Automated Deductive Requirements Analysis of Critical Systems

Angelo Gargantini
and
Angelo Morzenti
Politecnico di Milano, Italy

We advocate the need for automated support to System Requirement Analysis in the development of time- and safety-critical computer based systems. To this end we pursue an approach based on deductive analysis: high-level, real-world entities and notions, such as events, states, finite variability, cause-effect relations, are modeled through the temporal logic TRIO, and the resulting deductive system is implemented by means of the theorem prover PVS. Throughout the paper, the constructs and features of the deductive system are illustrated and validated by applying them to the well-known example of the Generalized Railway Crossing.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based systems**]: Real-time and embedded systems; D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies, tools*; D.2.4 [**Software Engineering**]: Software / Program Verification—*Formal methods*; I.6.4 [**Computing Methodologies**]: Model Validation and Analysis

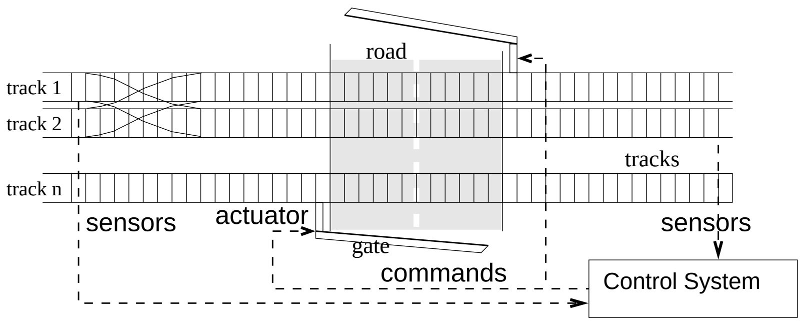
1. INTRODUCTION

Computer based time- and safety-critical systems are acquiring increasing social, economic, and environmental importance. Their intended purpose and properties must be clearly understood, explicitly and unambiguously stated, and formally verified, or even certified. This is particularly true for properties related to safety in fields such as transportation, where recent regulations require that the system certification be performed by a third-party authority, distinct from both the system constructor and the user or the transportation service provider.

Computer-based critical systems, however, are not simple devices confined inside

Work carried out under the financial support of the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications"
Address: Dipartimento di Elettronica e Informazione - Politecnico di Milano - Piazza Leonardo da Vinci 32,20133 Milano, Italy; e-mail: Gargantini@Elet.PoliMi.it or A.Gargantini@UniCt.it, and Morzenti@Elet.PoliMi.it

a box, just performing some numeric or symbolic computation: they are usually embedded systems in charge of supervision, signaling, or control tasks. To reach their intended purpose they must interact with their environment by means of sensors and actuators, trying to maintain the ambiance, or the part of it that is under their control, in a consistent and safe state. They typically reach this goal by reacting to stimuli, computing and updating internal state variables, acting on physical entities through their actuators. For instance a control system for a Railway Crossing is composed of a computer that interacts with the environment (the rail yard, the crossing street, the bars, etc.) through sensors on the tracks, and actuators on the bar.



High level design of the computerized component of a critical system (hereinafter called the device under construction, DUC) is not performed in a vacuum but strongly depends on models and assumptions regarding, besides the DUC itself, the environment, the sensors and the actuators.

Therefore writing the design specifications, from which the development of the DUC can start, is not the first activity of the development process, but must be the result of a preliminary phase whose purpose is to state, analyze and prove the user requirements (typically stated very abstractly in terms, e.g., of some safety or utility property) by modeling the system in its entirety, including the DUC and all the other components. In a very abstract logical setting, this preliminary activity consists of analyzing, validating, and eventually proving the following implication [Zave and Jackson 1997]:

$$EnvModel \land SensorModel \land ActuatorModel \land DesignSpec \rightarrow UserReq \qquad (1)$$

i.e., the user requirements are ensured if the DUC is constructed according to its specification, and placed, together with correctly functioning sensors and actuators, inside an environment that satisfies the original assumptions on the behavior of external entities.

This analysis is often considered trivial and performed, if at all, only informally. For real-world critical systems, however, the formalization of some of the components of the above implication (especially the user requirements, the environment model, and the design specification) may hide subtle difficulties, hence the necessary analysis requires a substantial effort, and can be complex and error prone.

Thanks to decades of research, the design and development of computer-based artifacts (e.g., programs, dedicated hardware) starting from well written, detailed, possibly formal design specifications is becoming increasingly systematic and reliable, being supported by suitable methods and tools. It is widely recognized [Leveson 1995] that for critical systems the great majority of catastrophic failures during operation are traced back to poor (incomplete, inconsistent, or simply wrong) design specifications. It is also widely acknowledged that the cost of correcting specification errors that are unveiled in the final testing phases or even at operation time, is order of magnitudes higher than that of fixing design or coding errors.

For this reason, relevant research efforts are being devoted to the above described preliminary analysis, which is often called System Requirements Analysis (SRA) [Komoda et al. 1981; Dutertre and Stavridou 1997] to emphasize that requirements (and hence specifications) are analyzed and validated, and that not only the DUC is analyzed, but also the other system components and the environment as well. SRA is a typical cooperative, interdisciplinary work, since it is performed jointly by experts of the application domain, users, representatives of regulating agencies, and computer engineers.

The above described purpose and features of SRA impose several, sometimes conflicting requirements on the modeling notation and on the methods and tools adopted in the analysis.

The modeling and analysis activities must be automated, to provide a support in dealing with complexity, to obtain mechanized checks for correctness, completeness, and consistency, and to certify the obtained results. This in turn requires the adoption of a formal notation, which also ensures absence of ambiguity, thus preventing misinterpretation among people, participating to SRA, who often have quite heterogeneous cultural backgrounds. To facilitate communication, discussion, and mutual understanding, the formal notation must be flexible, expressive, and high level. It must be able to model in a natural way real-world entities, basic notions such as events, actions, states (i.e., properties or values of system components, possibly having non-null duration), continuity or finite variability, (non)determinism, and cause-effect relations.

Operational notations (such as state-transition systems, Petri nets, ...) are very attractive as modeling languages: they include or easily express most of the above mentioned intuitive, real-world notions. However, SRA requires not only modeling capability, but also the ability to express requirements (i.e., desired properties) and relations among them (necessity, compatibility, mutual exclusion, ...). On the other hand pure descriptive notations, like first- or higher-order logic, are too low level if considered in isolation, as they do not encompass exactly those notions that, as noted above, are incorporated into most operational notations.

A special attention must be devoted, by notations and methods supporting SRA, to the representation of time: the notation must be rich and flexible enough to model both computer components, which are typically digital, synchronous, and clock based, and other non-digital components, which, often consisting of electrical, chemical, or mechanical processes, are time-continuous. Notice that many program or temporal logics originally devised to model execution on digital computers, are unable to describe continuous phenomena: their assumption of a discrete time and of a stepwise execution of state transitions, leads to models based on finite or denumerable infinite state sequences. Discrete state sequences model adequately program execution but are unable to describe truly asynchronous systems, where the distance in time between related events does not have a lower bound. Furthermore, since timing aspects are quite relevant for all critical systems, time should not be treated as just one more system component or (state) variable, but it should deserve a special treatment. Time modalities are very deeply embedded in human temporal representation and reasoning: think for instance of a phrase like: "If the train enters the safety region then, when the train will cross the road, the bar will have been completely lowered since at least 30 seconds". It is therefore apparent that the adoption of intuitive constructs supporting modeling and analysis of time-related system features can significantly facilitate understanding and communication in the framework of SRA.

In the present work we propose a framework for SRA where the system is modeled by means of a temporal logic language enriched with constructs representing high level intuitive notions (such as events, states, continuity, finite variability, cause-effect relation, etc.) formalized through logic entities (predicates, variables, functions) and suitable axioms. Analysis and proof of properties are carried out by means of deductive reasoning in a very uniform, systematic manner, based on simple propositional reasoning (e.g. on Case Analysis), first order generalization, induction. Thanks to the ability of the logic to represent real-world entities, derivations and their results -theorems, rejected conjectures, (counter)examples- are amenable to any engineer and easily translatable into natural language, to favor understanding by people lacking a mathematical background.

We choose TRIO [Ghezzi et al. 1990] as the modeling logic. TRIO is a typed, linear, metric temporal logic: the presence of types simplifies formulas and makes them more readable; linearity of the underlying time model makes the logic amenable also to engineers who are unfamiliar with Computer Science logics; the possibility to express quantitative time constraints allows specifiers to express precisely real-time requirements. TRIO has been adopted in a variety of industrial projects [Gargantini et al. 1996; Basso et al. 1998; Capobianchi et al. 1999; Ciapessoni et al. 1999], thus demonstrating its applicability to real-life industrial applications. TRIO is also embedded with object-oriented constructs for structuring specifications [Morzenti and San Pietro 1994] (not exploited in the present work), and is provided with a tool suite for analysis and verification based on simulation [Felder and Morzenti 1994] and testing [Mandrioli et al. 1995] that can be nicely integrated with the deductive approach described in the present work.

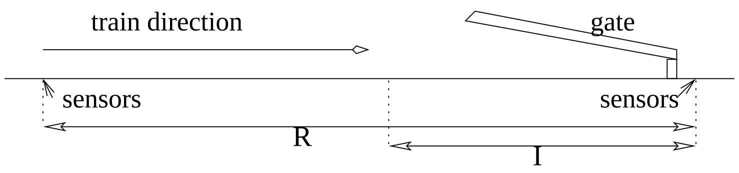
Deductive analysis is automated by using the PVS [Owre et al. 1995] theorem prover as a deductive computational engine: TRIO axioms stating properties and modeling the behavior of system components are translated into an internal representation in the higher order logic of PVS, and derivations are performed by means of PVS proof strategies and decision procedures.

Of course the results presented here are not restricted to the pair TRIO/PVS: any sufficiently expressive temporal logic and powerful theorem prover could be exploited in a similar setting to perform SRA.

The present paper is structured as follows. The remainder of this introduction provides basic context information concerning the case study we adopt as a running example (Section 1.1), and the temporal logic language TRIO (Section 1.2). Section 2 shows how real-world notions such as events, states, continuity, and finite variability can be modeled in TRIO by means of suitable specification items (predicates, variables ...) and axioms. Section 3 introduces typical ways of modeling temporal and causal relations among entities. Section 4 describes a tool supporting SRA, covering the encoding of TRIO in the logic of PVS, the proof strategies, the pretty-printer (useful for hiding the details of the encoding in the visualization of TRIO formulas), and illustrates its application to the proof of two relevant properties in the case study. Section 5 contains a brief review of related literature. Section 6 draws conclusions, and relates on possible applications of the proposed approach and on future research.

1.1 Our case study: General Railroad Crossing problem

To illustrate our method and to demonstrate its ability to deal with real world cases, we use the general railroad crossing (GRC) problem as a running example for almost every concept and construct hereafter introduced. The GRC problem was originally proposed in [Heitmeyer et al. 1993] and since then it has been used as a benchmark for a vast number of real-time languages. Furthermore it was recently used as a case study for comparing methods and tools for the analysis of critical systems [Heitmeyer and Mandrioli 1996]. The problem is here briefly reported. GRC describes a rail road crossing, i.e. an intersection between a road and several train tracks with a gate to prevent crossing during train passage. For sake of simplicity we assume that every train travels in the same direction. Two regions R and I, surrounding the crossing, are defined as depicted below:



Trains enter region R, then enter the critical region I and finally leave the area. Trains entering and leaving region R are detected by means of sensors placed on the track. Notice that several trains, up to the number of tracks, can simultaneously cross the region borders. Trains take a minimum time d_m and a maximum time d_M to go from the beginning of R to the beginning of I, and a minimum time h_m and a maximum time h_M to go from the beginning of region I to its end (thus exiting also the region of interest for the GRC). The system must ensure that the bar is closed when a train is in region I (*safety* property), but, to avoid needless blocks on the road, it must also ensure that the bar is down only when strictly necessary (*utility* property). The controller operates the bar by means of the following two commands *up* and *down*; the bar current position or state of motion is one of: closed, open, moving up (when opening), and moving down (when closing). The bar movement can be reversed by means of a command, i.e when it is moving up (down) and it receives a *down* (*up*) command it will start moving down (up). The angle of the bar increases (when the bar is moving up) and decreases (when the bar is moving down) with a constant speed. It takes the bar γ time units to reach the closed (respectively, open) position starting from an open (respectively, closed) state.

The controller adopts the following policy: starting from data coming from sen-

$Futr(F,\ d)$	$d \geq 0 \wedge Dist(F, d)$	future
$Past(F,\ d)$	$d \geq 0 \wedge Dist(F, -d)$	past
$Lasts(F,\ d)$	$\forall d'(0 < d' < d \rightarrow Futr(F, d'))$	F holds over a period of length d
$Lasted(F,\ d)$	$\forall d'(0 < d' < d \rightarrow Past(F, d'))$	F held over a period of length d
$Alw(F)$	$\forall d Dist(F, d)$	F always holds
$AlwF(F)$	$\forall d(d > 0 \rightarrow Futr(F, d))$	F will always hold in the future
$AlwP(F)$	$\forall d(d > 0 \rightarrow Past(F, d))$	F held always in the past
$Until(A_1, A_2)$	$\exists t\,(t > 0 \wedge Futr(A_2, t) \wedge Lasts(A_1, t))$	A_1 holds until A_2 becomes true
$Som(F)$	$\exists d Dist(F, d)$	Sometimes F held or will hold
$SomP(F)$	$\exists d(d < 0 \wedge Dist(F, d))$	F held sometimes in the past
$UpToNow(F)$	$\exists d\,(d > 0 \wedge Lasted(F, d))$	F held for an interval before now
$NowOn(F)$	$\exists d\,(d > 0 \wedge Lasts(F, d))$	F holds for an interval after now
$Becomes(F)$	$UpToNow(\neg F) \wedge (F \vee NowOn(F))$	F has not held before now, but it holds now (or now on) [1]
$LastTime(F,t)$	$Past(F, t) \wedge Lasted(\neg F, t)$	F occurred for the last time t units ago
$NextTime(F,t)$	$Futr(F, t) \wedge Lasts(\neg F, t)$	F will occur the first time at t units

Table 1. Derived temporal operators

sors it computes the number of trains that might possibly be in the region I. Whenever this number becomes positive, the controller issues a command to lower the bar, while whenever the number becomes null it issues a command to raise the bar. When lowering the bar, the controller takes into account the delay required by the bar to reach the closed position, and it sends the down command γ time units before the earliest expected time for train entrance in region I (no similar adjustment is necessary with reference to train exit from region I).

1.2 TRIO: a short language overview

TRIO is a first order logic augmented with temporal operators that allow one to express properties whose truth value may change over time. The meaning of a TRIO formula is not absolute, but is given with respect to a current time instant which is left implicit. The basic temporal operator is called *Dist*: for a given formula W, $Dist(W,\ t)$ means that W is true at a time instant whose distance is exactly t time units from the current instant, i.e., the instant when the sentence is claimed. Many other temporal operators can be derived from *Dist,* as shown in Table 1.

Notice that operators expressing a duration over a time interval (for example *Lasts*), do not specify the value of their argument outside the interval. Furthermore, note that for this kind of operators we gave definitions where the extremes of the specified time interval are excluded, i.e. the interval is open. Operators including either one or both of the extremes can be easily derived from the basic ones. For notational convenience, we indicate inclusion or exclusion of extremes of the interval by appending to the operator's name suitable subscripts, i or e, respectively. A few examples regarding the operators *Lasts*, *Lasted*, *AlwF* and *SomP* follow.

$$\begin{array}{ll} Lasts_{ie}(F, d) & \forall d'(0 \leq d' < d \rightarrow Dist(F, d')) \\ Lasted_{ii}(F, d) & \forall d'(0 \leq d' \leq d \rightarrow Dist(F, -d')) \\ AlwF_i(F) & \forall d(d \geq 0 \rightarrow Dist(F, d)) \\ SomP_i(F) & \exists d(d \leq 0 \wedge Dist(F, d)) \end{array}$$

[1]The definition of *Becomes* is slightly different from that provided in other previous works.

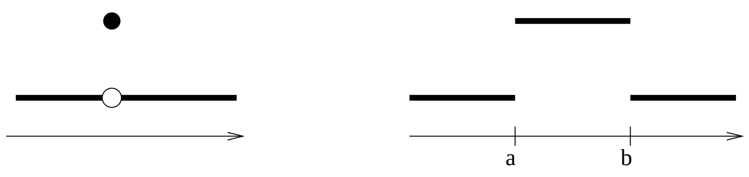


Fig. 1. Event and interval predicate

TRIO introduces time dependent (TD) predicates, possibly associated to a distinct relation for every time instant, and TD variables in a domain D, as variables whose value changes in D over time. For instance, TD variables are suitable to model enumerative values and continuously changing values, as many physical quantities. To refer to values of a variable or term in the past or in the future, the operator *dist* (as generalization of *Dist*) is introduced: for a given term x, *dist*(x,t) has the value that x had or will have at a time instant whose distance is t from now. The operators *futr* and *past,* derived from *dist,* refer to values of a TD variable respectively in the future and in the past.

2. FUNDAMENTAL TIME RELATED ENTITIES

In this section we introduce the notion of event (as a predicate that holds at isolated time points), the notion of interval predicate (a predicate that holds, or does not hold, for non-empty temporal intervals), and we define the notion of non-Zeno predicate (a predicate whose truth value does not change "infinitely often"), with some illustrative examples. Then we present some interesting properties of such types of predicates. Finally we generalize these definitions to formulas and to time dependent variables ranging on countable or uncountable domains. All the notions presented are formally defined in terms of TRIO axioms that are intended as always valid, hence implicitly enclosed in an outermost *Alw* operator.

2.1 Point-based predicates

We call *point-based* or *events* the predicates that hold in isolated time points, and are false elsewhere. Their behavior is visualized in Figure 1. Their formal definition in terms of TRIO is as follows.

Definition 1. **point-based predicate***: a TD predicate E is called point-based or event iff:*

$$E \to UpToNow(\neg E) \wedge NowOn(\neg E)$$

A point-based predicate has therefore a null duration. This definition is clearly an abstraction, since in nature no event has a null duration. This kind of abstraction is very common among formal methods modeling real time systems, as null duration events are suitable to formalize natural events, whose duration is small with respect to the reaction times of the system. Allowing events with null duration could introduce inconsistencies: for example, in presence of circular dependencies one could have an unlimited number of occurrences of the same event with no time progression. This problem can be avoided by introducing suitable properties and definitions as shown in Section 2.3. Some formal languages solve the problem by banning events having null duration. See [Gargantini et al. 1999] for a survey of problems and proposed solutions and for a general solution using non standard analysis.

Examples of point based predicates might be boolean messages between processes, methods invocations, external inputs. For instance *pure signals* in SIGNAL [Benveniste et al. 1991] might be modeled as events.

Example 1. In our GRC case study we use events to represent commands given to the bar: *up* and *down*. Informally, when the bar receives an *up* command and it is closed or it is moving down and closing, it starts opening. If it is opening or open and it receives a *down* command, it starts closing. We think of these commands as control pulses having a very small duration, which we consider null. ♢

2.2 Interval-based predicates

As opposed to point based predicates, we now consider predicates that keep their value for entire time intervals. We call these *interval-based* predicates or simply *interval* predicates. Informally, interval predicates hold true or false for intervals with non-null duration, thus they are never true or false in isolated time points. A diagrammatic representation of interval behavior is pictured in Figure 1.

From the intuitive definition we can define an interval predicate as follows:

Definition 2. **interval-based predicate:** *a TD predicate I is interval-based iff:*

$$(I \rightarrow (UpToNow(I) \vee NowOn(I))) \wedge (\neg I \rightarrow (UpToNow(\neg I) \vee NowOn(\neg I)))$$

The meaning is exactly that an interval predicate cannot keep its value in isolated time points, so if it is true (respectively, false) at a time point then there is an interval, following or preceding it, where it is true (respectively, false).

Notice that this definition does not tell anything about the value of I at the precise instants when I changes its value from true to false or vice versa (points a and b in Figure 1). Thus time intervals where the predicate keeps its value, might be open or closed (i.e. they may or may not contain their end-points). Several choices can be made about the predicate value at such instants, but there are two main possible behaviors (shown in Figure 2): we say that a predicate is left (right) continuous, if it has the value it has had in the immediate previous (next) neighborhood. In TRIO this can be formalized as follows:

Definition 3. **left continuous interval based predicate:**

$$(I \rightarrow UpToNow(I)) \wedge (\neg I \rightarrow UpToNow(\neg I))$$

right continuous interval based predicate:

$$(I \rightarrow NowOn(I)) \wedge (\neg I \rightarrow NowOn(\neg I))$$

Some "philosophical" arguments would support the choice of left continuity, others would favor right continuity: a thorough discussion is reported in [Gargantini and Morzenti 1999]. At this point one choice is worth the other one. For the time being we do not explicitly choose a type of behavior against the others: we will however return to this issue in Section 2.6 and in Section 2.10, when we will make a choice.

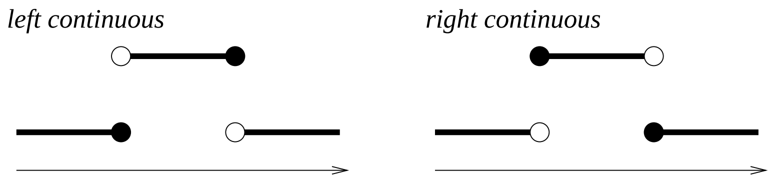


Fig. 2. Left and right continuous interval predicates

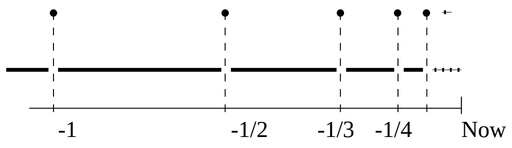
2.3 Non-Zeno requirement for predicates

In this Section we define the *non-Zeno* or *finite variability* requirement [Abadi and Lamport 1994] for a TD predicate, namely, that a predicate can only change its value a finite number of times in a finite time interval. Since we did not establish an a priori lower bound either on the duration of any interval predicate or on the distance between two occurrences of any event, a predicate may have a Zeno behavior, changing value an unbounded number of times in a finite interval. Only non-Zeno behaviors are physically meaningful. For this reason the non-Zeno requirement has to be explicitly introduced, otherwise, being the time model continuous, specifications would be exposed to incompleteness and even simplest intuitive properties would be false, as shown by the following example.

Example 2. **a simple Zeno event**: consider the event E, defined by the following formula, where t is a real, and n is a natural number:

$$\forall t \left(Past(E,t) \leftrightarrow \exists n \left(t = \frac{1}{n} \right) \right)$$

E occurs only in the past at a distance, from the current time, of 1, $\frac{1}{2}$, $\frac{1}{3}$, $\frac{1}{4}$, ... time units. Moreover E does not occur at the present time. Its behavior is pictured below:



E is a Zeno event, as it occurs infinite times near the current instant now. Predicate E does not satisfy the following property:

$$\neg E \ \rightarrow \ \exists\varepsilon \ Lasted(\neg E, \varepsilon)$$

having the following meaning, quite intuitive for a predicate E modeling the notion of event: if E is false now then there is a left neighborhood of now where E is false; otherwise E would occur an infinite number of times immediately before now. $\diamondsuit$

An informal definition of the non-Zeno requirement for a predicate A is that there exists a time interval (arbitrarily small) where A is constantly true or it is constantly false. Formally we suggest this definition:

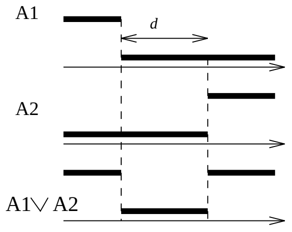
Definition 4. **non-Zeno requirement for TRIO predicate:** *a TD predicate A is non-Zeno iff:*

$$(UpToNow(\neg A) \vee UpToNow(A)) \wedge (NowOn(\neg A) \vee NowOn(A))$$

The first conjunct (with *UpToNow*) guarantees that there is no accumulation point of changing instants before the current instant, whereas the second part (with *NowOn*) guarantees the same fact after the current time. For a non-Zeno predicate it is therefore meaningful to use locutions such as "The value of predicate P immediately before (or after) the current time".

The predicate E in the previous example (example 2) does not satisfy the non-Zeno requirement. In the current instant neither $\mathit{UpToNow}(\neg E)$ nor $\mathit{UpToNow}(E)$ hold, because there is no ε such that $\mathit{Lasted}(E,\varepsilon)$ or $\mathit{Lasted}(\neg E,\varepsilon)$.

Note 1. The non-Zeno requirement does not imply that there is a *minimum* time distance between any two changes of value of a predicate, but that *there is such a distance.* Consider for example, two predicates A_1 and A_2, whose behavior is depicted in the following figure.



If they are independent (for example two inputs in an asynchronous system) the distance d between a change of A_1 and the change of A_2 might be arbitrarily small. If this distance had a lower bound then Zeno behavior would not be possible and we might as well adopt a model based on discrete time (like the integers). This is the case of synchronous, clock based systems, where every input arrives at time instants that are multiple of some quantity (the period of the clock). Indeed some formal languages follow this approach and enforce a fixed minimum delay between two actions. However, for asynchronous systems, there is no lower bound for quantity d, time must be considered continuous, and Zeno behaviors must be taken into account and explicitly excluded. Section 5 reports on related works adopting solutions similar to ours.

2.4 Non-Zeno point and interval predicates

The proposed definitions of point and interval predicates, of left- or right-continuous predicate, and of non-Zeno predicate are independent; they can be combined (as pictured in Figure 3) to obtain formal definitions of non-Zeno events and non-Zeno interval predicates (formal propositions are reported in Table 4 and proofs of equivalence are in [Gargantini and Morzenti 1999]). These are the predicates of practical interest. The events *up* and *down* introduced in the Example 1 are more precisely non-Zeno events.

2.5 Generalization to TRIO formulas

The definitions presented in the previous sections with reference to predicates can be generalized to formulas by just replacing in each definition the predicate with an entire TRIO formula. For example, the definition of a non-Zeno event formula becomes:

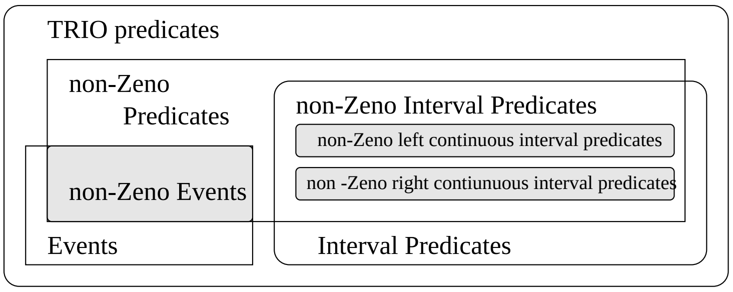

TRIO predicates
non-Zeno Predicates
non-Zeno Interval Predicates
non-Zeno left continuous interval predicates
non -Zeno right contiunuous interval predicates
non-Zeno Events
Events
Interval Predicates

Fig. 3. A subset system for TRIO Formula

$\wedge$	f	i	lci	e
e	$\boxed{e}$	e	e	e
lci	f	f	lci	
i	f	$\boxed{f}$		
f	f			

$\neg$	
e	f
lci	lci
i	i
f	f

$\vee$	f	i	lci	e
e	f	f	i	e
lci	f	f	lci	
i	f	$\boxed{f}$		
f	f			

Table 2. Closure properties of logical operators

Definition 1. **non-Zeno event formula** : *a TD formula F is a non-Zeno event if and only if:*

$$UpToNow(\neg F) \wedge NowOn(\neg F)$$

Similar trivial adaptations can be devised for all the other definitions and theorems.

2.6 Fundamental properties and operators

Based on these definitions, we can establish interesting properties about formulas obtained from the application of logical and temporal operators to the above defined entities. The result of the application of the propositional operators $\wedge$, $\neg$, and $\vee$ to operands of the various types is summarized by Table 2, where f stands for a generic formula, i for interval formula, lci for left continuous interval formula, and e is for event. All the reported properties have been proven with the automatic support of the TRIO theorem prover built on top of PVS.

Note 2. *Remarks on particular results of logical operations (cases enclosed in boxes in Table 2).* If E is an event, and F a generic formula, $E \wedge F$ is still an event, hence conditioned events (as defined in SCR [Heitmeyer et al. 1996]) are still events. If I_1 and I_2 are interval formulas, then neither $I_1 \vee I_2$ nor $I_1 \wedge I_2$ are necessarily interval formulas, i.e., the class of interval formulas is not closed with respect to logic conjunction nor disjunction. This fact is very unfortunate because it makes almost useless this type of formula (and it jeopardizes all the arguments given about the necessity of using interval formulas) (see also [Chaochen and Hansen 1997]). Fortunately the class of left (as well as right) continuous interval formulas is closed with respect to every operation. These closure properties provide a strong motivation for adopting, in system modeling, a definite and uniform notion of continuity for interval predicates. This approach is also adopted in other related proposals. We will return to this subject again in Section 2.10.

F is of type ...	e	lci
$Dist(F,d)$	e	lci
$UpToNow(F)$		lci
$Lasted_{ie}(F,d)$	lci	lci
$SomP_e(F)$	lci	lci
$AlwP_e(F)$	lci	lci
$WithinP_{ie}(F)$	lci	lci

F is of type ...	e	lci
$Becomes(F)$	e	$\boxed{e}$
$NowOn(F)$		lci
$Lasts_{ie}(F,d)$	lci	lci
$SomF_i(F)$	lci	lci
$AlwF_i(F)$	lci	lci
$WithinF_{ie}(F)$	lci	lci

Table 3. Temporal operators

Temporal operators

We have also proven (with the automated support of PVS) several interesting closure properties regarding temporal operators. Results are shown in Table 3. Note that the basic TRIO operator *Dist* does not change the type of its argument: this is easily understood by considering that *Dist* just performs a temporal shift on the time axis. The operator *Becomes* always returns an event.

The above properties can be used to determine, in a systematic and reliable way, the type of a compound TRIO formula, from that of its basic components, thus avoiding direct formal proofs based on the definition of the various types of entity. For instance if F and G are generic TD predicates, we can immediately be sure that the formula $Becomes(F) \wedge G$ is an event.

2.7 Generalization to time dependent variables

Most concepts and definitions given so far for TRIO predicates and formulas can be extended to TRIO time dependent (TD) variables. Indeed TD variables, as seen for predicates, might in principle behave in a bizarre way, unlike any possible real behavior. In this section we introduce some definitions providing constraints on time dependent variables, useful for modeling real-world systems. These definitions are the extension or generalization to variables of those given for predicates (in fact, predicates might as well be considered as boolean variables: in this case the following definitions for variables are equivalent to those given for predicates; this analogy is fully exploited in higher-order logic, as we will briefly discuss in Section 4.2).

2.7.1 *Point based variables and simultaneous events.* Let us consider a system variable that keeps a default value (for example null) at all times except for single instants, where it has values in a given domain. To model this kind of variable we introduce point based variables, defined as follows:

Definition 5. **point variable:** *a TD variable* v *in a domain D is a point variable with default value $d \in D$ if and only if*:

$$v \neq d \rightarrow UpToNow(v = d) \wedge NowOn(v = d)$$

Point variables might model data flows of LUSTRE [Halbwachs et al. 1992] and SIGNAL [Benveniste et al. 1991]. In this language the default value is called *absence* and denoted by $\perp$.

Furthermore point based variables with default value 0 can model events with possibly multiple simultaneous occurrences; notice that this cannot be done by event predicates, because at a given instant a (time dependent) predicate can model only whether an event occurs or not, not how many times. We will return to this subject in Example 4.

2.7.2 *Interval variables and counters.* Informally interval variables are piecewise constant variables, i.e., variables that keep their value for non-empty time intervals, and do not hold a value in isolated time points.

Definition 6. **interval variable:** *a TD variable* v *in a domain D is an interval variable iff, for every value $a \in D$*:

$$v = a \rightarrow (UpToNow(v = a) \vee NowOn(v = a))$$

Event counters, i.e., integer variables having as value the number of occurrences of a given event, are a particular kind of interval variable. Given an event E we denote its counter as $\#E$. Note that E might be an event with simultaneous occurrences. The definition of event counters will be given in Section 3.3.

Continuity. For interval variables definitions of continuity can be given in a way similar to that seen for predicates.

Definition 7. **left (right) continuity**: *a TD variable* v *in a domain D is a left (right) continuous interval variable iff, for every $a \in D$*:

$$\begin{array}{c} v = a \ \rightarrow \ UpToNow(v = a) \\ (v = a \ \rightarrow \ NowOn(v = a)) \end{array}$$

2.8 Non-Zeno requirement for variables

We now define the non-Zeno requirement, or finite variability requirement, for a time dependent variable.

First, we consider variables in a countable domain (i.e., a domain that is either finite or equal in cardinality to the set N of the naturals: it could be, for instance, the set of the integer or rational numbers, or any subset thereof). In this case we simply require that the variable changes its value a finite number of times in every finite time interval, so that every finite interval can be split into a finite number of intervals where the variable is constant. Therefore, at any time there are two (arbitrarily small) left and right time intervals where the variable is constant:

Definition 8. **non-Zeno variable in a countable domain:** *a TD variable* v *in a countable domain D is non-Zeno iff:*

$$\exists a \in D \; UpToNow(v = a) \wedge \exists b \in D \; NowOn(v = b)$$

Notice that we do not impose any requirement on the variable at single time points, where it could have any value.

Variables with an uncountable domain. Next, we consider variables on uncountable domains, like for instance the reals or any interval of reals. This is the most general case, and also quite frequent in practice: real-time systems are often hybrid systems involving both real-valued physical variables and digital components.

The definition previously provided for countable domains, which requires a variable to be piecewise constant, cannot be extended to uncountable domains, because real-valued quantities might as well change continuously, thus assuming an infinite

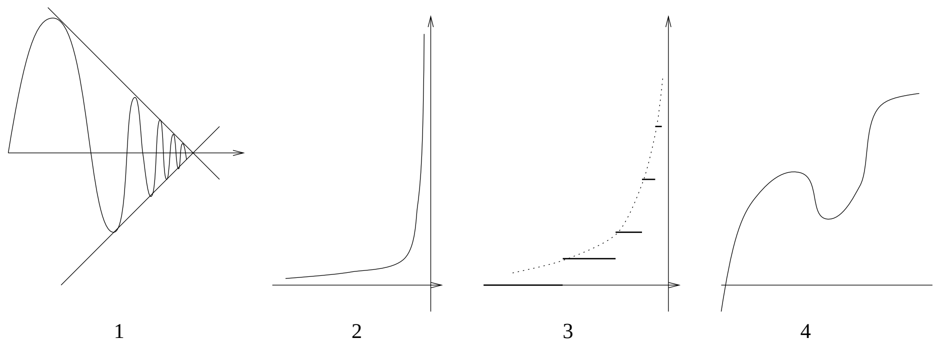


Fig. 4. Examples of Zeno and non-Zeno behaviors

number of values in a finite time interval: consider for instance a sinusoid or a ramp.

However we cannot accept every possible behavior for real valued variable, because they typically represent real-world entities that are subject to physical laws. Furthermore, we expect real-valued variables satisfying our non-Zenoness requirement to give rise to non-Zeno formulas when composed via arithmetic and relational operators.

Informally, we require a non-Zeno variable v in an uncountable domain D to be piecewise analytic when considered as a function of time[2].

More formally, we define a variable v with values in a domain D to be non-Zeno if, at every time, there exist two functions f and g: $\Re \rightarrow D$ that are analytic at 0, and such that v is equal to f in a right interval and to g in a left interval of the current time; if we denote as AF_o the set of functions that are analytic at 0, the non-Zeno requirement can be formalized as follows.

Definition 9. **non-Zeno variable**: *a TD variable* v *in an uncountable domain is non-Zeno iff:*

$$\exists f \exists g\, (f, g \in AF_o \wedge \exists d \forall t\, (0 < t < d \rightarrow futr(v,t) = f(t) \wedge past(v,t) = g(t)))$$

Notice that if the variable domain is countable this definition reduces to definition 8. Indeed, the only analytic functions with value in a countable domain are the constant functions. Then the variable must be piecewise constant (see [Gargantini and Morzenti 1999]).

Definition 9 provides us with a simple criterion to determine whether a variable is non-Zeno. Let us for instance apply this criterion to a few particular, yet very common cases. A constant variable is certainly non-Zeno, as well as a variable with polynomial behavior, harmonic functions (*sin* and *cos*) and exponential functions. Piecewise constant or linear or harmonic variables are non-Zeno. Sum, difference and product of two non-Zeno variables are non-Zeno; the division is non-Zeno if the denominator is always different from zero. Variables with constant or bound derivatives (as in [Manna and Sipma 1998]) are non Zeno.

[2]In extreme summary, an analytic function is very regular: it can be expanded in a power series, for example a Taylor series. See [Courant and Fritz 1974] for a deeper insight.

Not all the variables, however, are non-Zeno. Instances of Zeno variable (expressed as functions of time) are: $\frac{1}{t}$ (function 2 in Figure 4), $(int)\frac{1}{t}$ (function 3), $sin\frac{1}{t} \cdot t$ (function 1, continuous but Zeno).

Next we provide some more rationale for definition 9. We argued that the definition 8 would be too strong if applied to uncountable domains, as it would rule out acceptable behaviors. On the other hand, the weaker requirement of simple continuity would not be sufficiently accurate: requiring only continuity would not allow us to freely use variables in expressions, without running into Zeno predicates, as defined in Definition 4. Consider for example a time dependent variable v equal to the function $sin\frac{1}{t} \cdot t$ in a left interval of the origin. This function is pictured in the figure 4(1) and it is continuous. Nonetheless the formula $v = 0$ is Zeno because near the origin there is an accumulation point of isolated zeros. Indeed neither the formula $UpToNow(x = 0)$ holds at the origin, nor $UpToNow(x \neq 0)$; the variable takes and leaves infinitely often the 0 value.

In the same way, even if a function is in C_n, it could be Zeno. Another classic example is the function $sin\frac{1}{t} \cdot e^{-1/t^2}$: it is very regular, even C_∞, (but not analytic) and indeed it does not satisfy our intuitive requirements for non-Zenoness, because it changes sign an infinite number of times in every interval surrounding the origin.

The following theorem shows that only non-Zeno formulas are obtained comparing non-Zeno variables with constant values.

Theorem 1. *if the TD variable* v *in the domain D is non-Zeno, then forall a in D the formula* $v = a$ *is non-Zeno.*

Proof. The graph of an analytic function cannot have infinitely many intersections with a line y = constant (or any straight line) in a finite interval [Courant and Fritz 1974]. □

Therefore piecewise analytic variables correctly exclude undesired behaviors, allow their use without the risk of introducing Zeno entities, and moreover, they comprehend variables, such as sinusoids or ramps or other similar ones, often used to model real word quantities.

Example 3. In the GRC problem an example of non-Zeno time dependent variable in a uncountable domain is the angle of the bar, *gateAngle* with domain the interval of real numbers between 0 and 90. ◇

2.9 Non-Zeno point and interval variable

The following two propositions, whose proof is reported in [Gargantini and Morzenti 1999], combine the definition of non-Zeno and of point and interval variables, in order to obtain definitions of behaviors of practical interest.

Proposition 1. **non-Zeno point variable:** *a TD variable* x *in a domain D is a non-Zeno point variable with default value* $d \in D$ *iff:*

$$UpToNow(x = d) \wedge NowOn(x = d)$$

Non-Zeno point based variables with default value 0 can model non-Zeno events with simultaneous occurrences (as mentioned in 2.7.1).

Example 4. In the GRC case study consider the event *"a train enters region R"*. As there are several tracks, more than one train can enter the region R at the same time. Therefore we have to model possibly simultaneous occurrences of this event or equivalently the event "*N trains enter now region R*". Furthermore the number of trains that might enter the region R in a finite time interval is bound by the number of tracks. We can use the non-Zeno point integer variable RI with default value 0 to model this event. RI is always equal to 0 except in single time instants when one or more trains enter R. At these instants RI is equal to the number of trains entering R. In the same way we introduce the following point integer variables with default value 0: II models the trains now entering region I, IO those now exiting region I (and therefore region R as well). ♢

PROPOSITION 2. **non-Zeno interval variable:** *a TD variable* x *in a domain D is a non-Zeno interval variable iff it is piecewise constant:*

$$\exists a \in D\ \exists b \in D\ \ (UpToNow(x = a) \wedge NowOn(x = b) \wedge (x = a \vee x = b))$$

that means x is piecewise constant and, at time points when it changes, it either keeps the previous value (the value before the change) or it assumes the new value (the value after the change).

Since a non-Zeno interval variable is piecewise constant, expressions like "the value of x immediately before (after) the current instant" are always well defined. Therefore we can introduce in TRIO two operators $uptonow(x)$ and $nowon(x)$, denoting the value of x immediately before and after the current time instant.

PROPOSITION 3. **non-Zeno left (right) continuous interval variable:** *a TD variable* x *in a domain D is a non-Zeno and left (right) continuous interval variable iff:*

$$\exists a \in D\ \exists b \in D\ \ (UpToNow(x = a) \wedge NowOn(x = b) \wedge x = a)$$
$$(\exists a \in D\ \exists b \in D\ \ (UpToNow(x = a) \wedge NowOn(x = b) \wedge x = b))$$

Example 5. In the GRC case study an example of non-Zeno left continuous interval variable is the state of the bar. To represent the state of the bar we introduce a non-Zeno left continuous variable *gate* with domain { *open*, *closed*, *movingDown*, *movingUp*}. ♢

Example 6. Another example of interval variables are the event counters $\#RI$, $\#II$ and $\#IO$ respectively for the events RI, II, and IO. They count the total number of trains entered or exited from the regions of interest. For instance, $\#RI$ is equal to the total number of trains that have entered region R till now. ♢

2.10 Closure properties of variables

As seen for formulas, we are interested in closure properties of the various kinds of variables. The comparison between variables of the same type (application of the relational operators $=, \neq, <, \leq, \geq, >$) gives as result a formula of the same type; thus, for example, the comparison between two point-based variables is an event. Concerning numeric variables, the set of point-based variables and left- and right-continuous interval variables are closed with respect to arithmetic operations (+, -,

point-based predicate E	$E \rightarrow (UpToNow(\neg E) \wedge NowOn(\neg E))$
point-based variable v	$\exists d \in D\, v \neq d \rightarrow (UpToNow(v = d) \wedge NowOn(v = d))$
interval-based predicate I	$\begin{array}{l} I \rightarrow \quad (UpToNow(I) \vee NowOn(I)) \\ \neg I \rightarrow (UpToNow(\neg I) \vee NowOn(\neg I)) \end{array} \wedge$
interval-based variable v	$\forall a \in D\, v = a \rightarrow (UpToNow(v = a) \vee NowOn(v = a))$
left continuous interval pred. I	$(I \rightarrow UpToNow(I)) \wedge (\neg I \rightarrow UpToNow(\neg I))$
left continuous interval var. v	$\forall a \in D \;\, v = a \rightarrow UpToNow(v = a)$
non-Zeno predicate A	$\begin{array}{c} UpToNow(\neg A) \;\vee\; UpToNow(A) \\ NowOn(\neg A) \;\vee\; NowOn(A) \end{array} \wedge$
non-Zeno variable v	$\exists f \exists g \left[\begin{array}{c} f \in AF_o \wedge g \in AF_o \wedge \\ \exists d \forall t \left[\begin{array}{c} 0 < t \wedge \\ t < d \end{array} \rightarrow \left(\begin{array}{c} futr(v,t) = f(t) \wedge \\ past(v,t) = g(t) \end{array} \right) \right] \end{array} \right]$
non-Zeno point predicate E	$UpToNow(\neg E) \wedge NowOn(\neg E)$
non-Zeno point variable v	$\exists d \in D\; UpToNow(v = d) \wedge NowOn(v = d)$
non-Zeno interval predicate I	$\begin{array}{llll} UpToNow(I) & \wedge\; NowOn(I) & \wedge I & \vee \\ UpToNow(\neg I) & \wedge\; NowOn(\neg I) & \wedge \neg I & \vee \\ UpToNow(\neg I) & \wedge\; NowOn(I) & & \vee \\ UpToNow(I) & \wedge\; NowOn(\neg I) & & \end{array}$
non-Zeno interval variable v	$\begin{array}{l} \exists a \in D \wedge \\ \exists b \in D \end{array} \left(\begin{array}{c} UpToNow(v = a) \wedge (v = a \vee v = b) \\ \wedge \quad NowOn(v = b) \end{array} \right)$
non-Zeno left continuous interval predicate A	$\left(\begin{array}{c} UpToNow(A) \wedge A \\ UpToNow(\neg A) \wedge \neg A \end{array} \vee \right) \wedge \left(\begin{array}{c} NowOn(A) \\ NowOn(\neg A) \end{array} \vee \right)$
non-Zeno left continuous interval variable v	$\exists a \in D\, \exists b \in D \left(\begin{array}{c} UpToNow(v = a) \wedge v = a \\ \wedge NowOn(v = b) \end{array} \right)$

Table 4. All the definitions

*, and / - excluding division by 0) whereas this is not the case for generic interval variables. [3]

These properties, together with those seen in Section 2.6, make it more convenient to use only a particular kind of continuity. Using consistently one of the two conventions (left- or right- continuous) the specification and its analysis are greatly simplified; therefore in the following, unless otherwise explicitly stated, we will assume that in our specifications interval predicates and variables are non-Zeno and left-continuous (right-continuous would be equally acceptable). See Section 5 for a brief comparison with other languages.

2.11 Conclusive remarks

Table 4 [4] summarizes the definitions of the most relevant types of entities introduced in the present section. We have so far introduced the notion of point-based and interval based predicates, formulas, and variables, and we stated and proved some interesting properties. We expect that these notions would be useful in mod-

[3]These facts were proven in PVS as JUDGMENT about the types that in the PVS system represent the various kind of entities. For instance:

```
JUDGEMENT + HAS_TYPE [PEnt[real],PEnt[real] -> PEnt[real]]
```

expresses the fact that the sum of two point-based variables is point-based.

[4]In Table 4 v is a TD variable with domain D

eling components of real-world systems, but their use is not intended to be mandatory. In general, during System Requirement Analysis an engineer would be free to introduce any predicate or variable, together with suitable axioms describing relevant properties: only non-Zenoness is expected to be a truly general and necessary assumption. One of the purposes of the present work is to encourage engineers performing SRA to identify from the very beginning some typical patterns of behavior, and to model them in terms of suitable predefined entities. This allows the engineer to take for granted a set of properties, to use them for further analysis and proofs of more elaborate properties, and to easily perform simple but quite effective checks of correctness, completeness, and consistency, uncovering as early as possible errors in the model.

3. TEMPORAL AND CAUSAL RELATIONS AMONG ENTITIES

In the previous section we focused our attention on various types of temporal entities; now we introduce constructs to relate the system entities with each other. In TRIO, as well as in any descriptive formal notation, these relations could be modeled by means of suitable axioms, defined *ad hoc* for each single system to be modeled and analyzed. Even though ad hoc axioms can always be employed to express any kind of relation among entities, in the same spirit as in Section 1, we introduce here some general, high level and intuitively appealing constructs to model a few typical, very frequent types of relation. After a very brief explanation of our model, in Section 3.2 we introduce a direct way to define new derived entities from other previously defined ones. In Section 3.3 we introduce a method to model the change of interval variables as caused by events, thus formalizing a cause-effect relation between events and interval variables. In Section 3.4 we introduce periodic events. In the last subsection we discuss means to model cause- effect relations between events.

3.1 Model - Derived entities

As we briefly recalled in the introduction, a critical system may abstractly be viewed as composed of a computerized device (the Device Under Construction, DUC) that interacts with its environment, which it is in charge of monitoring or controlling, through an interface consisting of a set of sensors and actuators. This view is consistent with several models proposed in the literature, see for instance Parnas' Four Variables model [Parnas and Madey 1995] and Jackson & Zave's model [Zave and Jackson 1997].

Quite often the duty of the DUC may be described as computing the value of its output to actuators starting from the inputs produced by the sensors, therefore its specification should describe clearly and unambiguously the desired relation between input and output. To facilitate the definition of this relation, designers often find it useful to introduce new, derived "internal" entities, that do not directly correspond to real-world elements of the environment, but are computed from the input and typically account for the current state of the computation in an explicit and human-understandable way.

In the following subsections we will introduce constructs useful for defining derived entities and relations among them.

3.2 Directly Defined Entities

The simplest way to introduce derived entities is just to define them as directly corresponding to other system entities. For example an event E might be introduced by defining an axiom like:

$$E \leftrightarrow EventFormula$$

where E is the name of the event and *EventFormula* is a TRIO formula that has point behavior by construction (see Section 2.6 and Tables 2 and 3 for rules providing sufficient conditions to check whether a formula is an event). A definition of that type introduces a necessary and sufficient condition for E.

In the same way as for predicates, designers can define new variables, by simply introducing new axioms similar to the following one, used to derive the variable x.

$$x = expression$$

When the designer uses a definition of this type, he or she must check that x and *expression* are of the same type, using the definition of the supposed type (given in Table 4) or directly the rules given in Section 2.10. In the tool that we have built the constraint about *expression* is automatically generated (as TCC: Type Correctness Condition) and the user is in charge of proving it, possibly with the assistance of the tool itself.

Example 7. In our case study we define these two events:

$$\begin{array}{lcl} stopMovingUp & \leftrightarrow & Becomes(gateAngle = 90) \\ stopMovingDown & \leftrightarrow & Becomes(gateAngle = 0) \end{array}$$

stopMovingUp occurs when the gate reaches the open position, i.e. when its angle becomes equal to 90 degrees. The other event, *stopMovingDown,* occurs when the gate reaches the closed position, i.e. its angle becomes equal to 0. In reference to the equation (1) (page 2) these two definitions are considered as part of the bar actuator model.

We define also the following "internal" variables:

$$\begin{array}{lcl} CTI & = & \#II - \#IO \\ CTPI & = & past(\#RI, d_m) - \#IO \\ CTPI_\gamma & = & past(\#RI, d_m - \gamma) - \#IO \end{array}$$

CTI is the number of trains currently in region I. $CTPI$ is the maximum number of trains that can possibly be in region I given the inputs RI and IO from the sensors up to the present time: the length of time d_m in the past operators is derived from the pessimistic assumption of maximum speed of trains moving from region R to region I. $CTPI_\gamma$ includes a forward shift γ, taking into account the time it takes the bar to reach the down position starting from the open posture: $CTPI_\gamma$ models the number of trains that can possibly be in I within γ time units.

In our example we define directly also the outputs of the controller. *down* is the event that models the command to lower the bar. In our model it is issued as soon as the number of trains that can possibly be in region I within γ time units, i. e. $CTPI_\gamma$, becomes greater than 0. The other command, *up*, which models the controlled command to raise the bar, is issued as soon as $CTPI_\gamma$ becomes equal to or less than 0. In summary, the chosen policy of issuing bar commands can be specified by the following direct definitions.

$$\begin{array}{ll} down & \leftrightarrow Becomes(CTPI_\gamma > 0) \\ up & \leftrightarrow Becomes(CTPI_\gamma \leq 0) \end{array}$$

In reference to the equation (1) *CTI*, *CTPI*, $CTPI_\gamma$ and definitions for *down* and *up* are considered as part of the controller specification, i.e. parts introduced by the designer to model the device under construction and its behavior.

The angle of the bar, which belongs to the actuator model, is modeled as follows:

$$gateAngle = \begin{cases} gate = closed : 0 \\ gate = open \quad : 90 \\ gate = movingDown \quad : \\ \qquad LastTime(Becomes(gate = movingDown), t) \rightarrow \\ \qquad\qquad\qquad past(gateAngle, t) - speed \cdot t \\ gate = movingUp \quad : \\ \qquad LastTime(Becomes(gate = movingUp), t) \rightarrow \\ \qquad\qquad\qquad past(gateAngle, t) + speed \cdot t \end{cases}$$

If the bar is closed (open) the value of the angle is equal to 0 (90). If the bar is closing (i.e. moving down), the angle is decreasing and equal to the value it had when it started closing (t time units ago) minus the value of $speed \cdot t$, where *speed* is equal to $90/\gamma$. The case when the bar is moving up is treated symmetrically in the last clause above. $\diamondsuit$

3.3 Interval variables changed by events

Very commonly in a specification interval variables change when some event occurs. For example in the GRC case study the variable *gate* changes its value from *open* to *movingDown* when the command *down* (which is in fact modeled by an event) is issued. The other transitions of the variable *gate* can be modeled in a similar way.

Using events to trigger a value change of an interval variable is a solution adopted in many formal notations, see for instance [Heitmeyer et al. 1996].

A very general and compact way to formalize this kind of behavior is by introducing the following relation.

Definition 10. **Change_relation:** *for an interval variable* x *with domain D, given a set of events* E^5, *we define a relation over* $D \times E \times D$ *and we call it* ***change_relation***$_x$

For example for the interval variable *gate* we define its *change_relation*$_{gate}$ and call it ***gate_behavior*** as a relation among $D \times E \times D$ where $D = \{open, movingUp, movingDown, closed\}$ and $E = \{up, down\}$. The fact that *gate* changes from *open* to *movingDown* when *down* occurs, is formalized by the triple (*open, down, movingDown*) belonging to the relation *gate_behavior.*

[5]It is assumed that in practice the relation will be defined on a subset of all the events defined in the system, i.e., on the set of events that are relevant to the variable. However, since the relation admits tuples of the form *change_relation*(x,e,x) one could comprise in E all the events, including those that have no influence on the variable.

The intended meaning of the change relation is informally described by the following clauses. The first one considers the simplest case in which the variable does not change:

(1) for every value $x \in D$, if there is no $e \in E$ and no $y \in D$ such that $change\_relation_x(x, e, y)$, then the occurrence of event e when the value of the variable is x does not affect it;

In all other cases the variable changes its value, possibly in a non deterministic fashion:

(2) if $change\_relation_x(x, e_1, y)$ and $change\_relation_x(x, e_2, z)$, then, when e_1 and e_2 occur simultaneously, the variable may nondeterministically turn from x into y or z;

Point 2 includes these particular cases: if $e_1 = e_2$ (for brevity $= e$), and e occurs, then the variable can change either into y or z; if y is equal to x, then the variable can either change to z or keep its value x.

We can formally express this behavior of a variable x, by the following two TRIO axioms. Clause 1 above (describing the case when the variable keeps its value) is formalized by Axiom 1:

Axiom 1. **continue**

$$UpToNow(x = old) \wedge \neg\exists e, new(change\_relation_x(old, e, new) \wedge e) \\ \to NowOn(x = old)$$

The clause 2 (describing the cases when the variable x may change its value) is formalized by Axiom 2:

Axiom 2. **change**

$$UpToNow(x = old) \wedge \exists e_1, new_1\, (change\_relation_x(old, e_1, new_1) \wedge e_1) \\ \to \exists e_2, new_2\, (change\_relation_x(old, e_2, new_2) \wedge e_2 \wedge NowOn(x = new_2))$$

In Axiom 2 the event e_1 and the value new_1 in the antecedent of the implication can be different from those (e_2, new_2) in the consequent: the double existential quantification formalizes the possibility of non determinism.

Notice that the two axioms are *consistent* (i.e. not contradictory) and *complete.* They are *consistent* because at any time exactly one of the two has the antecedent of the implication satisfied and therefore only one of them determines the value of s in the future (from now on). They are *complete,* because one of two antecedents is certainly true, as there is always one and only one *old* value that satisfies $UpToNow(s{=}old)$ and one of the second conjuncts of the two conjunctions that constitute the premises is true.

To be more precisely, the two axioms are meta-axioms, since they refer to a generic variable x and its $change\_relation_x$. When the designer defines the $change\_relation_x$ for a variable x, he or she should derive the actual TRIO axioms substituting both x and $change\_relation_x$. Exploiting the higher order language of PVS we define the axioms for a generic variable and generic relation (as parameters of the PVS theory where generic axioms are defined) and the actual axioms are automatically derived by PVS.

Using tables. The use of tables in specifications is known to provide great benefits, and tables are particularly suitable to specify relations [Parnas 1992]. Some notations intensively use tables, since tables offer a concise and clear way to specify software and system requirements. SCR [Heitmeyer et al. 1996] is based on a tabular notation and offers a method to check interesting (and system independent) properties of tables (and then properties of the specification and of the system). In cases where for every value and every event there exists a finite number of possible new value, tables are a very intuitive means to specify the above described relation, as shown by the following example.

Example 8. In our case study, the relation *gate_behavior* can be specified by the following, quite simple and self-explanatory table.

UpToNow	event	NowOn
closed	*up*	*movingUp*
movingUp	*down*	*movingDown*
movingUp	$stopMovingUp \wedge \neg\ down$	*open*
movingDown	*up*	*movingUp*
movingDown	$stopMovingDown \wedge \neg\ up$	*closed*
open	*down*	*movingDown*

Note that, with reference to the equation (1), this table is part of the bar actuator model. ♢

Determinism. As previously noticed, the change relation may or may not describe a deterministic change. Deterministic behavior is quite frequent, especially in the design specification of a computerized device, whereas nondeterminism is a frequent feature of the environment. Here we determine the necessary and sufficient conditions on *change_relation* characterizing a deterministic behavior:

Definition 11. **Deterministic relation:** *a change_relation for an interval variable describes a deterministic behavior iff for every value old, n_1, n_2 in the domain D, and for every event e_1 and e_2*:

$$change\_relation(old, e_1, n_1) \wedge change\_relation(old, e_2, n_2) \rightarrow \\ Alw\,(\neg(e_1 \wedge e_2)) \vee n_1 = n_2$$

Note 3. It can be easily verified that a change relation is deterministic if and only if both the following properties hold:

1. *next state uniqueness* (if an event e can change the variable value, the next value is unique):

$$change\_relation(old, e, n_1) \wedge change\_relation(old, e, n_2) \rightarrow n_1 = n_2$$

2. *event disjointness* (if two different events change in a different way the variable value, they never simultaneously occur):

$$change\_relation(old, e_1, n_1) \wedge change\_relation(old, e_2, n_2) \wedge \\ e_1 \neq e_2 \wedge n_1 \neq n_2 \quad \rightarrow \quad Alw\,(\neg(e_1 \wedge e_2))$$

Definition 11 gives the designer a simple criterion to check whether the specification is deterministic or not. In the particular case of a deterministic relation, the axiom of change is simplified as follows:

PROPOSITION 4. **Change for a deterministic change_relation:** *for a deterministic relation axiom 2 (change) is equivalent to the formula:*

$$UpToNow(x = old) \wedge change\_relation_x(old, e, new) \wedge e \rightarrow NowOn(x = new)$$

while the first axiom (continue) remains unchanged. Proposition 4 has been proved with the TRIO prover based on PVS.

Note 4. *Non-Zenoness.* Variables whose value is ruled by a *change_relation* on a non-Zeno event set are non-Zeno[6].

Example 9. The relation given for the bar has been verified to be deterministic. ◊

Note 5. *Initial values.* The relation *change_relation* specifies only the way interval variables change, not their value at any given (possibly initial) time. These kind of values should somehow be explicitly provided by the designer.

Counters. The first application of the proposed construct is the definition of event counters. These are integer variables that count the number of occurrences of a given event. They have a very simple temporal behavior, in that they are normally stable unless the counted event occurs, in which case they increment their value.

The behavior of an event counter is described by the *change_relation* displayed in the table below:

UpToNow	event	NowOn
n	E	n+1

For a counter of an event admitting multiple simultaneous occurrences, the table becomes (with i > 0):

UpToNow	event	NowOn
n	E=i	n+i

Note 6. *First event occurrence and initial value of counters.* The relation given in the table models only the change of the counter and not its absolute value. If the number of occurrences is infinite in the past and in the future, the counter will be an integer number and the designer should fix its a value at some time (just like for other interval variables whose behavior is ruled by *change_relation*).

In the more realistic but less general hypothesis that there is a first occurrence of an event, i.e. that the formula $Som(AlwP(\neg E))$ holds, the initial value of the counter can be fixed as 0, and the counter assumes the meaning of total number of event occurrences. This hypothesis is often appropriate, because in every practical computer-based system there is a "start of operation" before which no significant event occurs. This is in fact the definition of counters adopted in our GRC case study and in the PVS encoding.

[6]Informally, a set of events is non-Zeno if there is no accumulation point of occurrences of events in the set. For a formal definition see [Gargantini and Morzenti 1999].

3.4 Periodic Events

Periodic events have occurrences that are repeated at regular time intervals. Next we list definitions for periodic events and some properties which were proved with the assistance of PVS.

Definition 12. **Semi-periodic Event** *A is a periodic event in the future, with period* d *iff*

$$A \rightarrow NextTime(A, d)$$

Starting from this definition we are able to derive the following propositions:

(1) A is actually an event
(2) $LastTime(A, d) \rightarrow A$: A is periodic in the future
(3) $A \rightarrow (LastTime(A, d) \vee AlwP(\neg A))$: A repeats itself in the past unless it never occurred before.
(4) $A \rightarrow \forall k Futr(A, k \cdot d)$: A repeats itself every d time units in the future

However, for semi-periodic events $Som(A)$ cannot be proven, i.e., it is not guaranteed that the event occurs sometime. For this reason we call it *semi* periodic. If we add the condition $Lasted(\neg A, d) \rightarrow A$, we obtain the following definition of periodic events.

Definition 13. **Periodic Event:** *An event A is called periodic with the period* d *iff:*

$$A \leftrightarrow Lasted(\neg A, d)$$

For periodic events we prove the following propositions:

(1) $A \rightarrow Lasts(\neg A, d)$, $A \rightarrow Futr(A, d)$, and $A \rightarrow NextTime(A, d)$: A is semi periodic in the future
(2) $Som(A)$: A sometime occurs
(3) $A \rightarrow Lasted(\neg A, d)$, $A \rightarrow Past(A, d)$, and $A \rightarrow LastTime(A, d)$: A is periodic in the past
(4) $A \rightarrow \forall k Dist(A, k \cdot d)$: A repeats itself every d time units in the future and in the past

3.5 Temporal relationships between events.

The simplest and most common relationship between events is cause-effect. To formalize the simple fact that an event B occurs exactly d time units after another event A that constitutes its cause the following axiom suffices.

$$A \rightarrow Futr(B, d)$$

If event A is the unique cause of event B then the implication holds in both directions, leading to the following formalization.[7]

$$A \leftrightarrow Futr(B, d)$$

[7]Since, as previously recalled, all axioms are intended as prefixed by an external Alw operator, the formula $A \leftrightarrow Futr(B, d)$ is equivalent to the following, perhaps more intuitive one: $(A \rightarrow Futr(B, d)) \wedge (B \rightarrow Past(A, d))$.

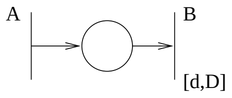
Notice that the above cases of cause-effect relation are *deterministic* as, once the cause (event A) occurs, then the occurrence time of the effect (event B) is precisely determined.

A more general and common relationship between two events is the one in which an event A causes another event $B,$ in a future time that is not known precisely, due to some nondeterminism of the system. Typically, the delay between A and B is characterized by means of a lower bound and an upper bound denoting the minimal and maximal time distance between related occurrences of the two events. The relation is therefore nondeterministic, being the exact instant in the future, when B will occur after A, unknown. We model here the more common and interesting case where the relation is one-to-one (for instance because A is the unique cause of B: every occurrence of A causes one B and every occurrence of B is caused by one occurrence of A). An example of this kind in a concurrent systems could be the relation between the event of taking a resource and that of returning it.

We provide a preliminary formal definition of this kind of relation as follows.

Definition 14. *An event A is a unique cause for an event B in $[d,D]$ time units, where d and D are positive real constants such that $d \leq D$, iff there exists a one-to-one function ϕ from the occurrence times of A to those of B such that $t + d \leq \phi(t) \leq t + D$.*

This relationship is widely used and in several formalisms it is the only temporal relationship between events: see for instance timed Automata in [Archer and Heitmeyer 1996; Merritt et al. 1991] and Time Petri Nets (TPN) in [Merlin and Farber 1976]. In TPN it is pictured as follows.



Next we formulate this definition in terms of TRIO axioms that refer to event and event counters. We will discuss two solutions, one introducing an additional predicate, and one based on counters; their equivalence is proved in [Gargantini and Morzenti 1999] by showing that they both model the relation introduced in Definition 14.

Notice that simple, apparently obvious formalizations are easily proved incorrect. For instance, in [Gargantini and Morzenti 1999] we show that the following pair of axioms

$$A \rightarrow \exists t(d \leq t \leq D \wedge Futr(B,t)) \qquad \text{and} \qquad B \rightarrow \exists t(d \leq t \leq D \wedge Past(A,t))$$

do not capture the one-to-one binding between occurrences of A and B[8].

Using special predicates to formalize the relationship. Proposed solutions can be found in [Felder et al. 1994] and in [Mandrioli et al. 1996]. In [Felder et al. 1994] the authors use special predicates to introduce the causal relationship between a firing of a transition and the firing of another transition in TPN. The same approach can

[8]We are assuming here that event occurrences *cannot* be uniquely identified; otherwise axiomatizations like the one above, with the addition to the event predicate of an argument identifying event occurrences, could suffice [Koymans 1989].

be exemplified for two generic events A and B as follows. We introduce a TRIO time dependent predicate $ACausesB(t)$, that, when true at a given time, means that A occurs at that time and causes an occurrence of B t time units later (with $t \geq 0$). The following axioms characterize predicate $ACausesB$.

1	occurrences	$ACausesB(t) \rightarrow A \wedge Futr(B,t)$
2	cause	$A \rightarrow \exists t \ (d \leq t \leq D \wedge ACausesB(t))$
3	effect	$B \rightarrow \exists t \ (d \leq t \leq D \wedge Past(ACausesB(t),t))$
4	cause uniqueness	$ACausesB(t_1) \wedge ACausesB(t_2) \rightarrow t_1{=}t_2$
5	effect uniqueness	$Past(ACausesB(t_1),t_1) \wedge Past(ACausesB(t_2),t_2) \rightarrow t_1 = t_2$

It is immediate to prove that this formalization allows one to introduce a one-to-one function ϕ between the occurrences of A at time t to those of B at time $\phi(t)$ such that $t + d \leq \phi(t) \leq t + D$ (see [Gargantini and Morzenti 1999]).

Using counters. We now introduce a way to bind events A and B through a simple formula of the counters of their occurrences, respectively denoted as $\#A$ and $\#B$. In the derivation of this formula [Gargantini and Morzenti 1999] we assumed that, for any event E, there exists a first occurrence, i.e., that there is an instant before which E never occurred, a fact that is formalized in TRIO as $Som(AlwP(\neg E))$. As noted above, this hypothesis is quite realistic for real-world systems. A less restrictive assumption is however adopted in the proof of the theorem 2, reported in [Gargantini and Morzenti 1999], showing that it is immaterial from a mathematical viewpoint.

Theorem 2. *Event A is a unique cause of event B in $[d,D]$ time iff:*

$$past(\#A, D) \leq \#B \leq past(\#A, d)$$

Intuitively, if the relation of Definition 14 holds, when an event A occurs, causing an increment in counter $\#A$, then counter $\#B$ is also bound to increase; however, due to the assumed delay ranging between d and D, counter $\#B$ will increase no earlier than d time units after the increase of $\#A$, hence the inequality $\#B \leq past(\#A,d)$ holds; moreover, and symmetrically, $\#B$ will increase no later than D time units after $\#A$, hence $\#B \geq past(\#A,D)$ holds.

Theorem 2 expresses the concept stated in Definition 14 with very simple relations between event counters. Thanks to their simplicity (they are just linear inequalities) these relations can be very easily and effectively used in the derivation of relevant properties. Furthermore, specifications based on counters are readily implementable, since counters are trivially computable by means of increments of integer-valued program variables.

Particular cases and generalizations. The model can be both generalized and applied to particular cases. For instance, if the minimum delay d is zero (event B can follow immediately event A) the relation becomes: $past(\#A, D) \leq \#B \leq \#A$.

If the delay has no upper bound, then $D = \infty$ and the relation reduces to: $\#B \leq past(\#A, d)$.

Definition 14, introduced for simple events, can be extended to events with multiple simultaneous occurrences. Theorem 2 maintains its validity also in this generalized framework.

A further, interesting generalization of the one-to-one relation introduced in Definition 14 is to allow a negative minimum time *d*. In this most general case one would not model a "cause-effect" relation, but a correspondence between event occurrences that are somehow related, for instance because they are both effect of a common (unique) cause. Theorem 2 can be generalized in a straightforward way to this case by just changing the *past* operators (which assume a positive time argument and necessarily refer to previous instants) into *dist* operators (which equally admit a positive, null, or negative time argument, thus referring to both past, present and future), obtaining the following relation

$$dist(\#B, d) \leq \#A \leq dist(\#B, D)$$

which holds under the unique assumption that $d \leq D$.

Similarly, the special predicate previously called $ACausesB(t)$ can be generalized to $AoneTooneB(t)$, where the argument t could be negative, and the related axioms adapted by changing the *Past* and *Futr* operators to *Dist*.

1	occurrences	$AoneTooneB(t) \rightarrow A \wedge Dist(B,t)$
2	relation one way	$A \rightarrow \exists t\ (d \leq t \leq D \wedge AoneTooneB(t))$
3	relation the other way	$B \rightarrow \exists t\ (d \leq t \leq D \wedge Dist(AoneTooneB(t),t))$
4	uniqueness one way	$AoneTooneB(t_1) \wedge AoneTooneB(t_2) \rightarrow t_1 = t_2$
5	uniqueness the other way	$Dist(AoneTooneB(t_1), t_1) \wedge Dist(AoneTooneB(t_2), t_2) \rightarrow t_1 = t_2$

As a concrete example, let us consider an electronic trading system where an order for some goods performed by a client gives rise subsequently, through independent chains of actions, to the physical delivery at the client's address of the parcel containing the ordered item, and to the billing of the price on the client's bank account. An important property of the trading system could be that there is a one-to-one matching between goods delivery and bank account transactions. These two events are clearly related, but there might be no strict temporal precedence between them. If we model goods delivery by the event predicate *GD* and bank account transactions by event predicate *BAT*, then we can abstractly specify that each occurrence of *GD* may at most precede the corresponding occurrence of *BAT* by 3 days, or at most follow it by 4 days, using the following inequalities

$$dist(\#GD, -3) \leq \#BAT \leq dist(\#GD, 4)$$

Example 10. In the GRC case study a one-to-one temporal relationship obviously exists between the entering of trains in the various regions surrounding the crossing. The system is nondeterministic due to the uncertainty about the trains speed, which may vary between minimum and maximum allowed values.

The informal specification asserts that the trains take a minimum time d_m and a maximum time d_M to go from the beginning of region R to the beginning of region I; it takes a minimum time h_m and a maximum time h_M to go from the beginning of region I to its end.

These relations are formalized by the following inequalities between counters of event occurrences (recall that *RI*, *II*, and *IO* are defined, respectively, as the events of trains entering region *R*, entering region *I*, and exiting region *I*, and that they

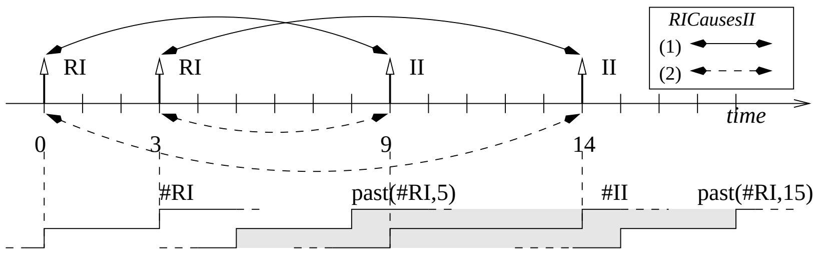


Fig. 5. Models for axiomatizations based on special predicates or on counters.

are multiple events).

$$past(\#RI, d_M) \leq \#II \leq past(\#RI, d_m)$$
$$past(\#II, h_M) \leq \#IO \leq past(\#II, h_m)$$

Notice that, with reference to the equation (1), this relation belongs to the model for the environment. Indeed the trains speed is a constraint about the environment where our controller will operate. ♢

Axiom alphabet and models. The two proposed axiomatizations of cause-effect relation between events, in terms of predicates (like *ACausesB*) or counters of event occurrences are both suitable to formalize in TRIO a one-to-one relation like that of Definition 14, but they are not completely equivalent for what concerns the information content of the resulting models. A model for an axiomatization based on a predicate like *ACausesB* contains, as interpretation of that predicate, a relation whose tuples establish the exact mapping between the various occurrences of the cause and effect events. On the contrary, a model for an axiomatization expressed in terms of counters of event occurrences defines just the values of the counters as events occur in time, without providing any information about the matching between event occurrences. Of course, the simpler but slightly less informative description in terms of counters suffices in the cases where the exact matching between the event occurrences is immaterial with respect to the desired system properties. In other cases, where the exact matching between event occurrences really matters, the slightly more complex but also more accurate description provided by predicates like *ACausesB* may be preferable.

As an illustration of this, consider again the GRC example, with a plant where d_m =5 and d_M =15. Suppose there are two occurrences of RI at times 0 and 3 (i.e., a train enters region R at time 0 and a second train enters at time 3), and two occurrences of event II at times 9 and 14 (i.e., a train enters region I at time 9 and another one at time 14). From a physical viewpoint there are two interpretations of this event sequences: either the trains enter in region I in the same order as they entered in region R, or the train that entered region R last passes the first one, and enters region I before it. Correspondingly, there are two models for predicate *RICausesII* including these event occurrences, as shown in Figure 5: in each model the relation shows which event of the kind "entrance in region I" corresponds to each event of the kind "entrance in region R".

When we use event counters, instead, we model the fact that "sensors do not recognize trains", so that there is just one possible model, shown in Figure 5,

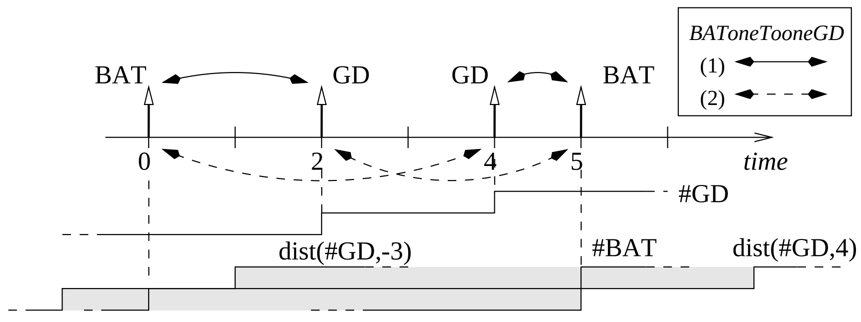


Fig. 6. BAT and GD

accounting for the total number, up to any given time, of event occurrences of kind "entrance in region R" and "entrance in region I". Notice, however, that the second, less precise description is perfectly adequate to the purpose of governing the Railway Crossing: the safety system does not need to "recognize trains": it can limit itself to counting them.

As an example where the exact matching between event occurrences can be relevant to the desired system properties, consider again the above described electronic trading system, with a sample "history", shown in Figure 6, where two bank account transactions take place at time 0 and 5, and goods delivery occur at time 2 and 4. Figure 6 shows that there is only one model based on counters $\#GD$ and $\#BAT$, and there are two models based on the predicate *BAToneTooneGD*.

(Notice that the specification of the electronic trading system can be further enriched by associating to every goods delivery and bank account transaction the description of the acquired item; in this case it can be easily verified that the number of candidate models increases, and a few additional simple axioms are needed to state the property that related *BAT* and *GD* occurrences must refer to the same acquired item.)

4. ENCODING TRIO IN PVS - TOOL AND PROOFS

Our tool has three components: a set of theories, containing the necessary definitions to encode in PVS the TRIO variables, operators, and high-level entities; a pretty printer to support a TRIO-based style in formulas and derivations; a set of strategies to simplify the reasoning with the given definitions and with time.

4.1 Definition of time

Since our method applies to asynchronous hybrid systems we chose to model time in PVS simply as a real value:

```
time : TYPE = real
```

According to the conventions adopted in TRIO, in our encoding the measure of time is supposed to be relative to the current instant: for instance, *"time 0"* means *" now"*, *"time 5"* means *" 5 time units in the future"*. This relative notion of time is however not appropriate to represent duration of time intervals, for which an ad hoc definition is provided as follows.

```
duration : TYPE = {t:time | t>=0}
```

4.2 Semantic vs. syntactic encodings

The problem of encoding TRIO in PVS is in fact a particular case of the most general problem of encoding a logic (called *source*) into another logic (called *base*). Proposed solutions to this problem can be classified in a framework based on two categories: *syntactic* encoding and *semantic* encoding.

In *syntactic* encodings the source logic is directly encoded in the base logic by means of a metalanguage, provided by the base logic. This metalanguage is used to represent both grammar and inference rules of the source logic. For many systems, for example Isabelle [Paulson 1994], this is the suggested way to encode another logic. Such systems are very versatile and capable, because almost every logic can be encoded with its own syntax and inference rules, nevertheless they provide only a few predefined theories and no powerful decision procedures, because every source logic is supposed to introduce its own inference rules. For example they generally lack decision procedures for arithmetic, which are essential in TRIO because of its quantitative treatment of time.

In *semantic* encodings the semantics of each construct of the source logic is defined using the constructs of the base logic and the base logic's inference rules are used in proofs. The base logic normally has a rich language and type system, and a powerful proof support. The main disadvantage of this approach is that the encoded formulas and proofs may look very different from the original ones. To overcome this disadvantage, semantic encodings are typically paired with a pretty-printer that supports visualization of the encoded formulas in a syntax similar or identical to the original one. On the other hand, the major advantage of semantic encodings is that all the constructs and proof techniques of the base logic, which are often quite powerful and sophisticated, can be exploited. PVS, as an example among many, better supports semantic encodings (even though, using ADT and introducing AXIOMS, syntactic encodings are still possible). Furthermore it includes decision procedures over arithmetic and propositional logic, and a powerful theorem prover. The semantic approach (with a dedicated pretty printer) was adopted for Duration Calculus in [Skakkebæk and Shankar 1994] and in [Dutertre and Stavridou 1997]. Further references can be found in Section 5 and in the web page of PVS (`http://pvs.csl.sri.com`).

Some approaches try to combine the most valuable features of syntactic and semantic encodings: among these we mention TAME for Timed Automata [Archer and Heitmeyer 1997a]; in a previous, preliminary work [Alborghetti et al. 1997], we adopted a so-called suppressed state encoding, where we considered TRIO formulas as an uninterpreted type (a feature typical of syntactic encodings) and, to provide the usual interpretation of TRIO formulas, we introduced a function *now* from the type of TRIO formulas to the booleans, and equipped the system with axioms characterizing the *now* function. This allowed us to avoid the overhead of constructing the pretty-printer for TRIO, at the price, however, of additional complexity and inefficiency in deductions. In the work described here, we chose a semantic encoding coupled with a pretty-printer, which allows us to obtain a maximum of efficiency in derivation and a satisfying visualization of TRIO formulas.

Definition of time dependent terms. Every time dependent term with values in a domain `D` is encoded as a function from time to `D`. This is implemented in PVS through a parametric theory:

```
trio_TD_Terms [D : TYPE+[9]] : THEORY
   BEGIN
   ...
   TD_Term : TYPE = [time -> D]
   ...
```

The domain `D` can be, for example, the integer set (in this case we encode time dependent integer variables) or a more complex type, such as a tuple, a record, or an abstract data-type or a function (then we have a time dependent tuple, a time dependent record, etc.). Considering time dependent entities as functions from time to their domain is a rather standard approach [Dutertre and Stavridou 1997; Skakkebæk and Shankar 1994]. As noted in [Hansen et al. 1998], the same approach is followed by conventional dynamic systems theory [Luenberger 1979], and such model is known to engineers in general, often through its graphical representation by timing diagram.

We define the operator `LV` (lift value) that translates a value `k` in `D` into a time dependent variable (with constant value `k`):

```
LV(k: D) : TD_Term[D] = lambda[10] (t:time): k
```

Since this operator is defined as `CONVERSION`, PVS automatically applies `LV` whenever it finds a time independent value instead of an expected TD value.

Higher order features of PVS allow us to use unique definitions for predicates and for variables in any domain. TD predicates and formulas are encoded simply taking as `D` the boolean set, as follows:

```
TD_Fmla : TYPE+ = TD_Term[bool]
```

Domain operators . Operations in usual domains (boolean, numbers,...) can be extended to time dependent variables in those domains as follows. Consider for instance the operator `AND`, defined in PVS:

```
AND: [bool, bool -> bool]
```

To extend `AND` to time dependent boolean terms, we define another operator `AND` that has TD bool operands and returns a TD bool:

[9]TYPE+ means that D must be a non empty type. Although an empty type would be equally acceptable, empty types are of no practical interest. Furthermore forcing the designer to use non-empty types, can expose inconsistencies in definitions, as pointed out in [Rushby et al. 1998]

[10]In PVS `lambda` expressions denote functions. For example the function which adds 3 to an integer may be written :

```
LAMBDA (x:int):  x+3
```

and its type is function from integer to integer: `[int->int]`

```
AND(a, b: TD_Term[bool]) :
               TD_Term[bool] = lambda (t: time) : a(t) AND b(t)
```

Hence the result of the application of AND is still a function from time to the domain bool, and this function has in every instant the value of the application of the operator AND between the boolean values of the arguments in that instant. Similar definitions are provided for other arithmetical and logical operations on time dependent entities.

Useful closure properties of those operators are proven. For example the fact that the sum of two TD integers is still a TD integer is automatically stated by PVS in the following form.

```
JUDGEMENT + HAS_TYPE [TD_Term[int],TD_Term[int] -> TD_Term[int]]
```

A JUDGEMENT is a statement that the user is required to prove (possibly with the support of PVS itself) and is used by PVS during type checking.

We redefine the equality between two TD variables in the domain D:

```
==(H,K: TD_Term[D]) : TD_Term[bool] = lambda (t: time): H(t) = K(t)
```

The operator == returns a time dependent boolean that is true only in the time instants where H and K are equal.

Temporal operators . Besides usual domain operators, TRIO introduces several temporal operators. The basic temporal operator $Dist(A, d)$ is encoded as a TD predicate (i.e., a function from time to boolean), equal to a translation of A for d time units:

```
Dist(A,d) : TD_Term[bool] = LAMBDA (t:time): A(t+d)
```

The encoding of all the other TRIO temporal operators is based on that of *Dist*. We report the definition of Alw (always) and Som (sometimes), which simply have as result a boolean[11]:

```
Alw(A) : bool = FORALL (t : time) : A(t)
Som(A) : bool = EXISTS (t : time) : A(t)
```

Other derived temporal operators have as result a TD formula (i.e., a time dependent boolean term). For example, if A and B are TD formulas, then $Lasts_{ee}(A, d)$ and $Until_{ee}(A, B)$ are defined as follows.

```
Lasts_ee(A,dur): TD_Term[bool] =
     FA![12] (t : {t:time | 0 <  t AND t <  dur}) : Dist(A,t)
Until_ee(A,B) : TD_Term[bool] =
     EX! ( pt : duration): Futr(B,pt) AND Lasts_ee(A,pt)
```

Thanks to these definitions the user can define in PVS TD variables, TRIO formula, theorems, axioms, etc. with the usual TRIO syntax. For example we define and prove the following theorem (introduced in [Felder et al. 1994]) :

[11]Since Alw(A) and Som(A) do not depend on the time they are stated (thanks to the theorems $Alw(A) \rightarrow Alw(Alw(A))$ and $Som(A) \rightarrow Alw(Som(A))$), we preferred to define them in PVS simply as boolean.

[12]FA! stands for FORALL, and EX! for EXISTS

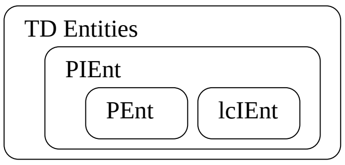


Fig. 7. TD types in our encoding.

```
th_i: theorem
FORALL (A: TD_Fmla, t: time):Alw(Alw(A) IMPLIES Dist(A,t))
```

Definitions for entities with bound behavior. In addition, we provide in PVS definitions for point and interval based entities. To simplify and optimize our encoding, we occasionally deviated from the conceptual path outlined in Section 2. For this reason the definitions that follow are slightly different from those seen in that section. Furthermore, since PVS has no predefined theory supporting the characterization of analytic functions [Dutertre 1996], and building a complete library would have been far beyond the scope of the present work, we exploited simple but very general sufficient conditions (some of which mentioned in Section 2.9), ensuring that functions encoding non-Zeno variables are analytic.

We group non-Zeno point and interval variables in one type called `PIEnt,` to gather common properties of these two types of behavior. `PIEnt` variables might behave sometimes as point variables (i.e. keeping a value only for a single time point), whereas sometimes as interval variables (i.e. piecewise constant). The relation among types we have defined in PVS, is shown in Figure 7.

The definition for `PIEnt` in the domain `D` is the following:

```
PIEnt : NONEMPTY_TYPE =
 {H: TD_Term[D] | Alw(EX! (l,r:D) : UpToNow(H==l) AND NowOn(H==r))}
```

Then we defined type `PEnt` for point variables in the domain `D` (as subtype of `PIEnt`), parametric respect its default value `q`:

```
PEnt(q:D) : NONEMPTY_TYPE =
 {PI : PIEnt[D] | Alw(UpToNow(PI==q) AND NowOn(PI==q))}
```

In a similar fashion we defined interval entities (the type `IEnt`) and left continuous interval entities `lcIEnt`. We report the definition for the latter:

```
lcIEnt : NONEMPTY_TYPE =
 {PI : PIEnt[D] | Alw(FA! (v :D) ( PI==v IMPLIES UpToNow(PI==v)))}
```

Closure properties and theorems for those types (given in the previous sections) have been proven in PVS using the tool itself.

Soundness and completeness. Soundness and completeness are of primary importance for a semantic encoding: the base logic or the encoding may include axioms and proof rules clashing with those of the source logic, or they may miss important properties. In our case we have to prove that every PVS rule is logically valid in TRIO (to prove that the encoding is sound), and all that TRIO axioms (as defined

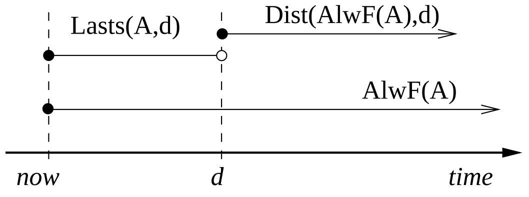


Fig. 8. Application of (merge-temp-ops)

in [Felder et al. 1994]) can be proved in our encoding (for completeness). These proofs are straightforward and reported in [Jeffords 1995].

4.3 Proofs and strategies

We enriched the PVS proving mechanism with a set of strategies that efficiently deal with time and TRIO temporal entities (time dependent variables, operators, and other constructs presented in previous sections). We can divide our strategies in two types: direct extensions of PVS strategies, and new strategies specifically aimed at dealing with time.

Extension of PVS strategies. A typical generalization consists of applying a rule at any instant different from the current time. For example PVS propositional rule $\vdash \wedge$, on the left below, is generalized to $\vdash Dist\wedge$ on the right below[13].

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \vdash \wedge \qquad \frac{\Gamma \vdash Dist(A,t) \qquad \Gamma \vdash Dist(B,t)}{\Gamma \vdash Dist(A \wedge B, t)} \vdash Dist\wedge$$

Then we define new PVS commands in order to apply the generalized rules. For example PVS command `split` is generalized to `trio-split` strategy, that splits conjunctive formula in the current goal sequent even if they are inside temporal operators. Our tool of course generalizes all other commands based on propositional reasoning, including, e.g., case analysis (using `case` and `trio-split` strategies).

Time related strategies . Other strategies, explicitly dealing with time and TRIO entities, are completely original. We have defined a set of strategies to manipulate sequents containing formulas with temporal operators. Here we show a rule $merge \vdash$ that is useful to merge temporal intervals.

$$\frac{\Gamma, AlwF_i(A) \vdash \Delta}{\Gamma, Dist(AlwF_i(A), d), Lasts_{ie}(A, d) \vdash \Delta} merge \vdash$$

This rule and other similar ones are applied by a new strategy, named `merge-te-mp-ops`. An instance application of the strategy is illustrated in Figure 8.

Temporal induction. The change relation introduced in Section 3.3 defines the behavior of interval variables only instant by instant. For an interval variable it

[13]In PVS *inference rules* (see pag 15 [Shankar et al. 1998]) are specified in the form

$$\frac{\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta} R$$

meaning that if our goal is to prove $\Gamma \vdash \Delta$, we can apply the rule R and obtain n (generally simpler) goals to prove: $\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n$.

specifies the next value according to the current value and the now happening events. Temporal induction allows us to move from this step-by-step view to a reasoning on temporal intervals.

Temporal induction can take several forms, depending on whether it is stated in the future or in the past, over an interval, or for ever. Therefore we introduced several definitions in our system and here we present as an illustrative example the one for a left-continuous formula in a future interval:

THEOREM 3. **temporal induction:** *for every left continuous TD formula* A *and duration* d, *if* A *is true now and, for a future time interval lasting* d *time units,* A → NowOn(A)*, then* A *holds for all that interval.*

$$A \wedge Lasts(A \to NowOn(A), d) \vdash Lasts(A, d)$$

As a simple application, this theorem can be used to prove the following lemma:

LEMMA 1. *If* x *is a time dependent variable in the domain D whose behavior is defined by its* $change\_relation_x$ *as in section 3.3, then for every* $a \in D$

$$x = a \wedge \forall e \forall b(change\_relation_x(a, e, b) \to Lasts(\neg e, d)) \to Lasts(x = a, d)$$

Informally, if during an interval no event occurs that can change the value of a variable, the variable keeps its value.

Example 11. For the GRC the previous lemma ensures the following property:

COROLLARY 1. $gate = closed \wedge Lasts(\neg up, pt) \to Lasts(gate = closed, pt)$
meaning that, if the gate is closed and for the next *pt* time units no *up* command will be issued then the bar will stay in the *closed* position. ♢

4.4 The pretty printer

Using a semantic encoding, current time shows up in the proofs, whereas a peculiarity of TRIO is to hide it. To restore the original TRIO syntax we have built a pretty printer that can be activated during proofs, and substitutes the original PVS display routines. The pretty printer takes over the printing of the sequent, restructuring formulas to remove undesired information.

Hiding the current time . In this first, trivial example, the user should prove that `alpha`, a time dependent predicate, is true now (i.e. at time 0). Indeed in the original goal is `alpha(0)`. The pretty printer hides the current time `(0)`, and reports that `alpha` is stated at current time in the first row of the sequent (`>>> AT: 0`)[14]:

without pretty printer	with pretty printer
`EX1 :`	`EX1 :`
	`>>> AT: 0`
`|-------`	`|-------`
`{1} alpha(0)`	`{1} alpha`

[14]The pretty printer shows at which time instant the sequent is stated. For instance, `>>> AT: d` means that the sequent is stated (and must be proven) at time instant *d*.

Hiding explicit time. For formulas stated not at the current time, but at a given distance (say t!1 time units[15]) in the future or in the past, the pretty printer substitutes the explicit time value in the formula with the proper *Dist* operator:

without pretty printer	with pretty printer
`EX1 :`	`EX1 :`
	`>>> AT: 0`
`|-------`	`|-------`
`{1} alpha(t!1)`	`{1} Dist(alpha,t!1)`

Temporal translation. Sometimes proofs are derived more easily adopting a point of view different from the current instant (at time 0), as shown in the previous examples, but another instant in the future or in the past. In this case the pretty printer allows a temporal translation of every formula in the sequent, simplifying it. The values of the translation is automatically computed or set by users using a particular command. In the following example the formulas stated at `t!1` are correctly shown without any time (being `t!1` the current time) and formulas stated at `t!1 + pt!1` are shown with a `Dist(...,pt!1)`

without pretty printer	with pretty printer and temporal translation
`EX1 :`	`EX1 :`
	`>>> AT: t!1`
`{-1} UpToNow(gate = closed)(t!1)`	`{-1} UpToNow(gate == closed )`
`{-2} up(t!1)`	`{-2} up`
`{-3} Lasts_ee(NOT down, pt!1)(t!1)`	`{-3} Lasts_ee(NOT down, pt!1)`
`|-------`	`|-------`
`{1} gate(t!1 + pt!1) = open`	`{1} Dist(gate == open, pt!1)`

The pretty printer can be activated using the command `(pprint on)` or deactivated using `(pprint off)`. Temporal translation can be set by the command `(pprint at x)`, where `x` is any real value.

4.5 Proofs for our case study

We briefly outline the proofs we constructed in our system for the safety and utility properties of the GRC. The safety property, simply stating that if the number of trains in the critical region I is greater than 0, then the bar is closed, is formalized by the following theorem:

THEOREM 1. *Safety:*
$CTI > 0 \rightarrow gate = closed$

The first lemma used to prove safety, is the following:

LEMMA 2. *gate_will_close :*
$down \wedge Lasts_{ie}(\neg up, pt) \wedge pt > \gamma \rightarrow Futr(gate = closed, pt)$

stating that if a *down* command is issued now and no *up* command is issued in the future for an interval lasting at least γ time units (the time that the bar takes to

[15]Notice that in PVS system-generated skolem constants are composed of an alphabetical string followed by an exclamation mark and a natural number, e.g., `t!1`.

reach the closed position), then at the end of that interval the bar will be closed. The lemma is proved using temporal induction and case analysis. Case analysis is done on the current state of the bar: *closed*, *open*, *movingUp* or *movingDown*. If the bar is already closed, then the lemma immediately follows from Corollary 1. In every other case the bar is either moving down or it starts moving down and after at most γ it will be closed, and afterwards it will stay closed because no up command is issued.

The other lemma used to prove safety is

LEMMA 3. *$CTPI_\gamma$_is_safe :*
$CTI > 0 \rightarrow Lasted_{ie}(CTPI_\gamma > 0, \gamma)$

meaning that if there is a train in I, then $CTPI_\gamma$ has been positive for at least γ time units. For this reason $CTPI_\gamma$ becomes greater than 0 at least γ time units earlier than CTI . Therefore $CTPI_\gamma$ can be safely used to foresee the number of trains in I for γ time units. This lemma is proved using the definition of $CTPI_\gamma$ and *CTI* (see page 19) and the temporal relations between *RI*, *II*, and *IO* (see page 27) and simply applying decision procedures for equalities and linear inequalities.

From these two lemmas, safety can be proved as follows:

(1) assume $CTI > 0$ (by hypothesis)
(2) then $CTPI_\gamma$ has been greater than 0 for at least γ time units (thanks to *$CTPI_\gamma$_is_safe*)
(3) $CTPI_\gamma$ became greater than 0 at least γ time units ago and then a *down* command was issued (thanks to the definition of *down*) and no *up* command has been issued, because $CTPI_\gamma$ has been greater than zero afterward.
(4) a *down* command was issued at least γ time units ago, then the bar is *closed* (thanks to *gate_will_close*)

Hence if $CTI > 0$ then the bar is *closed.*

Besides safety, the GRC should ensure the second user requirement, *utility*, stated as follows:

THEOREM 2. *Utility:*

$$Lasted(\neg CTI > 0, \gamma) \wedge Lasts(\neg CTI > 0, \gamma + d_M - d_m) \rightarrow gate = open$$

While safety asserts when the bar must be closed, utility specifies when it should be open. The bar has to be open under two conditions, modeled by the two conjuncts in the premise of the formula: no train has certainly been in the region I for γ time units and no train will be in region I for $\gamma + d_M - d_m$. The first condition corresponds to the γ time units necessary to raise the bar from the closed position after train exit from region *I* has been detected by sensors. In the second condition the time constant $\gamma + d_M - d_m$ accounts for both the time γ necessary to lower the bar, and the maximal advance in lowering the bar with respect to actual train entrance in region I due to the pessimistic assumption of maximal train speed (if the train is traveling at the lowest possible speed, then the bar is lowered $d_M - d_m$ time units in advance).

The proof of utility is similar to that for safety. It exploits a lemma *gate_will_open* (symmetric to *gate_will_close*) stating the conditions under which the gate will be open. Another lemma $CTPI_\gamma$*_is_useful,* taking the place of $CTPI_\gamma$*_is_safe,* binds $CTPI_\gamma$ with *CTI.* Other auxiliary minor lemmas are proved through intensive use of case analysis and temporal induction.

Notice that, with reference to the equation (1), *safety* and *utility* properties are clearly part of user requirements. Indeed proving these properties means proving that equation (1) is true, i.e. user requirements follow from the model of the environment and of the system (with sensors and actuators).

Note 7. *Comparison with the previous approach.* Comparing our current approach with the previous one [Alborghetti et al. 1997], proofs are now significantly simpler and shorter. We need fewer axioms (7 against 20) and intermediate lemmas (8 against 36). But the most notable and meaningful improvement is in effort for deriving the proof of the safety and utility property. To evaluate the effort we have adopted a weighted measure for each command, counting a complex command as the number of atomic actions it requires; for example `(ASSERT)` would count as 1, while `(TRIO-LEMMA 'futr_interv_induction' 'gate==closed')` would count as 3. A total effort of 380 has been necessary to prove *safety* (against 1433 for the previous approach), and 497 (it was 2165) for utility, and 51 (it was 842) for some auxiliary lemmas. The total effort is reduced to 928 against 4440, with an improvement of a factor near to 5.

4.6 Methodological Remarks

Although the three components (theories, pretty printer, strategies) are designed to work together, they might be used separately. This derives from our choice to adopt an open, incremental approach, where the user should be encouraged to using notations, constructs and tools as long as they fit his/her mentality and to the extent to which they can deal effectively with the real needs for modeling and analyzing.

Depending on his/her habits and preferences, a user might choose to use definitions of derived entities directly in PVS, without adopting the TRIO notation, using explicit current time and therefore with no need to employ the pretty printer.

Conversely, a user might prefer to work in pure TRIO with no additional derived entities, and therefore model the system to be developed only through plain TRIO axioms, and use the combination of the encoding, the pretty printer, and PVS as a sort of general purpose TRIO theorem prover.

Also, some of the notions or (fragments of) theories introduced in this paper could be usefully embedded into existing methods, such as SCR [Heitmeyer et al. 1996] or LUSTRE [Halbwachs et al. 1992], to provide a full formalization of some elements that are not completely formalized or to enrich their framework through additional concepts and modeling constructs or reasoning schemes.

We believe that our gradual and modular approach suits better the needs of the user: it allows a more gradual learning and makes our tool more usable. On the contrary, most other comparable encodings are monolithic and supposed to be used as a whole. Of course the most significant benefits can be obtained from our approach when all components of our framework are used in conjunction.

In the initial phases of system modeling the high-level entities (e.g., events, interval predicates and variables, non-Zenoness) can be used to identify the system components and characterize their basic properties. Some primary items denoting elements of the environment or representing physical quantities may be stated to belong to one of the types introduced in Section 2 (such types would be predefined for the user). Some other items may be defined as derived, by means of suitable axioms stating their logical equivalence to some TRIO formula, as shown in Section 3.2. Then, proving that these derived entities effectively belong to some types, like e.g. point or interval variables, is a useful means for checking the internal consistency of the model and for performing some preliminary validation activity. In most cases suitable formulas, stating this consistency between potentially conflicting parts of a system model, are generated automatically by PVS in the form of TCC (Type Correctness Conditions) [Rushby 1997] and automatically proved (discharged, in the lexicon of PVS), otherwise they remain as proof obligations to be fulfilled by the user with the interactive assistance of PVS. When these consistency conjectures prove to be false, this typically allows the designer to unveil some subtle flaw in the model that has been overlooked until then.

Once the alphabet of the model is defined, the relations introduced in Section 3 can be used to formalize more dynamic aspects of the introduced items. Counters and special predicates can be used to express constraints concerning the environment (e.g., to define minimal/maximal time elapsing between occurrences of sporadic events, or periodicity); a *change_relation* like that in Definition 10, Section 3.3, or its tabular representation can synthetically model portions of a system in a style based on states and transitions. This approach is particularly suitable for modeling sensors and actuators, devices relatively simple and with reduced amount of memory, which perform limited amounts of computation.

Cause/effect relations and counters can be effectively used to express, directly through ad hoc defined entities or more indirectly through suitable axioms, the internal and external actions performed by the Device Under Construction to reach the goals for which it is designed. This part of the model becomes the design specification of the Device Under Construction and, composed as in equation (1) with the models of the environment, the sensors and the actuators, can be used to prove that the user requirements are indeed satisfied.

In our experience the proof of formula (1) rarely succeeds at the first attempt. Most times difficulties in deriving such proofs are originated by defects in the crucial and most complex components of the comprehensive system model, namely the user requirements and the design specification.

To make such proofs more readable, understandable, and reusable, it is advisable to organize them into a series of lemmas. This facilitates the designer in detecting the cause of a difficulty or impossibility- to prove some conjectures and therefore to trace such difficulties to their cause: conceptual errors, inaccurate or poorly structured formalization of the system model, of the design specification or of the user requirements.

The relation of logical dependency among the lemmas can be represented as a tree or DAG graph: the upper part of such a graph will correspond to high-level global properties from which the overall formula (1) should be derivable rather directly, while the lower parts may correspond to details of a single component or of a limited portion of the model.

It is desirable that proofs be conducted, possibly in a completely automated fashion, by applying systematically a few simple and standard derivation rules, like, e.g., the most common propositional tautologies, simple properties of quantifiers, mathematical induction. Such basic derivation rules may suffice in case it is possible to model all system components totally in terms of the entities and constructs introduced in Sections 2 and 3. Otherwise the proof may require the adoption of ad hoc reasoning schemes. In any case it is important to structure proofs possibly in the same way as models and specifications. This can be obtained, for instance, through the introduction of suitable auxiliary predicates, variables, and intermediate lemmas, so to encapsulate and hide the most complex but immaterial details, and make the derivation of the concluding lemmas as clear and intuitive as possible.

Failed attempts to derive formula (1) lead to adjustments and corrections that eventually allow the designer to obtain a specification of superior quality, from which the development can proceed in a more accurate and less error-prone way. As example of correction of a specification obtained from System Requirement Analysis, we cite [Alborghetti et al. 1997], where a quite subtle but potentially severe error in a specification [Mandrioli et al. 1996] was detected during a failed attempt to derive the user requirements using the method and tool described in the present paper.

5. RELATED WORK

In this section we compare our work with other approaches and formal methods for the analysis of critical systems. We present a brief comparison on the different subjects we have tackled in this paper, mainly: non-Zenoness and finite variability, continuity of interval variables, abstract relations among events, encoding in PVS. Indeed PVS has been widely used by research groups to encode their formal languages and methods. Among approaches using other tools, we cite the encoding of hybrid automata (particularly suitable to model variables with continuous behavior) in STeP [Manna and Sipma 1998]. That approach employs a powerful method mixing deductive proofs and automatic invariant generation.

Non-Zenoness and finite variability. In some formal languages non-Zeno behavior is achieved by enforcing a fixed minimum system delay between two actions (with action we informally mean whatever might change the system state). A fixed minimum delay is implicitly assumed in temporal languages using discrete time (for instance integer) and no simultaneous events (like the simplified version of SREL used in [Yang et al. 1997]). However the same assumption is taken by some other formalisms using continuous time, like timed CSP [Reed and Roscoe 1988] and ASTRAL [Coen-Porisini et al. 1997]. We believe that the assumption of a minimum system delay, which is certainly of practical interest and may appear to be the only solution ensuring finite variability, yields however a complicated theory which also hampers effective abstraction on time.

The other approach, followed by our method and most other languages using real time, prefers to avoid establishing a minimum lower bound on the distance between event occurrences. Some methods, like Hybrid Automata [Henzinger 1996], can allow Zeno behaviors and try to overcome possible problems introducing notions like limits and techniques of regularization [Lygeros et al. 1999]. Other methods introduce an explicit non-Zeno requirement, admitting only finitely variable entities.

In some methods this requirement is also formally defined.

In languages modeling the system behavior as a sequence of events or states (with time associated to every state) [Caspi and Halbwachs 1985; Merritt et al. 1991; Manna and Sipma 1998], finite variability is defined requiring that time "eventually increases above any bound" [Shankar 1993]. This way only a finite number of events (but possibly greater than any fixed value) can occur simultaneously or fall within a given time bound. For a definition of such requirement in PVS see [Shankar 1993].

The same assumption, in a different form, can be found in interval temporal logics, see for instance the Duration Calculus [Chaochen and Hansen 1997] and, for a formal definition (but without using RTGIL itself), RTGIL [Moser et al. 1997]. Note that interval temporal logics deal only with piecewise constant variables. For this kind of variables we give a formal definition of non-Zenoness in TRIO.

An original contribution of our work on this subject is a formal definition of non-Zeno requirement for variables in uncountable domains. Using TRIO itself to define such requirement allows us to translate it in our encoding in PVS[16], and to prove several interesting theorems, and reuse them in proofs of system properties.

Continuity of interval entities. In [Caspi and Halbwachs 1985] continuity of entities is not established, but the operator (called *current*) to access at the variables values is left continuous. Also counters (called *counter*) are left continuous. Right continuous access operator (*lcurrent*) and counters (*lcounters*) are available. In Duration Calculus (DC) [Chaochen and Hansen 1997] continuity is not considered relevant, for the main DC operator, duration or $\int$, does not depend on the value in of variables in single points. Nevertheless the use of left continuous interval variables is suggested, to avoid problems using logical operators between predicates of different continuity as explained in Section 2.6. RTGIL [Moser et al. 1997] simply assumes variables to be right continuous.

Abstract relations among events. The idea of using counters to model relations between events was already presented in [Caspi and Halbwachs 1985]. That work used counters to model relations of precedence among events (that is an event or a set of events must precede or follow another event). Also in [Yang et al. 1997] counters are directly used to specify system requirements (like, for instance, "the number of missiles fired is no more than the number of targets located so far"). The generic requirement that a certain event must follow (or precede) another can be enriched specifying a bounded delay between such events. This kind of requirement is widely (also informally) used. Furthermore it is explicitly modeled (somehow embedded within the language itself) in timed Petri Net [Merlin and Farber 1976] and in MMT timed Automata [Merritt et al. 1991] (whose embedding in PVS is done in [Archer and Heitmeyer 1996; Archer and Heitmeyer 1997b]). We have shown how the use of counters can be extended to specify this relation in a very simple and effective way.

[16]PVS, however, does not have a predefined theory for mathematical analysis (with definitions for limit, analytic function ...). Therefore, the definition of non-Zenoness for variable in uncountable domains using those constructs, could not be directly translated into PVS. Several attempts have been made to define elements of mathematical analysis in PVS [Dutertre 1996]. Nevertheless, as already noted at page 32, we preferred to exploit simple but very general conditions ensuring that encoded variables are non-Zeno.

Encoding in PVS. Since PVS has been proved suitable to encode other logics, many formal real-time languages have been encoded in PVS. An interesting work concerns DC [Skakkebæk and Shankar 1994] and it adopts a semantic encoding (save the encoding of the operator *dur* or $\int$, a sort of integral operator). Moreover that encoding has a very powerful pretty printer with functionalities similar to ours (but a completely different implementation: they modified the parser used by PVS, while we essentially overload printing methods of PVS exploiting the CLOS overloading feature). DC is particularly suitable to express in a concise way complex constraints about duration of systems phenomena, however it is not well suited to express point entities, for instance events.

Many works use directly the logic of PVS to specify and verify real time systems. Some early works have treated real time systems as sequential systems: that is the system discretely changes assuming a sequence of states (containing time). For example in [Shankar 1993] systems evolve step by step and every step has a time associated with it. The same approach has been followed by [Hooman 1994], that extends Hoare triples with timing constraint in every assumption (pre-condition) and commitment (post-condition) pair. This approach is mainly oriented to the verification of sequential systems like, for instance, computer programs. However, an interesting application of this method to a real-time reactive system can be found in [Vitt and Hooman 1996]. Recently PVS has been used also in [Dutertre and Stavridou 1997], where the requirement analysis of a real avionic control system is conducted without relying on any formal language besides PVS; however, state variables used there are inspired to DC and data flows to LUSTRE [Halbwachs et al. 1992] and SIGNAL [Benveniste et al. 1991]. State variables represent the continuous variables of the system, like physical parameters, inputs and outputs, and they are encoded in PVS simply as functions from time to their domain. Data flows model the discrete components of the system, and they are synchronized by a clock. Their encoding is slightly different from that for state variables, and for this reason the authors introduce some conversion functions from a data flow to a state variable.

The main disadvantage of using an encoding of another logic in PVS is that experience with both (PVS and the source logic) is required. Furthermore these systems normally present themselves as a whole: source logic, PVS libraries, and proof strategies are strictly integrated and cannot be used separately. This significantly increases the efficiency of the automatic conduction of proof previously done by hand (as in [Archer and Heitmeyer 1997a; Archer et al. 2000]), but hinders their use by designers with limited expertise in the use of the source logic (even if expert in using PVS). Moreover single components like libraries or code cannot be reused at all, since they are tailored to that particular system.

On the other hand direct specification in PVS of real-time systems suffers, as pointed out in [Dutertre and Stavridou 1997], the lack of guidance in defining entities, writing the specification, and conducting proofs: therefore designers must write their own libraries from scratch, because PVS, like most higher-order logic systems, is not expressly thought to deal with real time. When writing *ad hoc* libraries, designers may obtain a simpler semantic, a clearer encoding of temporal dependency, and a more application-oriented set of strategies, at the price, however, of a lesser degree of generality.

In our work we have tried to combine benefits from both approaches. From TRIO we have taken the clear semantic of its entities like events and interval variables, and natural yet powerful operators. TRIO experts should have no problems to lead proofs in PVS using our encoding. Thanks to the pretty printer useless information (added by the encoding) is hidden. However, our system uses a simple encoding for temporal entities (used by many other approaches) and in this way libraries, proof strategies and pretty printer might be reused by people with no knowledge of TRIO.

6. CONCLUSIONS

In this paper we presented a framework consisting of the following components.

A descriptive notation for system modeling and requirements specification: the real-time temporal logic TRIO, which provides a quantitative notion of time, assumed as the linear set of real number, consistently with classical physics and dynamic system theory, disciplines with which most engineers and mathematicians have some familiarity.

A precise, formal definition, in terms of TRIO axioms, of several high-level notions having a significant relevance in modeling real-world entities, such as events, states, continuity, finite variability, (non)determinism, cause-effect relations.

An encoding of the TRIO logic and of the above mentioned high-level notions, into the powerful, general purpose theorem prover PVS. We exploited the higher-order features of PVS to simplify the encoding and to introduce suitable derived inference rules and proof strategies, based on the original ones of PVS and especially tailored to the proposed framework.

This framework is particularly well suited to supporting System Requirements Analysis, a preliminary activity of crucial importance in the development of highly critical systems. System Requirements Analysis requires modeling the environment together with the Device Under Construction, stating the user requirements and the design specifications, and combining all these to perform an accurate analysis aimed at proving that the system will actually exhibit the desired properties. To assess its actual usability, the framework was applied to model and analyze the Generalized Railway Crossing (GRC) system, a well known and widely adopted benchmark for the study of time- and safety-critical systems, whose timing features proved to be more complex and subtle than those of many industrial applications that we previously specified and analyzed using the TRIO language and its tool environment [Gargantini et al. 1996; Basso et al. 1998]. The results of the GRC case study have been satisfying, with significant improvements with respect to previous exploratory work aimed at investigating the feasibility of the approach [Alborghetti et al. 1997].

The contributions of the present work are both conceptual, technical and methodological.

From the conceptual viewpoint, we perform an in-depth analysis and formalization of notions that are often stated informally or assumed implicitly but are seldom adequately formalized, such as non-Zenoness for several kind of entities, finite variability, cause-effect relations. In doing so we provide precise formal semantics, in terms of a very simple and basic temporal logic like TRIO, to notations having a high relevance in the analysis and design of critical systems, such as tabular timed specifications, events, states, modes, etc.

On the technical side, we implemented our framework as a tool suite inside a widely available, powerful theorem prover such as PVS. The various components of the framework - encoding of the base logic, theories for modeling high level notions and constructs, proof strategies for facilitating analysis and derivations, and pretty printer to increase readability- can be used separately or in conjunction, depending on user needs and preferences. Our choice to actually build tools based on the framework is consistent with the now widespread belief that automated support to analysis and design constitutes the authentic added value that formal methods can provide to the computing community.

From a methodological perspective, our framework provides an effective support to System Requirements Analysis, thus making more amenable this activity, which is of crucial importance for the construction of correct, reliable critical systems, but is often disregarded, being at time considered trivial, infeasible, or unworthy. Our framework offers a predefined set of entities and tools, and it encourages engineers to use them, thus avoiding the continuous reinvention of well-known, deeply studied notions and constructs. For the majority of applications of practical interest, it should support the modeling and analysis of most system aspects, or of their totality, by just applying in a rather straightforward manner predefined notions and constructs. However the framework is open in nature: users can define new high-level notions or ad hoc entities and properties in TRIO, or even work at the level of the (higher-order) logic of PVS. It is obviously to be expected that using ad hoc defined entities results of models and analysis could be less understandable or reusable.

We successfully applied the framework also to several internally-generated examples, and we now believe that it could be usefully exercised on real-life industrial applications, in particular it could effectively support System Requirements Analysis of complex time- and safety-critical systems: a typical application would be in the Hazard Analysis for high-integrity safety related computerized control devices for transportation systems [CENELEC (European Committe for Electrotechnical Standardization) b; CENELEC (European Committe for Electrotechnical Standardization) a].

From a more conceptual side, further work will be addressed to providing a completely satisfying formalization, in terms of PVS theories, of the notion of finite variability: in the most general case we express it in terms of analytic functions, for which no predefined theory exists in PVS. A more accurate conceptual analysis could follow the direction of the work performed by [Dutertre and Stavridou 1997].

REFERENCES

Abadi, M. and Lamport, L. 1994. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems 5*, 16 (sep), 1543–1571.

Alborghetti, A., Gargantini, A., and Morzenti, A. 1997. Providing automated support to deductive analysis of time critical systems. In M. Jazayeri and H. Schauer Eds., *Software Engineering—ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Volume 1301 of *Lecture Notes in Computer Science* (Zurich, Switzerland, Sept. 1997), pp. 211–226. Springer-Verlag.

Archer, M. and Heitmeyer, C. 1996. TAME: A specialized specification and verification system for timed automata. In A. Bestavros Ed., *Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)* (Washington, DC, Dec. 1996), pp. 3–6.

Archer, M. and Heitmeyer, C. 1997a. Human-style theorem proving using PVS. In E. Gunter and A. Felty Eds., *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, Volume 1275 of *Lecture Notes in Computer Science* (Murray Hill, NJ, Aug. 1997), pp. 33–48. Springer-Verlag.

Archer, M. and Heitmeyer, C. 1997b. Verifying hybrid systems modeled as timed automata: A case study. In O. Maler Ed., *Proceedings of the International Workshop on Hybrid and Real-Time Systems (HART'97)*, Volume 1201 of *Lecture Notes in Computer Science* (Grenoble, France, March 1997), pp. 171–185. Springer-Verlag.

Archer, M., Heitmeyer, C., and Riccobene, E. 2000. Using TAME to prove invariants of automata models: Two case studies. In *Third ACM Workshop on Formal Methods in Software Practice (FMSP'00)* (2000).

Basso, M., Ciapessoni, E., Crivelli, E., Mandrioli, D., Morzenti, A., Ratto, E., and San Pietro, P. 1998. A logic-based approach to the specification and design of the control system of a pondage power plant. In C. Tully Ed., *Improving Software Practice: Case Experience*, Series in Software Based Systems, pp. 79–96. Wiley.

Benveniste, A., Guernic, P. L., and Jacquemot, C. 1991. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming 16*, 2 (Sept.), 103–149.

Capobianchi, R., Coen-Porisini, A., Mandrioli, D., and Morzenti, A. 1999. A framework architecture for supervision and control systems. In *ACM Computing Surveys*.

Caspi, P. and Halbwachs, N. 1985. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica 22*, 595–627.

CENELEC (European Committe for Electrotechnical Standardization). *Railway applications - Safety related electronic systems for signalling* (Ref. No. prEN 50129 ed.). CENELEC (European Committe for Electrotechnical Standardization).

CENELEC (European Committe for Electrotechnical Standardization). *Railway applications - Software for railway control and protection systems* (Ref. No. prEN 50128 ed.). CENELEC (European Committe for Electrotechnical Standardization).

Chaochen, Z. and Hansen, M. R. 1997. Duration calculus: Logical foundations. *Formal Aspects of Computing 9*, 3, 283 – 330.

Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., and Morzenti, A. 1999. From formal models to formally-based methods: an industrial experience. *ACM TOSEM - Transactions On Software Engineering and Methodologies 8*, 1, 79–113.

Coen-Porisini, A., Ghezzi, C., and Kemmerer, R. A. 1997. Specification of realtime systems using ASTRAL. Technical Report TRCS96-30 (Jan.), University of California, Santa Barbara. Computer Science.

Courant, R. and Fritz, J. 1974. *Introduction to Calculus and Analysis*. Springer.

Dutertre, B. 1996. Elements of mathematical analysis in PVS. In J. von Wright, J. Grundy, and J. Harrison Eds., *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, Volume 1125 of *Lecture Notes in Computer Science* (Turku, Finland, Aug. 1996), pp. 141–156. Springer-Verlag.

Dutertre, B. and Stavridou, V. 1997. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering 23*, 5 (May), 267–278.

Felder, M., Mandrioli, D., and Morzenti, A. 1994. Proving properties of real-time systems through logical specifications and Petri net models. *IEEE Transactions on Software Engineering 20*, 2 (Feb.), 127–141.

Felder, M. and Morzenti, A. 1994. Validating real-time systems by history-checking TRIO specifications. *ACM Transactions on Software Engineering and Methodology 3*, 4 (Oct.), 308–339.

Gargantini, A., Liberati, L., Morzenti, A., and Zacchetti, C. 1996. Specifying, validating and testing a traffic management system in the TRIO environment. In *Compass'96: Eleventh Annual Conference on Computer Assurance* (Gaithersburg, Maryland, 1996), pp. 65. National Institute of Standards and Technology.

Gargantini, A., Mandrioli, D., and Morzenti, A. 1999. Dealing with zero-time transitions in axiom systems. *INFCTRL: Information and Computation (formerly Information and Control) 150*, 119–131.

Gargantini, A. and Morzenti, A. 1999. Automated deductive analysis of time critical systems based on methodical formal specification. Technical Report 50, Dipartimento di Elettronica e Informazione, Politecnico di Milano.

Ghezzi, C., Mandrioli, D., and Morzenti, A. 1990. TRIO: A logic language for executable specifications of real-time systems. *The Journal of Systems and Software 12*, 2 (May), 107–123.

Halbwachs, N., Lagnier, F., and Ratel, C. 1992. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering 18*, 9 (Sept.), 785–793.

Hansen, K., Ravn, A., and Stavridou, V. 1998. From Safety Analysis to Software Requirements. *IEEE Transactions on Software Engineering 24*, 7, 573–584.

Heitmeyer, C., Jeffords, R., and Labaw, B. 1996. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology 5*, 3 (July), 231–261.

Heitmeyer, C. and Mandrioli, D. 1996. *Formal Methods for Real-Time Computing*, Volume 5 of *Trends in Software*. Wiley.

Heitmeyer, C. L., Jeffords, R. D., and Labaw, B. G. 1993. A benchmark for comparing different approaches for specifying and verifying real-time systems. In *Tenth International Workshop on Real-Time Operating Systems and Software* (May 1993).

Henzinger, T. 1996. The theory of hybrid automata. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science* (1996), pp. 278–292. IEEE Computer Society Press.

Hooman, J. 1994. Correctness of real time systems by construction. In H. Langmaack, W.-P. de Roever, and J. Vytopil Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 863 of *Lecture Notes in Computer Science* (L'ubeck, Germany, Sept. 1994), pp. 19–40. Springer-Verlag.

Jeffords, R. D. 1995. An approach to encoding the TRIO logic in PVS. Technical report, Naval Research Laboratory.

Komoda, N., Haruna, K., Kaji, H., and Shinozawa, H. 1981. An innovative approach to system requirements analysis by using structural modeling method. In *5th International Conference On Software Engineering* (Los Alamitos, Ca., USA, March 1981), pp. 305–313. IEEE Computer Society Press.

Koymans, R. 1989. *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* Ph. D. thesis, Eindhoven University of Technology, Netherland.

Leveson, N. G. 1995. *Safeware: System Safety and Computers.* Addison-Wesley.

Luenberger, D. G. 1979. *Introduction to Dynamic Systems: Theory, Models, & Applications.* Wiley.

Lygeros, J., Johansson, K., Sastry, S., and Egerstedt, M. 1999. On the existence of executions of hybrid automata. In *IEEE Conference on Decision and Control* (Phoenix, AZ, Dec 1999).

Mandrioli, D., Morasca, S., and Morzenti, A. 1995. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems 13*, 4 (Nov.), 365–398.

Mandrioli, D., Morzenti, A., Pezze', M., San Pietro, P., and Silva, S. 1996. A petri net and logic approach to the specification and verification of real-time systems. In C. Heitmeyer and D. Mandrioli Eds., *Formal Methods for Real-Time Computing*, Volume 5 of *Trends in Software*, Chapter 5. Wiley.

MANNA, Z. AND SIPMA, H. 1998. Deductive verification of hybrid systems using STeP. In S. SASTRY AND T. A. HENZINGER Eds., *Hybrid Systems: Computation and Control*, Number 1386 in LNCS. Springer-Verlag.

MERLIN, P. M. AND FARBER, D. J. 1976. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Transactions on Communication COM-24*, 1036–1043.

MERRITT, M., MODUGNO, F., AND TUTTLE, M. R. 1991. Time-constrained automata (extended abstract). In J. C. M. BAETEN AND J. F. GROOTE Eds., *CONCUR '91: 2nd International Conference on Concurrency Theory*, Volume 527 of *Lecture Notes in Computer Science* (Amsterdam, The Netherlands, 26–29 Aug. 1991), pp. 408–423. Springer-Verlag.

MORZENTI, A. AND SAN PIETRO, P. 1994. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology 3*, 1 (Jan.), 56–98.

MOSER, L. E., RAMAKRISHNA, Y. S., KUTTY, G., MELLIAR-SMITH, P. M., AND DILLON, L. K. 1997. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology 6*, 1 (Jan.), 31–79.

OWRE, S., RUSHBY, J., SHANKAR, N., AND VON HENKE, F. 1995. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering 21*, 2 (Feb.), 107–125.

PARNAS, D. L. 1992. Tabular representation of relations. Technical Report 260, CRL, McMaster University, Canada.

PARNAS, D. L. AND MADEY, J. 1995. Functional documents for computer systems. *Science of Computer Programming 25*, 1 (Oct.), 41–61.

PAULSON, L. C. 1994. *Isabelle: A Generic Theorem Prover.* Springer-Verlag LNCS 828.

REED, G. M. AND ROSCOE, A. W. 1988. A timed model for communicating sequential processes. *Theoretical Computer Science 58*, 1-3 (June), 249–261.

RUSHBY, J. 1997. Subtypes for specifications. In M. JAZAYERI AND H. SCHAUER Eds., *Software Engineering—-ESEC/FSE '97: Sixth European Software Engineering Conference and Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Volume 1301 of *Lecture Notes in Computer Science* (Zurich, Switzerland, Sept. 1997), pp. 4–19. Springer-Verlag.

RUSHBY, J., OWRE, S., AND SHANKAR, N. 1998. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering 24*, 9 (Sept.), 709–720.

SHANKAR, N. 1993. Verification of real-time systems using PVS. In C. COURCOUBETIS Ed., *Computer-Aided Verification, CAV '93*, Volume 697 of *Lecture Notes in Computer Science* (Elounda, Greece, June/July 1993), pp. 280–291. Springer-Verlag.

SHANKAR, N., OWRE, S., RUSHBY, J. M., AND STRINGER-CALVERT, D. W. J. 1998. *PVS Prover Guide.* Menlo Park, CA: Computer Science Laboratory, SRI International.

SKAKKEBÆK, J. U. AND SHANKAR, N. 1994. Towards a Duration Calculus proof assistant in PVS. In H. LANGMAACK, W.-P. DE ROEVER, AND J. VYTOPIL Eds., *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Volume 863 of *Lecture Notes in Computer Science* (Lübeck, Germany, Sept. 1994), pp. 660–679. Springer-Verlag.

VITT, J. AND HOOMAN, J. 1996. Assertional specification and verification using PVS of the steam boiler control system. In J.-R. ABRIAL, E. BOERGER, AND H. LANGMAACK Eds., *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Volume 1165 of *Lecture Notes in Computer Science* (1996), pp. 453–472. Springer-Verlag.

YANG, J., MOK, A. K., AND WANG, F. 1997. Symbolic model checking for event-driven real-time systems. *ACM Transactions on Programming Languages and Systems 19*, 2 (March), 386–412.

ZAVE, P. AND JACKSON, M. 1997. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology 6*, 1 (Jan.), 1–30.