checking (e. g. [CGP99])

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

we gain in the quality of the verification result, in particular in completeness, and a faster preparation of a given problem for verification, but loose in automation of the final phase.

We develop the approach described in [BS00] (and other papers available at http://www.univ-paris12.fr/lacl/) towards using proof search tools. To check how it works we analyze a well-known academic example, namely, the Generalized Railroad Crossing Problem (abbreviated to GRCP below) introduced in [HL94]. We take the version of [GH96] that is practically equivalent to the original one. The GRCP was studied in many papers, e. g. see [HM96]. A PVS verification of GRCP has been done in 70-page report [AH98]. This report is very hard to read because it uses not sufficiently efficient formalism that demands a complicated modeling. We use richer, simpler and more general languages for the specifications, in particular, First Order Timed Logic (FOTL) for requirements and timed Gurevich Abstract State Machines (ASM) [Gur95] to specify algorithms. The latter are closer to programming languages, and much stronger and much easier to use than automata based formalisms (the approach works as well for other specification languages with clear semantics). Our approach is ideologically closer to that of [MMP+96] where the authors use some kind of first order logic to specify the requirements and Petri nets to specify algorithms.

Here we present our analysis of the entire problem, based on the *initial, almost unmodified* specification. We clearly separate the specifications to analyze: requirements to functioning, specification of environment and specification of controller, and give a direct PVS proof that the controller verifies the requirements to functioning under the specification of environment. In our solution the requirements, including the environment, are described on 1-page easy to read formulas, the controller is specified by, in fact, 4-line algorithm as a Gurevich ASM (that we take with some minor modifications and rectifications from [GH96]), and our PVS proof is 3-page long. Succinctness and clarity of our solution distinguish it from [AH98]. The efficiency of our proof is based, in particular, on *general, automatically generated properties* of timed Gurevich ASM that, in particular, 'pre-process' some inductions to use.

New results of this paper are the following ones: an efficient embedding of runs of timed block Gurevich ASM in First Order Timed Logic (FOTL), an automatic tool to transform such an ASM into PVS formulas, and a short PVS proof of safety for the GRCP (liveness is somehow simpler to prove).

The paper is organized as follows. In section 2 we describe our logic framework for verification of real-time algorithms, in particular FOTL and timed block Gurevich ASM. Section 2.3 contains specifications of the GRCP. In section 3 we give FOTL formulas describing ASM runs and how to translate automatically an ASM into such formulas in PVS format. Section 4 presents the concluding part of PVS proof of (Safety) for the GRCP.

2 Background

We start with a brief summary of our methodology to verify a real-time system:

– Write down environment and requirement specifications as FOTL formulas Φ_{Env} and Φ_{Func}. Using FOTL permits, without much effort, to remain very close to the initial specifications given usually in a language close to natural ones and to avoid errors appearing in less direct modelings.

– Specify the algorithm as a timed ASM. The ASM formalism permits to be close to programming language and to express directly our algorithmic ideas.

– Automatically translate the ASM into FOTL formulas describing its runs with the help of the translator. (Notice that our method can easily be applied to another formalism than ASMs, we chosed this formalism for its clarity and its conciseness.)

– Use and develop strategies to automate partially the verification which consists in the proof of $\big((\Phi_{Env} \wedge \Phi_{Runs}) \rightarrow \Phi_{Func} \big)$.

In this section below we remind the predicate logic framework for real-time verification from [BS97], [BS00] and the notion of timed block Gurevich ASM [GH96].

2.1 First Order Timed Logic (FOTL)

A First Order Timed Logic used in this framework is constructed in two steps. Firstly, we choose a simple, if possible decidable theory to deal with concrete mathematical objects (the underlying theory), like reals, and, secondly, we extend it in a 'minimal' way by abstract functions to deal with our specifications. Here we take as the underlying theory the theory of real addition and unary multiplications by rational numbers, and

extend it by functions with at most one time argument and with other arguments being of finite abstract sorts. Precisions are given below.

Syntax of FOTL.

The vocabulary W° of a FOTL consists of a finite set of *sorts* and of a set of *function symbols*. Predicates are treated as a particular case of functions, sure except equalities for each sort.

Sorts are classified as *predefined* and *abstract*. Predefined sorts are those which interpretation is fixed. Here we consider only finite abstract sorts (though the cardinality of such a sort may be fixed, bounded or arbitrary).

We limit predefined sorts to *real numbers* $\mathbb{R}$, *time* $\mathcal{T} =_{df} \mathbb{R}_{\geq 0}$ treated as a subsort of $\mathbb{R}$, and *boolean values* $Bool$. We explicitely 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* on the one hand, and as *dynamic* and *static* on the other hand. We explain these notions just below.

The *predefined* functions are completely interpreted. We take as predefined functions the Boolean constants $true$ and $false$ of the type $\rightarrow Bool$, rational numbers $\mathbb{Q}$ treated as reals, i. e. an infinite set of functions of the type $\rightarrow \mathbb{R}$, addition $+$ and substraction $-$ of reals, infinite number of unary multiplications by rational numbers, usual binary predicates over reals: $=$, $\leq$, $<$, equality for each abstarct sort and at last, identity function over time that will represent the "current time"; we will denote it by CT° as it will appear in examples.

The set of *abstract* function symbols is finite and any abstract function symbol has at most one time argument, and the other arguments, if any, are of finite sorts.

The *dynamic* functions are those that have a time argument, and the *static* ones are those that have no such an argument.

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

Remark 1. We treat predicates as functions, nevertheless, to give more succint descriptions and avoid excessive use of equality we will use often for these functions the predicate notation.

To apply the logic to verification one needs more detailed classification of abstract symbols, in particular, one needs to distinguish *input functions* and *output functions*. We will do it when necessary.

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. Thus $\mathcal{M} \models F$, $\mathcal{M} \not\models F$ and $\models F$ where $\mathcal{M}$ is an interpretation and F is a formula, denote respectively that $\mathcal{M}$ is a model of F, $\mathcal{M}$ is a counter-model of F and F is valid.

Remark that an interpretation f^* of a function f of type $\mathcal{T} \times \mathcal{X} \rightarrow Z$ describes a family of temporal processes parametrized by the elements of interpretation $\mathcal{X}^*$ of $\mathcal{X}$.

2.2 Timed Gurevich Abstract State Machines

To represent the algorithms we use Gurevich Abstract State Machine (ASM) [Gur95]. This formalism is powerful, gives a clear vision of semantics of timed algorithms and permits to easily change the level of abstraction. In principle, Gurevich ASM may serve as an intermediate language between user's languages for algorithm specification and a logic framework. (This claim is supported by numerous applications of Gurevich ASM, see http://www.eecs.umich.edu/gasm/.)

A *timed block ASM* is a tuple of the form $(W, Init, Prog)$, where W is a vocabulary, $Init$ is a closed formula describing the initial state and $Prog$ is a program.

Sorts, variables and functions are like in subsection 2.1 except that *time cannot be an argument of functions*. We classify the functions using the same terms as in subsection 2.1, namely abstract or predefined on the one hand and static or dynamic on the other hand. Sure, to reason about the behavior of an ASM we

are to embed the functions of the vocabulary of the machine into FOTL. And at this moment, time becomes explicit to represent our vision of the functioning of the machine. To 'time' the dynamic functions of an ASM we proceed as follows.

If f is a dynamic function of type $\mathcal{X} \to Z$ in the vocabulary of an ASM, the corresponding logical function is denoted by f° and is of type $\mathcal{T} \times \mathcal{X} \to Z$. We assume that any ASM vocabulary contains a predefined dynamic function CT of type $\to \mathcal{T}$ which becomes CT°, that is the identity: $CT^\circ(t) = t$.

Dynamic functions are classified into *external* and *internal*. External functions are not changed by the ASM, internal functions, on the contrary, are computed by the ASM and are obviously abstract and dynamic. The function CT is external and predefined.

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.

The initial condition $Init$ is a closed formula over W presumed to have the following 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.

We consider a particular case of ASM, namely block ASM that consists of a set of **If-Then**-operators (*rules*). This case is, however, a very practical one. The notations of ASM are self-explanatory so we give the form of ASM we consider (our notation slightly differs from that of [Gur95] to diminish explanations).

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**
 EndInParallelDo
EndForAll
EndRepeat

Here each G_i is a *guard*, i. e. a formula over the vocabulary described above, not having free variables different from ω, and each A_i is a list of assignments (called *updates*) whose terms also do not have free variables different from ω. Each assignment is of the form $f(T) := \theta$, where f is an internal function, θ is a term and T is a list of terms of the size corresponding to the arity of f.

Informally all guards are checked simultaneously and instantaneously, and all the updates of rules with true guards are executed also simultaneously and instantaneously.

Semantics of an ASM.

Precise semantics is given in [BS00] and follows [GH96]. We give here just an intuitive description. For a given interpretation of abstract sorts we define the semantics of the program in terms of runs (executions). Informally, given an input, that is an interpretation of external functions for each moment of time, the machine computes a run which is an interpretation of internal functions for each moment of time or at least for an initial segment of $\mathcal{T}$. Notice that external functions which are classified as static have the same interpretation for every moment of time.

The behavior of the machine is deterministic. All the **If-Then**-statements are executed simultaneously in parallel and instantaneously as well as all the assignments in any **Then**-part if the corresponding guard is true. Sure, if they are inconsistent, the execution is interrupted, and the run of the algorithm becomes undefined. Notice that the effect of an assignment executed at time t takes place *after* time t but not at time t.

We consider only *total runs*, i.e. those defined on whole $\mathcal{T}$. Below "run" means "total run".

As we mentioned in Introduction, the set of runs can be described by FOTL-formulas. In the next section 3 we discuss some features of these formulas describing runs in a way apt to verification and how to produce them automatically for PVS proof checker. Before that, we illustrate this staff with the Generalized Railroad Crossing Problem.

2.3 Generalized Railroad Crossing Problem (GRCP)

Informal Description of GRCP.

We take the description of GRCP from [GH96]. A railroad crossing has several one-directional tracks and a common gate. Each track admits 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. 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 environment of the functioning of the controller to construct is described by the following assumptions. 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 $0 < d_{close} < d_{min}$. The time is presumed to be continuous.

The requirements to the controller to construct are the following ones:
(Safety). *If a train is in the crossing, the gate is closed.*
(Liveness). *The gate is open as much as possible.*

Liveness in this formulation implies second order quantifiers (for discussion see [BS00]) and has never been treated in literature in this form. So we will take below a first order formulation in terms of input/output signals (one can show that it gives the liveness in the initial formulation, but it is out the scope of this paper).

Formal Specification of the Requirements to GRCP.

FOTL permits to represent the informal requirements *directly* without any changes (modulo our remark concerning liveness).

The predefined part of vocabulary W was described above in subsection 2.1. So we define only the abstract part of W.

Abstract sorts consist of one sort $Tracks$ that represents the set of tracks which number is finite but not fixed. The *variables* for $Tracks$ are x and x with indices.

The part of vocabulary containing *abstract functions* consists of *abstract constants* (static functions of zero arity) d_{min}, d_{open}, d_{close}, all of the type $\rightarrow \mathcal{T}$, and of *abstract dynamic functions*. The latter are the following ones:

- $Cmg^{\circ} : \mathcal{T} \times Tracks \rightarrow Bool$ means a presence (coming) of a train on a track at a given time moment;
- $DirOp^{\circ} : \mathcal{T} \rightarrow Bool$ means that a signal to open the gate takes place at a given time moment, and $\neg DirOp^{\circ}$ means that the signal is to close the gate;
- $InCr^{\circ} : \mathcal{T} \rightarrow Bool$ says that there is a train in the crossing at a given time moment;
- $GtClsd^{\circ} : \mathcal{T} \rightarrow Bool$ says that the gate is closed at a given time moment;
- $GtOpnd^{\circ} : \mathcal{T} \rightarrow Bool$ 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.)

Requirements Specifications of GRCP.

Requirements consist of 2 parts: environment (formula Φ_{Env} mentioned at the beginning of section 2) and demands to the functioning (formula Φ_{Func}).

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 $\neg Cmg^{\circ}$. The alternation $\neg Cmg^{\circ}/\ Cmg^{\circ}/\ \neg Cmg^{\circ} \ldots$ corresponds to appearance of successive trains on a given track.

Notations:

- $WaitTime = WT =_{df} d_{min} - d_{close}$ will be used to describe a period of time when a train, though having been detected, is far enough from the crossing to permit to open or to not to close the gate.
- For every function f of type $\mathcal{T} \times \mathcal{X} \rightarrow \mathcal{Y}$, every term X of type $\mathcal{X}$ and every term Y of type $\mathcal{Y}$ $LimPlus_f(t,\, X,\, Y) =_{df} \exists\, t_1\, (t_1 > t \wedge \forall\, \tau\, ((t < \tau \leq t_1) \rightarrow f(\tau, X) = Y))$

$LimMoins_f(t,\, X,\, Y) =_{df} \exists\, t_1\, (t_1 < t \wedge \forall\, \tau\, ((t_1 \leq \tau < t) \rightarrow f(\tau, X) = Y))$

- A notion describing when the controller may open the gate is stated as follows:
  $SafeToOpenSp(t) =_{df}$

$\forall\, x\, [\, \neg Cmg^{\circ}(t, x) \vee \forall \tau \leq t\, (\, \forall \tau' \in [\tau, t)\, Cmg^{\circ}(\tau', x) \rightarrow t < \tau + WaitTime\,)\,]\, .$

Specification of the Environment.

(TrStInit) $\forall\, x \neg Cmg^{\circ}(0, x)$
(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) $\forall t\, (\, InCr^{\circ}(t) \rightarrow (t \geq d_{min} \wedge \exists\, x\, \forall\, \tau \in [t - d_{min},\, t]\, Cmg^{\circ}(\tau, x))\,)$
(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) $\forall\, t\, (\, \forall\, \tau \in (t, t + d_{open}] DirOp^{\circ}(\tau) \rightarrow GtOpnd^{\circ}(t + d_{open})\,)$
(If at time $t + d_{open}$ the command has been *open* for at least a duration d_{open} then the gate is opened at this time.)

(ClsClsd) $\forall t\, (\, \forall \tau \in (t,\, t + d_{close}] \neg DirOp^{\circ}(\tau) \rightarrow GtClsd^{\circ}(t + d_{close})\,)$
(If at time $t + d_{close}$ the command has been *close* for at least a duration d_{close} then the gate is closed at this time.)

(Cmg)
$\forall x \forall t\, [\, Cmg^{\circ}(t, x) \rightarrow$
$\exists\, t_0\, (\, 0 < t_0 \leq t \wedge \forall \tau \in [t_0, t]\, Cmg^{\circ}(\tau, x) \wedge LimMoins_{CmgL}(t_0, x, false)\,)\,]$

(The last property expresses 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) $0 < d_{close} < d_{min} \wedge 0 < d_{open}$
(These is trivial constraints on the durations involved, the time for closing is smaller than the minimum time of reaching the crossing by any train detected as coming.)

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).

(LIveness) or (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 using FOTL permits us to rewrite almost *directly* the environment and requirements specifications without any modelisation which could introduce a lot of errors.

Railroad Crossing Controller.

The part of the vocabulary of the ASM specific to this example consists of
Static functions:

- d_{min}, d_{open}, d_{close}; as above in the logic signature.

External functions:

- CT the current time has type $\rightarrow \mathcal{T}$.

• $Cmg : Tracks \to Bool$ is an input function giving for every track its status (coming or empty).

Internal functions:

• $DirOP$ says that the signal to open the gate is being generated by the algorithm, its type is $\to Bool$.

• $DL : Tracks \to \mathcal{T}$ is the first moment of appearance of a train on a given track plus $WaitTime$, and this value is then used as a *DeadLine* to decide on control of the gate, see $SafeToOpen$ condition below.

• $NoDL : Tracks \to Bool$ says that there is no deadline on a given track.

Notation:

$SafeToOpen =_{df} \forall\, x\ (\ \neg Cmg(x) \vee NoDL(x) \vee CT < DL(x)\ )$.

Remark that this $SafeToOpen$ is presumed to represent adequately the $SafeToOpenSp$ condition, but it is to be proved.

An algorithm to control the railroad crossing is given below. To distinguish it from that of [GH96] we will name it Symmetric Controller as it uses our version of $SafeToOpen$ condition in a symmetric way. Below we will refer to this algorithm simply as Controller.

The initial values of internal functions are defined by the condition

$$Init =_{df} \forall\, x\ (\ NoDL(x) \wedge DL(x) = 0\ ) \wedge DirOp$$

Repeat
 ForAll $x \in Tracks$
 InParallelDo
  If $Cmg(x)$ **and** $NoDL(x)$ **Then** $NoDL(x) := false$;
  $DL(x) := CT + WT$ **EndIf**
  If $\neg Cmg(x)$ **and** $\neg NoDL(x)$ **Then** $NoDL(x) := true$ **EndIf**
  If $DirOp$ **and** $\neg SafeToOpen$ **Then** $DirOp := false$ **EndIf**
  If $\neg DirOp$ **and** $SafeToOpen$ **Then** $DirOp := true$ **EndIf**
 EndInParallelDo
 EndForAll
EndRepeat

Figure 1: Railroad Crossing Controller

3 Automatic Translation of ASM runs into PVS

We describe here how ASM runs are represented by FOTL-formulas. We show how to generate automatically these formulas using the translator that we developed for this purpose[2]. The target of the translation is currently a PVS theory.

3.1 FOTL Representation of Runs of Timed ASM

It turns out that on can characterize the set of total runs of an ASM by an FOTL formula ([BS00]). Before giving this formula, we need to introduce some notations. Let W be the vocabulary of a block ASM $(W, Init, Prog)$ with the program of the form given above.

Denote by W_k the set of terms that appear to the left of $:=$ in the assignment 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 $v := \theta_{k,v},\ v \in W_k$. Denote by W° the 'timed' 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).

[2] Available at http://www.univ-paris12.fr/lacl/crolard.

(b) For terms θ over W of the form $f(\theta_1, \ldots, \theta_n)$, where f is a static function symbol, $\widehat{\theta}(t) = f(\widehat{\theta_1}(t), \ldots, \widehat{\theta_n}(t))$.

(c) For terms θ over W of the form $f(\theta_1, \ldots, \theta_n)$, where f is a dynamic function symbol, $\widehat{\theta}(t) = f^\circ(t, \widehat{\theta_1}(t), \ldots, \widehat{\theta_n}(t))$.

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

Notations:

- $Func(v)$ is the outmost function symbol of a term v, i. e. $Func(f(\theta_1, \ldots, \theta_n)) = f$.
- I_f is the set of indices k in $\{1, \ldots, m\}$ for which a term θ with $Func(\theta) = f$, that is of the form $f(\theta_1, \ldots, \theta_n)$, is in W_k.
- $Arg(\theta)$ is the list of arguments of a term θ, i. e. $Arg(f(\theta_1, \ldots, \theta_n)) = (\theta_1, \ldots, \theta_n)$. For lists of terms the operations $^\circ$ and $\widehat{\ }$ are componentwise, as well as the equality.

Below we assume that a function symbol $f \in V_{Intrn}$ has the type $\mathcal{X} \to Z$, where $\mathcal{X}$ is a direct product of sorts, and by X we will denote a vector of variables of the sort $\mathcal{X}$; to remind it we will write also $X \in dom(f)$.

To describe the total run corresponding to a given input we express the following properties related to a given moment of time t.

First some notations:

$NoGrds(t, \omega) =_{df} \bigwedge_k \neg \widehat{G_k(\omega)}(t)$

$NoGrds(t) =_{df} \forall \omega \; NoGrds(t, \omega)$

For every $f \in V_{Intrn}$:

$NoGrds_f(t, \omega) =_{df} \bigwedge_{k \in I_f} \neg \widehat{G_k(\omega)}(t)$

where $I_f = \{k \mid \exists\, v \in W_k \; Func(v) = f\}$.

$[NoGrds_f(t) =_{df} \forall \omega \; NoGrds_f(t, \omega)$

We give here a list of properties, the conjunction of which characterizes the set of total runs of an ASM.

- If no guard is valid at t then no guard is valid in some neighborhood of t :

$$OpnNoGrds(t) =_{df} \; [\, NoGrds(t) \; \to \; \exists\, t_1 t_2 \; (\, Neib(t_1, t, t_2) \; \wedge \; \forall\, \tau \, (Neib(t_1, \tau, t_2) \; \to \; NoGrds(\tau)) \;) \;] \, .$$

$\Psi_0(t) =_{df} OpnNoGrds(t)$.

- If a guard is valid at t for some ω then no guard is valid in some neighborhood of t except t itself:

$$PointWiseNoGrds(t) =_{df} \forall \omega \; \bigwedge_k \; [\, \widehat{G_k(\omega)}(t) \; \to \; \exists\, t_1 t_2 \; (\, Neib(t_1, t, t_2) \; \wedge \; \forall\, \tau (Neib(t_1, \tau, t_2) \; \to (\tau = t \; \vee \; NoGrds(\tau))) \;) \;] \, .$$

$\Psi_1(t) =_{df} PointWiseNoGrds(t)$.

- The value of an internal function at time t is equal to its value at the left of t:

For every $f \in V_{Intrn}$:

$$LeftOpn_f(t) =_{df} \; (\; t > 0 \; \to \; \exists\, t_1 < t \, \forall\, \tau \; \in \; (t_1, t) \, \forall\, X \in dom(f) \; f^\circ(\tau, X) \; = \; f^\circ(t, X) \;) \, .$$

$LeftOpn(t) =_{df} \bigwedge_{f \in V_{Intrn}} LeftOpn_f(t)$

$\Psi_2(t) =_{df} LeftOpn(t)$.

- Values of internal functions do not change as long as related guards remain false:

$NoGrdNoChange_f(t) =_{df}$

$$(\; \forall\, t_1 > t \, (\forall\, \tau \; \in \; [t, t_1) \, NoGrds_f(\tau) \; \to \; \forall\, \tau \; \in \; (t, t_1] \, \forall\, X \; \in \; dom(f) \, f^\circ(\tau, X) \; = \; f^\circ(t, X)) \;)$$

$NoGrdNoChange(t) =_{df} \bigwedge_{f \in V_{Intrn}} NoGrdNoChange_f(t)$.

$\Psi_3(t) =_{df} NoGrdNoChange(t)$.

- If some guard related to the internal function f is true at t, f is updated in according with the update rule of this guard, and the update holds as long as the guards related to f remain false:

$UpDate_f(t, \omega) =_{df} \bigwedge_{k \in I_f} (\, \widehat{G_k(\omega)}(t) \; \to$

$$\forall\, t_1 > t \, (\forall\, \tau \; \in \; (t, t_1) NoGrds_f(\tau)) \; \to \; \forall\, \tau \; \in \; (t, t_1] \bigwedge_{v \in W_k, \, Func(v) = f} \widehat{v(\omega)}(\tau) = \widehat{\theta_{k,v}(\omega)}(t)) \;)$$

$UpDate(t) =_{df} \forall\, \omega \; \bigwedge_{f \in V_{Intrn}} UpDate_f(t, \omega)$

$\Psi_4(t) =_{df} UpDate(t)$.

• If some guard related to the internal function f is true at t, values of f not updated remain the same as long as the guards related to f remain false:

$NoChange_f(t, X) =_{df} \forall \omega \bigwedge_{k \in I_f,\, v \in W_k,\, Func(v)=f} \widehat{Arg(v(\omega))}(t) = X \rightarrow \neg\widehat{G_k(\omega)}(t)$

$NoUpDate_f(t) =_{df} \exists \omega \bigvee_{k \in I_f} \widehat{G_k(\omega)}(t) \rightarrow \forall t_1 > t \; (\forall \tau \in (t, t_1) NoGrds_f(t) \rightarrow$
$\forall X \in dom(f)(NoChange_f(t, X) \rightarrow \forall \tau \in (t, t_1]\, f^\circ(\tau, X) = f^\circ(t, X)) \;)$

$NoUpDate(t) =_{df} \bigwedge_{f \in V_{Intrn}} NoUpDate_f(t)$.

$\Psi_5(t) =_{df} NoUpDate(t)$.

Finally, the total runs are described by the formula Ψ

$$\Psi =_{df} \widehat{Init}(0) \bigwedge_{i=0,\ldots,5} \forall t \, \Psi_i(t)$$

In other words we have the following theorem [BS00]:

Theorem 1. *Every model of Ψ is a total run of $\mathcal{A}$, and conversely, every total run of $\mathcal{A}$ is a model of Ψ.*

Theoretically, every property satisfied by a run of the ASM can be deduced from Ψ. Nevertheless, from a practical point of view, formulas involved in Ψ are not always the most convenient ones. For this reason a richer library of formulas deducible from Φ_{Runs} has been developed.

Library of formulas deducible from Φ_{Runs}.

We describe these formulas in a natural language. The exact formulas are in Appendix D.

• $OpnNoGrds(t)$: The set of time moments t when some guard is false is an open set.

• $NotInt_G$: A guard G cannot be true on an open interval.

• $PointWiseNoGrds(t)$: Guards are pointwise.

• $FirstChange_f(t)$: If an internal function f has not the same value in t and t', there is a first moment between t and t' when it changes its value.

• $LastChange_f(t)$: If an internal function f has different values at t and t', there is a last moment between t and t' when it changes its value.

• $UpDateLoc_f(t)$: If some guard related to the internal function f is true at t, then f is updated according to the update rule of this guard at the right of t.

Well Parametrized GASM.

The verification of the Railroad Crossing Problem has emphasized the fact that some special ASM have interesting properties. Sure, these properties can be deduced from Φ_{Runs}, but it is more convenient to write them directly and automatically.

Definition. An ASM is *well-parametrized* if

(a) every internal function f has a type $\mathcal{X} \rightarrow \mathcal{Z}$ or $\Omega \times \mathcal{X} \rightarrow \mathcal{Z}$, where $\mathcal{X}$ is a product of finite sorts *different* from Ω;

(b) in every update concerning f, i. e. of the form $f(\Theta) := \Theta'$, with f of the type $\Omega \times \mathcal{X} \rightarrow \mathcal{Z}$ the first argument is the *variable* ω of sort Ω, mentioned explicitly in the general form of ASM given in subsection 2.2, that is $f(\Theta)$ has the form $f(\omega, \theta_1, \ldots, \theta_n)$.

For a well parametrized block ASM, one can prove a property that is not true in the general case: internal functions depend only "locally" on the parameter $\omega \in \Omega$.

More precisely, for every internal function f, for every $k \in I_f$ let us define:

$UpDatePar_{f,k}(t, \omega) =_{df}$
$\forall t_1 \, (t < t_1 \wedge G_k(t, \omega) \wedge (\forall \tau \in (t, t_1)\, NoGrds_f(t, \omega)) \rightarrow$
$\forall \tau \in (t, t_1] \bigwedge_{v \in W_k,\, Funv(v)=f} \widehat{v(\omega)}(\tau) = \widehat{\theta_{k,v}(\omega)}(t))$

Theorem 2 *If an ASM is well-parametrized then the following property holds*

$\bigwedge_{f \in V_{Intrn}} \bigwedge_{k \in I_f} \forall t \forall \omega \, UpDatePar_{f,k}(t, \omega)$.

The proof of this theorem is by induction on the number of time moments τ between t and t_1 when $NoGrds_f(\tau)$ is false. Using Ψ_3, we prove that due to the fact that the ASM is well-parametrized, in such time moments the value of $\widehat{v(\omega)}(\tau)$ remains equal to $\widehat{\theta_{k,v}(\omega)}(t)$.

3.2 Automatic Generation of Description of ASM Runs

The main task of the translator is to manipulate abstract terms and formulas, it is thus easier to program in a language with concrete data types and a powerful type system like ML (we used Ocaml 3.0 [Ler00]).

Composition of the Translator.

The translator is composed of a parser, a type checker, a library of functions which actually perform the translation and a printer which can generate a correct PVS theory from definitions and axioms (or lemmas) given in our abstract syntax (see below). Because of this modular design, it is quite simple to extend the syntax of ASM, to generate new lemmas or to use another prover than PVS.

The input file.

An example of input file for the Railroad Crossing Problem as well as the formal grammar are given in Appendix C and Appendix B respectively.

The input file containing the description of the ASM is three-fold:

• The first part contains the signature, where we declare the type of every function or predicate symbol. A type has either the form τ_0 or $\tau_1 \times \ldots \times \tau_n \to \tau_0$ where each τ_i is an atomic type (i.e. a name). Then we give the (optional) list of static functions and the (optional) list of internal functions.

• The second part contains the signature of logical variables (variables which may occur only bound by a quantifier or as formal parameters in a definition) and the definitions of some functions (or predicates). The general form of such a definition is: $f(x_1, \ldots, x_n) : \tau = t$, where $x_1, \ldots, x_n$ are variables, τ is an atomic type and t is a term.

• The third part contains the rules of the ASM following the syntax given in subsection 2.2. Eventually, a list of symbols that should be exported (i.e. since they were not defined in the environment) can be specified.

Abstract syntax.

Parsing the input file results in abstract syntactic trees which belong to the following ML data types. Since this abstract syntax corresponds directly to the concrete syntax defined by the formal grammar, it does not require much comment. Note however that we do not distinguish between terms and formulas at the syntactic level (formulas are just boolean terms) and that the ASM may or not be parametrized.

```
type asm_type = name list * name;;
type term = Constant of rational
          | Apply of (name * term list)
          | Forall of (variable list * term)
          | Exists of (variable list * term);;
type rule = term * (term * term) list;;
type prog = { decl :  (variable * asm_type) list;
              static :  variable list;
              internal :  variable list;
              var :  (variable * asm_type) list;
              def :  (term * name * term) list;
              body :  (variable * asm_type) option * rule list;
              export :  variable list };;
```

Type checking.

When the parsing stage is over, the abstract syntactic tree is checked. We check that no function is declared both static and internal, that no external function is assigned in the ASM and finally we type check the definitions and the conditional rules. Note that since some provers (such as PVS) use stronger type systems

(possibly generating TCC), a type error during the translation does not stop the processing (but should be considered as a warning).

Predefined atomic types are `bool`, `rational` (for constants), `time` and `real`. The types of the predefined functions and predicates enumerated in section 2.1 are also known by the translator:

Predefined static functions:

```
+ :  time * time -> time
= :  time * time -> bool
/ :  rational * rational -> rational
* :  real * rational -> real
+ :  real * real -> real
- :  real * real -> real
= :  real * real -> bool
< :  real * real -> bool
=> :  bool * bool -> bool
= :  bool * bool -> bool
AND : bool * bool -> bool
OR : bool * bool -> bool
NOT : bool -> bool
true :  bool
false :  bool
```

Predefined dynamic function

```
CT : time
```

Subtyping.

A very simple but convenient form of subtyping is provided. The usual typing rule of application is generalized to the following rule:

$$\frac{t_i : \tau_i \quad \tau_i \subset \tau_i' \quad f : \tau_1' \times \ldots \times \tau_n' \to \tau}{f(t_1, \ldots, t_n) : \tau}$$

As expected, the predefined types obey the following inclusion: `time`$\subset$`real` and `rational`$\subset$`real`.

Overloading.

Two kinds of overloading are allowed: a function can be re-declared either if the type of its arguments are incomparable with those of the first declaration (true overloading), or if the type of its arguments and results are subtypes of those of the first declaration (restriction). An example of true overloading is `= : bool * bool -> bool` and `= : real * real -> bool`. An example of restriction is given by `+ : time * time -> time` and `+ : real * real -> real`.

The translation.

The implementation of the translation is very close to its mathematical definition given in subsection 3.1. For instance, a function `hat : name list -> term -> term -> term` implements "$\hat{}$". Its parameters are the list of dynamic functions, a term (which is usually a variable of type `time`) and the term to be transformed. Likewise, some functions `circ`, `func` and `arg` implements "$\circ$", $Func$ and Arg exactly as they are defined in section 3. A few auxilliary functions are also needed to generate generalized quantifications like $\forall X \in dom(f)$ (which occur for instance in the definition of *NoGrdNoChange*) according to the actual type of f. For that purpose, the variables named $x_1, x_2, x_3, \ldots$ might be used by the translator and thus they should not occur in the input file (to avoid any capture). For the same reason, the variables t, t_1, t_2, t_3 are also reserved.

Since the details of the translation are rather technical, we will just describe the main steps:

- we translate the signature of the ASM (we use `circ` to make the time explicit in the type of dynamic functions)

- we translate the definitions of the ASM (we use `hat` to add a time argument when dynamic functions are applied)
- we build definitions named:
  - `Guard`*i* for each guard *i* .
  - `NoGrdsF` and `NoChangeF` for each *internal* function `F`.
  - `LimPlusF` and `LimMinusF` for each *dynamic* function `F`.
- we build axioms named:
  - `OpnNoG`*i* and `PointWise`*i* for each guard *i*.
  - `NoGrdNoChangeF` and `LeftOpnNoF` for each *internal* function `F`.
  - `UpDateNoF`*i* and `NoUpDateNoF`*i* for each occurrence*i* of an update of the *internal* function `F`.
- we build lemmas named:
  - `OpenNoGrds`
  - `NotIntG`*i* and `PointWiseNoGrds`*i* for each guard *i*.
  - `FirstChangeF` and `LastChangeF` for each *internal* function `F`.
  - `UpDateLocF`*i* for any occurrence *i* of an update of the *internal* function `F`.
- eventually, we check if the ASM is well parametrized according to the criterion defined in section 3 and if so, we generate additional lemmas named:
  - `UpDateParF`*i* for any occurrence *i* of an update of the *internal* function `F`.

About Higher Order Logic.

We chose to translate the first order theory which characterizes runs of an ASM directly as a first order theory in PVS. Thus we do not benefit from the full logic of PVS (which is a Higher Order Logic). We could use functionals (for the definition of limits) and axiom schemes to translate the general theory of runs parametrized by some encoding of the ASM. The proofs of properties of a specific ASM would not be made easier, but the generated theory would be shorter and far more elegant (2-page theory instead of a 9-page one). Moreover, we could consider to try and derive our lemmas from the axioms in PVS (since FOTL is undecidable, the success of such an attempt depends on the lemmas). In fact, we chose to restrict ourselves to first order theories because it is the only common part for a variety of existing provers.

3.3 Generating a PVS theory

PVS.

PVS is a proof checker with some facilities of theorem prover, see [PVS]. It permits to introduce richly typed logic syntax of any order. PVS has a type checker that is very useful. There is some number basic strategies and a limited language to write down user's strategies. The interaction with the user is well done that makes the system a rather efficient proof checker. There are incorporated decision algorithms, automatic rewriting and other procedures.

The resulting file of our translation is a PVS theory which has the following form:

```
ASM : THEORY
BEGIN
IMPORTING Environment
% Signature
% Definitions
% Axioms and Lemmas
END ASM
```

In the signature part, we only declare functions which are exported by the ASM (which follow the keyword `export`), the other functions should be declared in the theory "`Environment`".

4 A Case Study: Verification of the Generalized Railroad Crossing Problem

In this section we give the main steps of a PVS proof of (Safety) for GRCP.

4.1 PVS Proof of Safety for the GRCP

The PVS checked proof is based on three lemmas ($InCrArr$), ($ArrDL$) and ($NotSfDirCl$).

Lemma *(InCrArr)* says that if a train is in the crossing at time t then there exists a time $t_1 \geq d_{min}$ before t such that this train was arriving at t_1 and remained in the zone of control during the period $[t_1, t]$.

Lemma *(ArrDL)* expresses the fact that if a train arrives at time t on track x, and remains in the zone of control during a period $[t, t_1]$, then there is no deadline for x at time t but on the interval $(t, t_1]$, there is a deadline for x equal to $t + WT$.

Lemma *(NotSfDirCl)* says that if during a period $[t, t_1]$ with $t < t_1$ it is not safe to open (from the point of view of the algorithm) then during the period $(t, t_1]$ there is a command to close the gate.

As shown by the lemmas introduced, an important event is the arrival of a train on some track x at time moment t, this event can be expressed as an FOTL formula $Arrive(t, x)$.

The proof of each lemma was checked with the help of PVS. The complete proof is less than 4-page, it is given in Appendix E. As an illustration, we compare a sketch of our hand proof of ($Safety$) that uses the above lemmas and the corresponding PVS proof.

Remind that ($Safety$) means
$\forall t\,(InCr^{\circ}(t) \rightarrow GtClsd^{\circ}(t))$.

Hand Proof of Safety.

(Sf0) Φ_{Env} contains an axiom (dIneq): $0 < d_{close} < d_{min} \ \wedge 0 < d_{open}$ that we will use.

(Sf1) Fix a time moment t_1 and suppose $InCr^{\circ}(t_1)$, our goal is to prove
$GtClsd^{\circ}(t_1)$.

(Sf2) From environment formulas (ClsClsd) and (InCr) given at the end of section 2.1 (using basic first order proof search considerations) we reduce our goal to $\big(\ t_1 \geq d_{close} \ \wedge \forall \tau \in (t_1 - d_{close}, t_1]\, \neg DirOp^{\circ}(\tau)\ \big)$.

(Sf3) Applying Lemma (InCrArr) at time t_1 we get that there is some track x_0 and some time t_2 such that
(Sf3.1): $t_2 \leq t_1 - d_{min}$,
(Sf3.2): $Arrives(t_2, x_0)$ and
(Sf3.3): $\forall \tau \in [t_2, t_1]\, Cmg^{\circ}(\tau, x_0)$.

(Sf4) Applying Lemma (ArrDL) for t_1, x_0, t_2 we get
$\forall\, \tau \in (t_2, t_1]\, (\neg NoDL^{\circ}(\tau, x_0) \wedge DL^{\circ}(\tau, x_0) = t_2 + WT)$

(Sf5) The first part of our goal stated in (Sf2), i. e. $t_1 \geq d_{close}$, follows from (Sf3.1) and axiom (dIneq) mentioned in (Sf0).

(Sf6) To prove the second part of the goal (Sf2) we apply Lemma (NotSfDirCl) for $t_1 - d_{close}$, t_1 and get
$\big(\ t_1 - d_{close} < t_1 \ \wedge \forall \tau \in [t_1 - d_{close}, t_1]\, \neg \widehat{SafeToOpen}(\tau)\ \big)$
$\rightarrow \forall \tau \in (t_1 - d_{close}, t_1]\, \neg DirOp^{\circ}(\tau)$.

(Sf7) To finish the proof of goal (Sf2) it is enough to prove
(Sf7.1): $t_1 - d_{close} < t_1$ and
(Sf7.2): $\forall \tau \in [t_1 - d_{close}, t_1]\, \neg \widehat{SafeToOpen}(\tau)$.

(Sf8) If we expand the definition of $\widehat{SafeToOpen}$ formula (Sf7.2) becomes
$\forall \tau \in [t_1 - d_{close}, t_1]\, \exists x\, \big(\ Cmg^{\circ}(\tau, x) \ \wedge \neg NoDL^{\circ}(\tau, x) \wedge CT^{\circ}(\tau) \geq DL^{\circ}(\tau, x)\ \big)$.

(Sf9) Let $\tau_1 \in [t_1 - d_{close}, t_1]$. We are to prove:

$\exists x \ \big(\ Cmg^\circ(\tau, x) \ \wedge \ \neg NoDL^\circ(\tau, x) \wedge CT^\circ(\tau) \geq DL^\circ(\tau, x) \ \big)$.

(Sf10) To prove (Sf9) it is enough to prove

$\big(\ Cmg^\circ(\tau_1, x_0) \ \wedge \ \neg NoDL^\circ(\tau_1, x_0) \wedge CT^\circ(\tau_1) \geq DL^\circ(\tau_1, x_0)) \ \big)$.

(Sf11) Instantiating (Sf3.3) and (Sf4) with τ_1 and using (dIneq) we deduce from the definition of CT° the desired result.

PVS Proof of Safety.

We briefly comment PVS commands and strategies used below: SKOSIMP is a skolemization followed by a disjunctive simplification, GRIND is a catch-all strategy used to automatically complete a proof branch, GROUND is a less powerful command which invokes propositional simplification and decision procedures, APPLY_LEMMA is a strategy (its definition is given in Appendix E.1) we have written which permits to apply some lemma of the form $\forall X(\phi(X) \rightarrow \psi(X))$, instantiating it, and then splitting it.

Numbers in square brackets are added by us to show the correspondence between hand-made proof above and the PVS proof below.

```
;;; Proof for formula verif.Safety
(""
[0](APPLY_LEMMA "dIneq" NIL)
[1](SKOSIMP*)
[2](APPLY_LEMMA "InCrArr" ("t!1"))
    (("1"
[3]   (APPLY_LEMMA "ClsClsd" ("t!1"))
      (("1" (GRIND))
       ("2"
        (SKOSIMP*)
[4]     (APPLY_LEMMA "ArrDL" ("t3!1" "t!1" "x!1"))
        (("1"
[6]      (LEMMA "NotSfDirCl")
         (INST -1 "t!1-dclose" "t!1")
         (("1"
[7]        (CASE "(FORALL tau:((t!1 - dclose <= tau AND tau <= t!1) =>
                SafeToOpen(tau) = FALSE))")
           (("1" (GRIND))
            ("2"
[8]          (EXPAND "SafeToOpen")
[9]          (SKOSIMP 1)
[10]         (INST -3 "x!1")
[11]         (INST -6 "tau!2")
             (INST -11 "tau!2")
             (GRIND))))
[5]      ("2" (ASSERT))))
        ("2" (GROUND))))))
```

Conclusion

The result presented in this paper shows that a rather complete logic based verification of real-time systems is feasible and that this approach has quite visible advantages with respect to other ones. To make this technique more practical we are developing proof search strategies that considerably augment the automated part of the proof search. Besides that we plan to extend our approach to other algorithm specification languages (in particular, fragments of programming languages) and to use more powerful logics (second order and with probability operator) that permit to treat richer classes of verification problems.

References

[AH98] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. Technical Report 5546–98–8180, University Paris-12, Department of Informatics, Naval Reserach Laboratory, Washington, 1998. NRL Memorandum Report.

[BS97] 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/.

[BS00] D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms: Basic properties and a decidable class. 36 pages. To appear in *Annals of Pure and Applied Logic.*, 2000.

[CGP99] E. Clarke, O. Grumberg, and D. Peleg. *Model Checking.* MIT Press, Boston, MA., 1999.

[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.

[Gur95] Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–93. Oxford University Press, 1995.

[HL94] C. Heitmeyer and N. Lynch. The generalized railroad crossing: a case study in formal verification of real-time systems. In *Proc. of Real-Time Systems Symp., San Juan, Puerto Rico.* IEEE, 1994.

[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.

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

[Ler00] X. Leroy et al. *The Objective Caml system release 3.00. Documentation and user 's manual.* INRIA, April 2000. ftp://ftp.inria.fr/lang/caml-light/ocaml-3.00-refman.ps.gz.

[MMP+96] D. Mandrioli, A. Morzenti, M. Pezzè, P. San Pietro, and S. Silva. A Petri net and logic approach to the specification and verification of real-time systems. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, pages 135–165. John Wiley & Sons, 1996.

[Pau94] L. C. Paulson. *Isabelle. A generic Theorem Prover.* Springer-verlag, 1994. Lect. Notes in Comput. Sci., vol. 828.

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

Appendices

A Library of formulas deducible from Φ_{Runs}

• The set of time moments t when some guard is false is an open set:

$$OpnNoG_k(t,\omega) =_{df} \left(\neg \widehat{G_k(\omega)}(t) \rightarrow \exists\, t_1 t_2 \ \left(\ Neib(t_1,t,t_2) \wedge \forall\, \tau\, (Neib(t_1,\tau,t_2) \rightarrow \neg \widehat{G_k(\omega)}(\tau)) \ \right) \right)$$

$$OpnNoG_k =_{df} \forall t\, \forall \omega\, OpnNoG_k(t,\omega)$$

We have the following property:

$$\bigwedge_k OpnNoG_k.$$

• Guards are pointwise :

$$PointWiseG_k(t,\omega) =_{df} \left(\widehat{G_k(\omega)}(t) \rightarrow \right.$$

$$\left. \exists\, t_1 t_2 \ \left(\ Neib(t_1,t,t_2) \wedge \forall\, \tau\, (Neib(t_1,\tau,t_2) \rightarrow (\tau = t \vee \neg \widehat{G_k(\omega)}(\tau))) \ \right) \right)$$

$$PointWiseG_k =_{df} \forall t\, \forall \omega\, PointWiseG_k(t,\omega)$$

We have:

$$\bigwedge_k PointWiseG_k.$$

• If some guard related to the internal function f is true at t, f is updated in according with the update rule of this guard at the right of t.

$$UpDateLoc_f(t,\omega) =_{df} \bigwedge_{k \in I_f} \left(\widehat{G_k(\omega)}(t) \rightarrow \bigwedge_{v \in W_k,\, Func(v)=f} LimPlus_{\widehat{v(\omega)}}(t,\ \widehat{\theta_{k,v}(\omega)}(t)) \right)$$

$$UpDateLoc(t) =_{df} \forall\, \omega \bigwedge_{f \in V_{Intrn}} UpDateLoc_f(t,\omega)$$

We have:

$\forall\, t\, UpDateLoc(t)$.

• A guard cannot be true on an open interval:

$$NotIntG_k(t,\omega) =_{df} \forall\, t_1 \ \left(\ t < t_1 \rightarrow \neg(\forall\, \tau \in (t,t_1)\ G_k(\tau,\omega) \right)$$

$$NotIntGrd(t) =_{df} \forall\, \omega \bigwedge_k NotIntG_k(t,\omega)$$

We have

$\forall\, t\, NotIntGrd(t)$.

• If an internal function f has not the same value in t and t', there is a first moment between t and t' when it changes its value:

$$Change_f(t,X) =_{df} \bigvee_{k \in I_f} \widehat{G_k(\omega)}(t) \ \wedge \ \exists\, y\ (LimMoins_{\widehat{f(X)}}(t,y) \wedge y \neq f^{\circ}(t,X))$$

$$FirstChange_f(t,t',X) =_{df} (t < t' \wedge f^\circ(t,X) \neq f^\circ(t',X)) \rightarrow$$

$$(\exists\, t" \in [t,t')(Change_f(t",X) \wedge \forall\, \tau \in [t,t") \bigwedge_{k \in I_f} \forall\, \omega \neg \widehat{G_k(\omega)}(\tau))$$

We have : $\bigwedge_{f \in V_{Intrn}} \forall t t' \forall X \in dom(f)\ FirstChange_f(t,t',X)$.

• If an internal function f has not the same value in t and t', there is a last moment between t and t' when it changes its value:

$$LastChange_f(t,t',X) =_{df} (t < t' \wedge f^\circ(t,X) \neq f^\circ(t',X)) \rightarrow$$

$$(\exists\, t" \in [t,t')(Change_f(t",X) \wedge \forall\, \tau \in (t",t') \bigwedge_{k \in I_f} \forall\, \omega \neg \widehat{G_k(\omega)}(\tau))$$

B Concrete syntax of an input file (ASM) for the translator

```
        prog ::= "Signature" ident decl-list def-body export-list

    decl-list ::= "End"
                | "Internal" ident var-list "End"
                | "Static" ident var-list internal-list
                | decl decl-list

internal-list ::= "End"
                | "Internal" ident var-list "End"

  export-list ::= "End"
                | "Export" ident var-list "End"

        decl ::= ident ":" ident type
                | string ":" ident type

     var-list ::= ";"
                | "," ident var-list

        type ::= "->" ident ";"
                | "*" ident type
                | ";"

     def-body ::= "Variables" typed-var var-list def-list body "EndRepeat"
                | "Define" def-list body "EndRepeat"
                | "Repeat" body "EndRepeat"

    typed-var ::= "Define"
                | decl typed-var

     def-list ::= "Repeat"
                | definition ";" def-list

   definition ::= term ":" ident "=" term

        body ::= "Forall" ident "in" ident "InParallelDo" if-list "EndForall"
                | "InParallelDo" if-list

      if-list ::= "EndDo"
                | "If" if if-list

          if ::= term "Then" assign-list

  assign-list ::= "EndIf"
                | assign ";" assign-list

       assign ::= term ":=" term

    term-list ::= ")"
                | "," term term-list
                | op term op-list
```

```
    term ::= int
           | float
           | ident arg-list
           | "(" term op-list
           | "not" term
           | "forall" ident term
           | "exists" ident term

 op-list ::= ")"
           | op term op-list

arg-list ::= "(" term term-list

      op ::= "AND" | "OR" | "=>" | "=" | "<" | ">" | "<=" | ">="
           | "*" | "/" | "+" | "-"
```

C The Railroad Crossing Problem ASM (input file for the translator)

```
Signature Railroad
  WT : time;
  dclose :  time;
  dmin :  time;
  DL : Tracks -> time;
  NoDL : Tracks -> bool;
  Cmg :  Tracks -> bool;
  DirOp :  bool;
  SafeToOpen :  bool;
  Init :  bool;

  Static WT, dmin, dclose;

  Internal NoDL, DL, DirOp;
 End


Variables

  x :  Tracks;

Define

  WT : time = (dmin - dclose);
  SafeToOpen :  bool = forall x (not Cmg(x) or NoDL(x) or CT < DL(x));
  Init :  bool = forall x (DirOp and NoDL(x) and DL(x) = 0);

  Repeat
    Forall x in Tracks
      InParallelDo
        If (Cmg(x) and NoDL(x)) Then DL(x) := (CT +WT); NoDL(x) := false;
        EndIf
        If (not Cmg(x) and not NoDL(x)) Then NoDL(x) := true;
        EndIf
        If (DirOp and not SafeToOpen) Then DirOp := false;
        EndIf
        If (not DirOp and SafeToOpen) Then DirOp := true;
        EndIf
      EndDo
    EndForall
  EndRepeat

  Export DL, NoDL;

End
```

D PVS Theories for the Generalized Railroad Crossing Problem

There are four theories, *deftime* which contains the definitions related to time, *Environment* which contains the specifications of the environment and imports *deftime*, *Railroad* which is *automatically* produced from the ASM by our translator program modeling the control system and which imports *Environment* and at last *verif* which imports *Railroad* and contains the lemmas to prove.

D.1 Theory: deftime

deftime: THEORY
BEGIN

time: TYPE = $\{t$: real $\mid t \geq 0\}$

t, t_1, t_2, a, b, c: VAR time;

CT(t): time = t

Neigh(t_1, t, t_2): bool = $(t = 0 \wedge t_1 = 0 \wedge t_2 > 0) \vee (t_1 < t \wedge t < t_2)$

P: VAR [time $\rightarrow$ bool]

union_int: LEMMA
$(\forall\ t$: $((a \leq t \wedge t \leq b) \Rightarrow$
$P(t)) \wedge (\forall\ t$: $(b \leq t \wedge t \leq c) \Rightarrow P(t)) \wedge (a \leq b \wedge b \leq c)) \Rightarrow (\forall\ t$: $(a \leq t \wedge t \leq c) \Rightarrow P(t))$
END deftime

D.2 Theory: Environment

Environment: THEORY
BEGIN

IMPORTING deftime

Tracks: TYPE;

dclose: time;

dmin: time;

InCr, GtClsd, GtOpnd: [time $\rightarrow$ bool]

Cmg: [time, Tracks $\rightarrow$ bool];

DirOp: [time $\rightarrow$ bool];

t, t_0, t_1, t_2, t_3, τ: VAR time;

x: VAR Tracks;

dopen: time;

LimMinusCmg(t: time, x_1: Tracks, y: bool): bool =
$(\exists\ t_1$: $(t_1 < t \wedge (\forall\ t_3$: $((t_1 \leq t_3 \wedge t_3 < t) \Rightarrow$ Cmg(t_3, x_1) = $y))))$

dIneq: AXIOM $\text{dclose} > 0 \wedge \text{dmin} > 0 \wedge \text{dmin} > \text{dclose} \wedge \text{dopen} > 0$

TrStInit: AXIOM $\text{Cmg}(0,\ x) = \text{FALSE}$

GtStInit: AXIOM $\text{GtOpnd}(0) = \text{TRUE}$

GtSt: AXIOM $\neg\ (\text{GtOpnd}(t) = \text{TRUE} \wedge \text{GtClsd}(t) = \text{TRUE})$

CrCm: AXIOM
$\text{InCr}(t) = \text{TRUE} \Rightarrow$
$(t \geq \text{dmin} \wedge$
$(\exists\ x\colon \forall\ \tau\colon\ (t - \text{dmin} \leq \tau \wedge \tau \leq t) \Rightarrow \text{Cmg}(\tau,\ x) = \text{TRUE}))$

OpnOpnd: AXIOM
$(t \geq \text{dopen} \wedge (\forall\ \tau\colon\ ((\tau > t - \text{dopen} \wedge \tau \leq t) \Rightarrow \text{DirOp}(\tau) = \text{TRUE}))) \Rightarrow$
$\text{GtOpnd}(t) = \text{TRUE}$

ClsClsd: AXIOM
$(t \geq \text{dclose} \wedge (\forall\ \tau\colon\ ((\tau > t - \text{dclose} \wedge \tau \leq t) \Rightarrow \text{DirOp}(\tau) = \text{FALSE}))) \Rightarrow$
$\text{GtClsd}(t) = \text{TRUE}$

Coming: AXIOM
$\text{Cmg}(t,\ x) = \text{TRUE} \Rightarrow$
$(\exists\ t_0\colon$
$(0 < t_0 \wedge$
$t_0 \leq t \wedge$
$(\forall\ \tau\colon\ (t_0 \leq \tau \wedge \tau \leq t) \Rightarrow (\text{Cmg}(\tau,\ x) \wedge \text{LimMinusCmg}(t_0,\ x,\ \text{FALSE}))) \wedge$
$(\exists\ t_1\colon$
$(t_1 > t \wedge$
$(\forall\ \tau\colon\ (((\text{Neigh}(t,\ \tau,\ t_1)) \Rightarrow \text{Cmg}(\tau,\ x) = \text{TRUE})))))))$

END Environment

D.3 Theory: Railroad

This theory is generated automatically by our translator from the ASM specification of Appendix C.

%%% Parsing : ok

%%% Type Checking :

%%% Type error in definition WT=(dmin - dclose)

%%% Static : x, WT, dmin, dclose,

%%% Internal : NoDL, DL, DirOp,

Railroad: THEORY
BEGIN

IMPORTING Environment

% dclose : time;

% dmin : time;

DL: [time, Tracks → time];

NoDL: [time, Tracks → bool];

% Cmg : [time, Tracks -> bool];

% DirOp : [time -> bool];

x: VAR Tracks;

t, t_1, t_2, t_3: VAR time;

WT: time = (dmin − dclose)

SafeToOpen(t): bool =
(∀ x: (¬ Cmg(t, x) ∨ (NoDL(t, x) ∨ CT(t) < DL(t, x))))

Init(t): bool = (∀ x: (DirOp(t) ∧ (NoDL(t, x) ∧ DL(t, x) = 0)))

Grd1(t: time, x: Tracks): bool = (Cmg(t, x) ∧ NoDL(t, x))

Grd2(t: time, x: Tracks): bool = (¬ Cmg(t, x) ∧ ¬ NoDL(t, x))

Grd3(t: time, x: Tracks): bool = (DirOp(t) ∧ ¬ SafeToOpen(t))

Grd4(t: time, x: Tracks): bool = (¬ DirOp(t) ∧ SafeToOpen(t))

NoGrdsNoDL(t: time, x: Tracks): bool = (¬ Grd1(t, x) ∧ ¬ Grd2(t, x))

NoGrdsDL(t: time, x: Tracks): bool = ¬ Grd1(t, x)

NoGrdsDirOp(t: time, x: Tracks): bool = (¬ Grd3(t, x) ∧ ¬ Grd4(t, x))

```
LimMinusNoDL(t: time, x1: Tracks, y: bool): bool =
    (∃ t1: (t1 < t ∧ (∀ t3: ((t1 ≤ t3 ∧ t3 < t) ⇒ NoDL(t3, x1) = y))))

LimMinusDL(t: time, x1: Tracks, y: time): bool =
    (∃ t1: (t1 < t ∧ (∀ t3: ((t1 ≤ t3 ∧ t3 < t) ⇒ DL(t3, x1) = y))))

LimMinusDirOp(t: time, y: bool): bool =
    (∃ t1: (t1 < t ∧ (∀ t3: ((t1 ≤ t3 ∧ t3 < t) ⇒ DirOp(t3) = y))))

LimPlusNoDL(t: time, x1: Tracks, y: bool): bool =
    (∃ t1: (t1 > t ∧ (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ NoDL(t3, x1) = y))))

LimPlusDL(t: time, x1: Tracks, y: time): bool =
    (∃ t1: (t1 > t ∧ (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DL(t3, x1) = y))))

LimPlusDirOp(t: time, y: bool): bool =
    (∃ t1: (t1 > t ∧ (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t3) = y))))

ChangeNoDL(t: time, x1: Tracks): bool =
    ((∃ x: (Grd1(t, x) ∨ Grd2(t, x))) ∧
      (∃ (y: bool): (LimMinusNoDL(t, x1, y) ∧ ¬ y = NoDL(t, x1))))

ChangeDL(t: time, x1: Tracks): bool =
    ((∃ x: Grd1(t, x)) ∧
      (∃ (y: time): (LimMinusDL(t, x1, y) ∧ ¬ y = DL(t, x1))))

ChangeDirOp(t: time): bool =
    ((∃ x: (Grd3(t, x) ∨ Grd4(t, x))) ∧
      (∃ (y: bool): (LimMinusDirOp(t, y) ∧ ¬ y = DirOp(t))))

NoChangeNoDL(t: time, x1: Tracks): bool =
    (∀ x: ((x = x1 ⇒ ¬ Grd1(t, x)) ∧ (x = x1 ⇒ ¬ Grd2(t, x))))

NoChangeDL(t: time, x1: Tracks): bool = (∀ x: (x = x1 ⇒ ¬ Grd1(t, x)))

NoChangeDirOp(t: time): bool =
    (∀ x: ((TRUE ⇒ ¬ Grd3(t, x)) ∧ (TRUE ⇒ ¬ Grd4(t, x))))

SomeGrd(t: time, x: Tracks): bool =
    (((Grd1(t, x) ∨ Grd2(t, x)) ∨ Grd3(t, x)) ∨ Grd4(t, x))

NoGrds(t: time): bool =
    (∀ x:(((¬ Grd1(t, x) ∧ ¬ Grd2(t, x))∧ ¬ Grd3(t, x)) ∧ ¬ Grd4(t,x)))

Init: AXIOM Init(0)

OpnNoG1: AXIOM
  (¬ Grd1(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ ¬ Grd1(t3, x))))))

OpnNoG2: AXIOM
  (¬ Grd2(t, x) ⇒
```

```
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ ¬ Grd2(t3, x))))))

OpnNoG3: AXIOM
  (¬ Grd3(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ ¬ Grd3(t3, x))))))

OpnNoG4: AXIOM
  (¬ Grd4(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ ¬ Grd4(t3, x))))))

PointWise1: AXIOM
  (Grd1(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ Grd1(t3, x)))))))

PointWise2: AXIOM
  (Grd2(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ Grd2(t3, x)))))))

PointWise3: AXIOM
  (Grd3(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ Grd3(t3, x)))))))

PointWise4: AXIOM
  (Grd4(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ Grd4(t3, x)))))))

NoGrdNoChangeNoDL: AXIOM
  (∀ t1:
     (t1 > t ⇒
       ((∀ t3: ((t ≤ t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsNoDL(t3, x)))) ⇒
         (∀ t3:
            ((t < t3 ∧ t3 ≤ t1) ⇒
              (∀ (x1: Tracks): NoDL(t, x1) = NoDL(t3, x1))))))))

NoGrdNoChangeDL: AXIOM
  (∀ t1:
     (t1 > t ⇒
       ((∀ t3: ((t ≤ t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDL(t3, x)))) ⇒
         (∀ t3:
            ((t < t3 ∧ t3 ≤ t1) ⇒
```

```
                (∀ (x1: Tracks): DL(t, x1) = DL(t3, x1)))))))

NoGrdNoChangeDirOp: AXIOM
  (∀ t1:
     (t1 > t ⇒
       ((∀ t3: ((t ≤ t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDirOp(t3, x)))) ⇒
         (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t) = DirOp(t3))))))

UpDateNoDL1: AXIOM
  (Grd1(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsNoDL(t3, x)))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ NoDL(t3, x) = FALSE))))))

UpDateNoDL2: AXIOM
  (Grd2(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsNoDL(t3, x)))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ NoDL(t3, x) = TRUE))))))

UpDateDL1: AXIOM
  (Grd1(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDL(t3, x)))) ⇒
           (∀ t3:
              ((t < t3 ∧ t3 ≤ t1) ⇒
                DL(t3, x) = (CT(t) + WT)))))))

UpDateDirOp1: AXIOM
  (Grd3(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDirOp(t3, x)))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t3) = FALSE))))))

UpDateDirOp2: AXIOM
  (Grd4(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDirOp(t3, x)))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t3) = TRUE))))))

NoUpDateNoDL1: AXIOM
  (NoGrds(t) ∨
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsNoDL(t3, x)))) ⇒
           (∀ (x1: Tracks):
              (NoChangeNoDL(t3, x1) ⇒
                (∀ t3:
                   ((t < t3 ∧ t3 ≤ t1) ⇒
```

```
                    (∀ (x1: Tracks):
                       NoDL(t, x1) = NoDL(t3, x1)))))))))))

NoUpDateNoDL2: AXIOM
  (NoGrds(t) ∨
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsNoDL(t3, x)))) ⇒
           (∀ (x1: Tracks):
              (NoChangeNoDL(t3, x1) ⇒
                (∀ t3:
                   ((t < t3 ∧ t3 ≤ t1) ⇒
                     (∀ (x1: Tracks):
                        NoDL(t, x1) = NoDL(t3, x1)))))))))))

NoUpDateDL1: AXIOM
  (NoGrds(t) ∨
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDL(t3, x)))) ⇒
           (∀ (x1: Tracks):
              (NoChangeDL(t3, x1) ⇒
                (∀ t3:
                   ((t < t3 ∧ t3 ≤ t1) ⇒
                     (∀ (x1: Tracks): DL(t, x1) = DL(t3, x1)))))))))))

NoUpDateDirOp1: AXIOM
  (NoGrds(t) ∨
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDirOp(t3, x)))) ⇒
           (NoChangeDirOp(t3) ⇒
             (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t) = DirOp(t3)))))))))

NoUpDateDirOp2: AXIOM
  (NoGrds(t) ∨
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ (∀ x: NoGrdsDirOp(t3, x)))) ⇒
           (NoChangeDirOp(t3) ⇒
             (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t) = DirOp(t3)))))))))

LeftOpnNoDL: AXIOM
  (t > 0 ⇒
    (∃ t1:
       (t1 < t ∧
         (∀ t3:
            ((t1 < t3 ∧ t3 < t) ⇒
              (∀ (x1: Tracks): NoDL(t, x1) = NoDL(t3, x1)))))))

LeftOpnDL: AXIOM
  (t > 0 ⇒
    (∃ t1:
       (t1 < t ∧
```

```
          (∀ t3:
             ((t1 < t3 ∧ t3 < t) ⇒
               (∀ (x1: Tracks): DL(t, x1) = DL(t3, x1)))))))

LeftOpnDirOp: AXIOM
  (t > 0 ⇒
    (∃ t1:
       (t1 < t ∧ (∀ t3: ((t1 < t3 ∧ t3 < t) ⇒ DirOp(t) = DirOp(t3))))))

NotIntG1: LEMMA
  (∀ t1: (t < t1 ⇒ ¬ ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ Grd1(t3, x))))))

NotIntG2: LEMMA
  (∀ t1: (t < t1 ⇒ ¬ ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ Grd2(t3, x))))))

NotIntG3: LEMMA
  (∀ t1: (t < t1 ⇒ ¬ ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ Grd3(t3, x))))))

NotIntG4: LEMMA
  (∀ t1: (t < t1 ⇒ ¬ ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ Grd4(t3, x))))))

OpnNoGrds: LEMMA
  (NoGrds(t) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧ (∀ t3: (Neigh(t1, t3, t2) ⇒ NoGrds(t3))))))

PointWiseNoGrds1: LEMMA
  (Grd1(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ NoGrds(t3)))))))

PointWiseNoGrds2: LEMMA
  (Grd2(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ NoGrds(t3)))))))

PointWiseNoGrds3: LEMMA
  (Grd3(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ NoGrds(t3)))))))

PointWiseNoGrds4: LEMMA
  (Grd4(t, x) ⇒
    (∃ t1, t2:
       (Neigh(t1, t, t2) ∧
         (∀ t3: (Neigh(t1, t3, t2) ⇒ (t = t3 ∨ ¬ NoGrds(t3)))))))

FirstChangeNoDL: LEMMA
  (∀ (x1: Tracks):
     ((t < t1 ∧ ¬ NoDL(t, x1) = NoDL(t1, x1)) ⇒
       (∃ t2:
```

```
          ((t ≤ t2 ∧ t2 < t1) ∧
            (ChangeNoDL(t2, x1) ∧
              (∀ t3:
                 ((t ≤ t3 ∧ t3 < t2) ⇒
                   (∀ x: (∀ x: NoGrdsNoDL(t3, x))))))))))
```

FirstChangeDL: LEMMA

```
  (∀ (x1: Tracks):
     ((t < t1 ∧ ¬ DL(t, x1) = DL(t1, x1)) ⇒
       (∃ t2:
          ((t ≤ t2 ∧ t2 < t1) ∧
            (ChangeDL(t2, x1) ∧
              (∀ t3:
                 ((t ≤ t3 ∧ t3 < t2) ⇒
                   (∀ x: (∀ x: NoGrdsDL(t3, x))))))))))
```

FirstChangeDirOp: LEMMA

```
  ((t < t1 ∧ ¬ DirOp(t) = DirOp(t1)) ⇒
    (∃ t2:
       ((t ≤ t2 ∧ t2 < t1) ∧
         (ChangeDirOp(t2) ∧
           (∀ t3:
              ((t ≤ t3 ∧ t3 < t2) ⇒
                (∀ x: (∀ x: NoGrdsDirOp(t3, x)))))))))
```

LastChangeNoDL: LEMMA

```
  (∀ (x1: Tracks):
     ((t < t1 ∧ ¬ NoDL(t, x1) = NoDL(t1, x1)) ⇒
       (∃ t2:
          ((t ≤ t2 ∧ t2 < t1) ∧
            (ChangeNoDL(t2, x1) ∧
              (∀ t3:
                 ((t2 < t3 ∧ t3 < t1) ⇒
                   (∀ x: (∀ x: NoGrdsNoDL(t3, x))))))))))
```

LastChangeDL: LEMMA

```
  (∀ (x1: Tracks):
     ((t < t1 ∧ ¬ DL(t, x1) = DL(t1, x1)) ⇒
       (∃ t2:
          ((t ≤ t2 ∧ t2 < t1) ∧
            (ChangeDL(t2, x1) ∧
              (∀ t3:
                 ((t2 < t3 ∧ t3 < t1) ⇒
                   (∀ x: (∀ x: NoGrdsDL(t3, x))))))))))
```

LastChangeDirOp: LEMMA

```
  ((t < t1 ∧ ¬ DirOp(t) = DirOp(t1)) ⇒
    (∃ t2:
       ((t ≤ t2 ∧ t2 < t1) ∧
         (ChangeDirOp(t2) ∧
           (∀ t3:
              ((t2 < t3 ∧ t3 < t1) ⇒
                (∀ x: (∀ x: NoGrdsDirOp(t3, x)))))))))
```

```
UpDateLocNoDL1: LEMMA (Grd1(t, x) ⇒ LimPlusNoDL(t, x, FALSE))

UpDateLocNoDL2: LEMMA (Grd2(t, x) ⇒ LimPlusNoDL(t, x, TRUE))

UpDateLocDL1: LEMMA (Grd1(t, x) ⇒ LimPlusDL(t, x, (CT(t) + WT)))

UpDateLocDirOp1: LEMMA (Grd3(t, x) ⇒ LimPlusDirOp(t, FALSE))

UpDateLocDirOp2: LEMMA (Grd4(t, x) ⇒ LimPlusDirOp(t, TRUE))

UpDateParNoDL1: LEMMA
  (Grd1(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ NoGrdsNoDL(t3, x))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ NoDL(t3, x) = FALSE))))))

UpDateParNoDL2: LEMMA
  (Grd2(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ NoGrdsNoDL(t3, x))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ NoDL(t3, x) = TRUE))))))

UpDateParDL1: LEMMA
  (Grd1(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ NoGrdsDL(t3, x))) ⇒
           (∀ t3:
              ((t < t3 ∧ t3 ≤ t1) ⇒
                DL(t3, x) = (CT(t) + WT)))))))

UpDateParDirOp1: LEMMA
  (Grd3(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ NoGrdsDirOp(t3, x))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t3) = FALSE))))))

UpDateParDirOp2: LEMMA
  (Grd4(t, x) ⇒
    (∀ t1:
       (t1 > t ⇒
         ((∀ t3: ((t < t3 ∧ t3 < t1) ⇒ NoGrdsDirOp(t3, x))) ⇒
           (∀ t3: ((t < t3 ∧ t3 ≤ t1) ⇒ DirOp(t3) = TRUE))))))
END Railroad
```

D.4 Theory: verif

```
verif: THEORY
 BEGIN

  IMPORTING Railroad

  t, t0, t1, t2, t3, τ: VAR time;

  x: VAR Tracks;

  Arrives(t, x): bool = Cmg(t, x) ∧ LimMinusCmg(t, x, FALSE)

  LemArr: LEMMA Arrives(t, x) ⇒ t > 0

  InCrArr: LEMMA
    InCr(t) ⇒
     (∃ t3:
        ∃ x:
          (t3 ≤ t − dmin ∧
            Arrives(t3, x) ∧ (∀ τ: ((t3 ≤ τ ∧ τ ≤ t) ⇒ Cmg(τ, x)))))

  ArrDL: LEMMA
    (Arrives(t, x) ∧ t < t3 ∧ (∀ τ: ((t ≤ τ ∧ τ ≤ t3) ⇒ Cmg(τ, x)))) ⇒
     (NoDL(t, x) ∧
       (∀ τ:
          ((t < τ ∧ τ ≤ t3) ⇒
            (¬ NoDL(τ, x) ∧ DL(τ, x) = t + WT))))

  NotSfDirCl: LEMMA
    (t < t3 ∧ (∀ τ: ((t ≤ τ ∧ τ ≤ t3) ⇒ SafeToOpen(τ) = FALSE))) ⇒
     (∀ τ: ((t < τ ∧ τ ≤ t3) ⇒ DirOp(τ) = FALSE))

  Safety: THEOREM InCr(t) = TRUE ⇒ GtClsd(t) = TRUE
 END verif
```

E PVS Proof of the Verification for the Generalized Railroad Crossing Problem

Here we give the strategy APPLY_LEMMA and the complete proof of (Safety) which consists of proofs of 3 lemmas: $InCrArr$, $ArrDL$, $NotSfDirCl$ and at last of that of (Safety).

E.1 A Strategy

```
(defstep  apply_lemma (lem args)
   (let ((x(cons 'inst(cons -1 args))))
   (then (lemma lem) x (split -1)))
" " "applying some lemma to some argument"
   )
```

E.2 Proof of Lemma InCrArr

```
;;; Proof for formula verif.InCrArr
;;; developed with old decision procedures
(""
 (SKOSIMP 1)
 (APPLY_LEMMA "CrCm" ("t!1"))
 (("1"
   (FLATTEN)
   (SKOSIMP -2)
   (APPLY_LEMMA "Coming" ("t!1-dmin" "x!1"))
   (("1"
     (SKOLEM -1 "t1")
     (INST 1 "t1" "x!1")
     (FLATTEN)
     (CASE "FORALL tau:(t1 <= tau AND tau <= t!1 - dmin)=>(Cmg(tau,x!1))")
     (("1"
       (CASE "FORALL tau: ((t1 <= tau AND tau <= t!1) => Cmg(tau, x!1))")
       (("1"
         (SPLIT 1)
         (("1" (PROPAX))
          ("2" (INST -1 "t1") (EXPAND "Arrives") (GRIND))
          ("3" (PROPAX))))
        ("2"
         (MERGE-FNUMS (-1 -7))
         (SKOSIMP 1)
         (APPLY_LEMMA "union_int"
          ("lambda (tau:time): Cmg(tau,x!1)" "t1" "t!1-dmin" "t!1"))
         (("1" (GRIND)) ("2" (GRIND)) ("3" (GROUND))))))
      ("2" (SKOSIMP 1) (INST -5 "tau!1") (GROUND))))
    ("2" (GRIND)) ("3" (GRIND))))
  ("2" (GROUND))))
```

E.3 Proof of Lemma ArrDL

```
;;; Proof for formula verif.ArrDL
;;; developed with new decision procedures
(""
 (SKOSIMP 1)
 (CASE "not NoDL(t!1,x!1)")
 (("1"
   (APPLY_LEMMA "LeftOpnNoDL" ("t!1"))
   (("1"
```

```
   (APPLY_LEMMA "LemArr" ("t!1" "x!1"))
   (EXPAND "Arrives")
   (FLATTEN)
   (EXPAND "LimMinusCmg")
   (SKOSIMP -2)
   (SKOSIMP -5)
   (NAME "maximum" "if t1!2> t1!1 then t1!2 else t1!1 endif")
   (APPLY_LEMMA "NotIntG2" ("maximum" "x!1" "t!1"))
   (("1" (SKOSIMP 1) (INST -6 "t3!2") (INST -9 "t3!2") (GRIND))
    ("2" (GRIND))))
  ("2" (GRIND))))
("2"
 (APPLY_LEMMA "UpDateLocNoDL1" ("t!1" "x!1"))
 (("1"
   (APPLY_LEMMA "UpDateLocDL1" ("t!1" "x!1"))
   (("1"
     (CASE "forall tau:((t!1<tau AND tau <t3!1)=>(NoGrdsNoDL(tau,x!1) AND NoGrdsDL(tau,x!1)))")
     (("1"
       (SPLIT 1)
       (("1" (PROPAX))
        ("2"
         (SKOSIMP 1)
         (SPLIT 1)
         (("1"
           (APPLY_LEMMA "UpDateParNoDL1" ("t!1" "x!1"))
           (("1"
             (INST -1 "t3!1")
             (SPLIT -1)
             (("1" (GRIND))
              ("2" (SKOSIMP 1) (INST -6 "t3!2") (GRIND))
              ("3" (GROUND))))
            ("2" (GRIND))))
          ("2"
           (APPLY_LEMMA "UpDateParDL1" ("t!1" "x!1"))
           (("1"
             (INST -1 "t3!1")
             (SPLIT -1)
             (("1" (INST -1 "tau!1") (GRIND))
              ("2" (SKOSIMP 1) (INST -5 "t3!2") (GRIND))
              ("3" (GROUND))))
            ("2" (GRIND))))))))
      ("2"
       (SKOSIMP 1)
       (INST-CP -8 "tau!1")
       (EXPAND "NoGrdsNoDL")
       (EXPAND "NoGrdsDL")
       (CASE "(NOT Grd1(tau!1, x!1) AND NOT Grd2(tau!1, x!1))")
       (("1" (GRIND))
        ("2"
         (SPLIT 1)
         (("1"
           (APPLY_LEMMA "LeftOpnNoDL" ("tau!1"))
           (("1"
             (SKOSIMP -1)
             (NAME "borne" "if t1!1>t!1 then t1!1 else t!1 endif")
             (APPLY_LEMMA "NotIntG1" ("borne" "x!1" "tau!1"))
             (("1"
               (SKOSIMP 1)
               (INST -5 "t3!2")
```

```
                (INST -14 "t3!2")
                (GRIND))
               ("2" (GRIND))))
             ("2" (GROUND))))
           ("2" (GRIND)))))))
      ("2" (GRIND))))
    ("2" (GRIND))))))
```

E.4 Proof of Lemma NotSfDirCl

```
;;; Proof for formula verif.NotSfDirCl
;;; developed with old decision procedures
(""
 (SKOSIMP*)
 (APPLY_LEMMA "LeftOpnDirOp" ("tau!1"))
 (("1"
   (SKOSIMP -1)
   (NAME "lim" "if t1!1>t!1 then t1!1 else t!1 endif")
   (NAME "lim1" "(lim+tau!1)/2")
   (CASE "DirOp(lim1)")
   (("1"
     (CASE "SafeToOpen(lim1)=FALSE")
     (("1"
       (EXPAND " SafeToOpen")
       (SKOSIMP 1)
       (APPLY_LEMMA "UpDateLocDirOp1" ("lim1" "x!1"))
       (("1"
         (EXPAND "LimPlusDirOp")
         (SKOSIMP -1)
         (NAME "sup" "if t1!2>tau!1 then tau!1 else t1!2 endif")
         (INST -3 "sup")
         (GRIND))
        ("2" (GRIND))))
      ("2" (INST -7 "lim1") (GRIND))))
    ("2" (GRIND))))
  ("2" (GROUND))))
```

E.5 Proof of Safety

```
;;; Proof for formula verif.Safety
;;; developed with old decision procedures
(""
 (APPLY_LEMMA "dIneq" NIL)
 (SKOSIMP*)
 (APPLY_LEMMA "InCrArr" ("t!1"))
 (("1"
   (APPLY_LEMMA "ClsClsd"("t!1"))
   (("1" (GRIND))
    ("2"
     (SKOSIMP*)
     (APPLY_LEMMA "ArrDL" ("t3!1" "t!1" "x!1"))
     (("1"
       (LEMMA "NotSfDirCl")
       (INST -1 "t!1-dclose" "t!1")
       (("1"
         (CASE "(FORALL tau:((t!1 - dclose <= tau AND tau <= t!1) =>
                    SafeToOpen(tau) = FALSE))")
         (("1" (GRIND))
          ("2"
```

```
          (EXPAND "SafeToOpen")
          (SKOSIMP 1)
          (INST -3 "x!1")
          (INST -6 "tau!2")
          (INST -11 "tau!2")
          (GRIND))))
       ("2" (ASSERT))))
     ("2" (GROUND))))))
("2" (GROUND))))
```