POLITECNICO DI MILANO
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA
Dipartimento di Elettronica e Informazione

Requirements Specification and Analysis for Real-time Systems

Ph.D. Thesis of:
Angelo Gargantini

Supervisor:
Prof. Dino Mandrioli

Co-supervisor:
Prof. Angelo Morzenti

Supervisor of the Ph.D. Program:
Prof. Carlo Ghezzi

XII ciclo

POLITECNICO DI MILANO
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA
Dipartimento di Elettronica e Informazione

Specifica, convalida e verifica dei requisiti di sistemi in tempo reale mediante metodi formali

Tesi di dottorato di:
Angelo Gargantini

Relatore e Tutore:
Prof. Dino Mandrioli

Correlatore:
Prof. Angelo Morzenti

Coordinatore del Dottorato:
Prof. Carlo Ghezzi

XII ciclo

to my family

Sommario

Computer e sistemi basati su computer svolgono un ruolo sempre piú importante nella nostra società anche in applicazioni critiche per la sicurezza. Esempi di sistemi critici per la sicurezza includono sistemi per il controllo di mezzi di volo, sistemi medicali, sistemi per il controllo di impianti chimici. Dal loro corretto comportamento spesso dipendono molte vite umane ed eventuali loro malfunzionamenti possono causare gravi perdite economiche e danni irreparabili all'ambiente [Lev95]. A questi sistemi viene richiesta (talvolta anche tramite opportune certificazioni) una affidabilità incomparabilmente superiore a quella usualmente fornita da sistemi o computer che svolgano operazioni per nulla critiche (come la video scrittura o strumenti di produttività individuale).

Per questa ragione i sistemi critici per la sicurezza, con i loro requisiti (cioè le proprietà desiderate) e il loro funzionamento, vanno documentati, studiati e analizzati ancora prima che possono essere implementati e utilizzati in applicazioni reali. Il fine principale di queste attività è quello di conoscere meglio il sistema dandone una descrizione piú precisa possibile, acquisire la fiducia nella correttezza di questa descrizione o modello e scoprire eventuali errori il prima possibile, dato che la correzione di errori nelle fasi successive è di gran lunga piú costosa.

In questi ultimi anni si è sviluppata un particolare ramo dell'ingegneria, detta *ingegneria dei requisiti*, che si occupa proprio dei metodi per documentare, specificare, e analizzare i sistemi e i loro requisiti. Questo compito è particolarmente arduo per sistemi *embedded* poichè non sono semplici dispositivi isolati chiusi in una scatola, che effettuano semplici calcoli numerici: sono invece sistemi integrati nell'ambiente che devono mandare dei segnali o tenere sotto controllo altre parti del sistema. Per svolgere correttamente la loro attività devono interagire con l'ambiente per mezzo di sensori e attuatori, cercando di mantenere l'ambiente in uno stato sicuro. In genere sono sistemi *reattivi*, cioè sistemi che reagiscono a certi ingressi cambiando il loro stato interno e agendo sull'esterno mediante attuatori. Inoltre spesso questi sistemi sono anche sistemi in *tempo reale*, cioè sistemi cui correttezza dipende non solo dalla loro reazione a certi stimoli, ma anche dai tempi con cui questa reazione avviene. Il tempo svolge quindi un ruolo fondamentale e i metodi che si usano per analizzare questi sistemi devo adeguatamente trattare gli aspetti temporali.

Per esempio un sistema di controllo per un passaggio a livello (vedi Figura 1) è un sistema integrato, in tempo reale e reattivo: è composto da un (micro)computer che interagisce con l'ambiente (la strada, i binari e le sbarre) mediante sensori e attuatori; a partire dai dati rilevati mediante i sensori, manovra le sbarre mediante gli attuatori, in modo che non si verifichi la situazione pericolosa, che il treno stia passando e le sbarre siano alzate.

In questi anni la comunità scientifica ha proposto per la specifica e l'analisi di sistemi, alcuni metodi detti *formali* perché si appoggiano su formalismi matematici o logici con una semantica chiara (formale). Il capitolo 2 ha proprio come oggetto i metodi formali, con particolare riguardo ai metodi formali per sistemi real-time.

I metodi formali includono una notazione o linguaggio per descrivere il sistema in modo chiaro, preciso e non ambiguo, oltre ad alcune linee guida per la loro applicazione e metodologie per l'analisi delle specifiche così scritte.

Il primo uso dei metodi formali consiste nella specifica formale, cioè nel descrivere il sistema e il suo comportamento voluto in modo chiaro e dettagliato (però anche astratto rispetto dettagli ininfluenti che dipendano dall'implementazione). Le specifiche formali così ottenute possono essere usate come documentazione e come mezzo di comunicazione tra il progettista

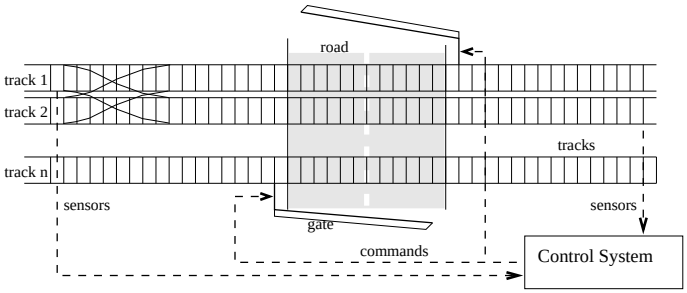


Figure 1: esempio del passaggio a livello

e l'implementatore, tra cliente e fornitore o come base per una eventuale certificazione. Il linguaggio per la specifica deve essere chiaro e con una semantica ben precisa, tuttavia deve poter adattarsi ai sistemi che deve descrivere e deve poter essere usato da persone non esperte del linguaggio stesso: dovrebbe quindi far uso di concetti familiari e intuitivi (come stati, eventi ....). Per sistemi in tempo reale è anche importante che il tempo venga trattato in modo chiaro e preciso ma anche in modo naturale, simile al nostro modo di ragionare. Nei capitoli 3 e 4 mostreremo un metodo per specificare i sistemi in tempo reale che cerca di soddisfare tutti questi requisiti.

La specifica formale permette l'analisi formale (cioè basata sul ragionamento formale) per scoprire errori nel progetto, inconsistenze, ambiguità e incompletezze od/ed eventualmente per dimostrarne invece la correttezza.

Esistono molti tipi di analisi: alcune (dette *leggere*) mirano a controllare che la specifica sia internamente consistente e ben formata (ad esempio, correttezza dei tipi, assenza di casi mancanti, assenza di non-determinismo non voluto). Alcune ricerche [HJL96] hanno mostrato come questo tipo di analisi è utile e utilizzabile in casi reali.

Un'analisi più completa ma anche più difficile, consiste nella verifica formale (logica) di proprietà del sistema a partire dalla specifica: in questo caso si vuole dimostrare che il sistema come specificato soddisfi i requisiti (in genere di sicurezza, o di prestazione) voluti. Lo scopo della verifica si può riassumere in modo molto astratto [ZJ97] con la seguente formula:

$$\text{ModelloAmbiente} \wedge \text{ModelloSensori} \wedge \text{ModelloAttuatori} \wedge \text{SpecificheProgetto} \rightarrow \text{Requisiti}$$

Dalla specifica del sistema e di come questo interagisce con l'ambiente, si può cioè dimostrare che i suoi requisiti sono soddisfatti.

E' ampiamente riconosciuto che le attività di scrittura dei modelli e la loro analisi devono essere supportate da strumenti automatici, per permettere di trattare sistemi complessi, per eventualmente ottenere una verifica automatica dell'analisi che si vuole portare a termine e per certificare i risultati ottenuti. Esperienze in questo campo hanno mostrato come molti errori fossero ancora presenti in specifiche formali scritte senza supporto di strumenti; questi errori sono stati scoperti solo successivamente, usando strumenti che fossero in grado di supportare qualche tipo di analisi (anche semplicemente controllando l'uso dei nomi, l'uso consistente dei tipi, e altre analisi semplici). Esempio classico sono le dimostrazioni di proprietà svolte a mano e cui dimostrazioni sono state poi controllate mediante strumenti: si sono trovati molti errori, assunzioni sbagliate o mancanti, applicazione inaccurata delle regole per l'inferenza o semplicemente errori di scrittura. Il capitolo 5 di questa tesi discuterà proprio la verifica dei requisiti per specifiche scritte usando la metodologia introdotta al capitolo 3 e il suo supporto mediante strumenti automatici.

Oltre che per la specifica e la verifica di proprietà, il modello formale astratto così ottenuto può essere usato in modi diversi: ad esempio per la validazione mediante simulazione. La validazione mediante simulazione o mediante history checking [FM94] consiste nel controllare che il comportamento del sistema previsto dalle specifiche soddisfi effettivamente i bisogni dell'utente

finale, cioè che non siano permessi comportamenti non voluti e che invece siano modellati i comportamenti desiderati.

La specifica formale può inoltre essere usata successivamente come guida per implementare il sistema concretamente nelle sue parti. Nel capitolo 6 vedremo come si può progettare un sistema ottenendo l'implementazione da specifiche ad alto livello, applicando delle regole di raffinamento successivo.

Un'altro uso è quello della generazione di casi di test a partire da specifiche formali [GH99, GLMZ96], cioè la costruzione di possibili scenari o storie che modellino il comportamento del sistema così come è stato specificato, e il loro successivo utilizzo per controllare che il sistema implementato abbia gli stessi comportamenti.

Contents

Chapter 1

Introduction

Computers and computer based systems play an important role in our society and in our daily life, often also in safety-critical applications. Safety-critical applications include weapons systems, flight control systems, medical systems, and plant control systems. Often human lives depend on their correct behavior, and possible faults can cause economic losses and irreparable environmental damages [Lev95]. These systems must provide the highest level of assurance and reliability (often to be assessed by standard certifications), and must be treated in a completely different manner than no critical applications (like for example, word processing and personal computing).

For this reason, systems of this kind, together with their requirements and desired properties and features, and their behavior, must be well documented, deeply studied and analyzed even before they might be implemented or used in real applications. The main goal of this activity is to gain a deeper understanding of the system, to become confident in the correctness of the proposed description and model, and discover possible errors or faults as early as possible, since it is widely acknowledged that the cost of correcting specification errors is order of magnitudes higher in the later stages of the life cycle of the system (like during testing or even during its normal functioning).

Recently a new engineering branch has gained interest in both academia and industry, the so called *requirements engineering*, devoted to studying and promoting methods for documenting and analyzing systems and their requirements. This activity is particularly difficult for critical systems, since computer-based critical systems are not simple devices confined inside a box, just performing some computation tasks: they are usually *embedded* systems in charge of supervision, signaling, or control tasks. To reach their intended purpose they must interact with the 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 are typically *reactive* systems, for they reach this goal by reacting to changes in monitored quantities, computing and updating internal state variables, acting on physical entities through their actuators. Moreover they often are *real-time* systems, thus systems which must not only be functionally correct (produce the expected results), but must also produce those results within specified temporal bounds. Often they are requested to act before a strict deadline (requirements like "in 10 seconds perform this task"), but in many situations they must not act too soon (e.g. "do this action after 5 seconds"). Moreover time regards not only the computation of outputs, but also the monitoring of the inputs: in requirements like "if the pressure has dropped too *fast*, then do this action" time appears in the monitored quantities. Safety-critical real-time systems are the subject of this thesis.

For instance a control system for a Railway Crossing (Figure 1.1) is an embedded, reactive, and real-time system, compound of a (micro)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. The system must prevent the situation in which bars are open and trains crossing. It must consider the inputs from the sensors, decide when to close the bars (not too late), and finally decide when to open the bar (not too soon).

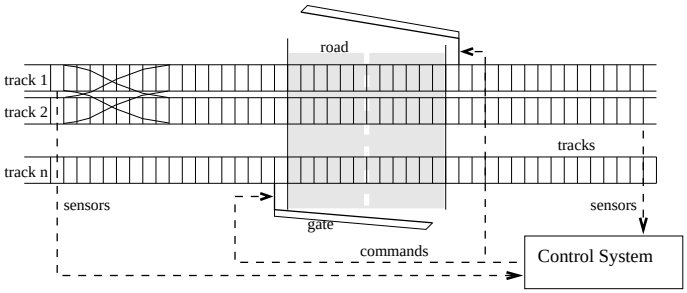


Figure 1.1: Railway crossing example

For modeling and analyzing such systems the scientific community has proposed for many years as promising and efficient means, particular methods called *formal* methods, based on mathematical or logical languages with a precise and clear (formal) semantics. We will present some formal methods and their main peculiar features in the chapter 2, with particular attention towards formal methods dealing with time and temporal aspects and suitable to be applied to real-time systems.

Formal methods include a notation with a well-defined syntax and semantics, able to describe a system, and likely its components, its behavior and its requirements: this description or model is called *specification*. Such methods often give also some guidelines, procedures or advises for their own application, and most important, they propose some techniques or methods for analyzing specifications written in the proposed notation.

The most traditional (and also most accepted) use of formal methods is the formal specification. Through formal specification the designer is able to clearly describe the system, define its behavior and how it interacts with the environment and state system requirements. This specification has the advantage of a clear and unambiguous semantics and it might be as precise as the designer wants (however it should be also abstract and avoid implementation dependent details). Formal specifications might be used as documents (or contracts) between customer and supplier, designer and implementor and might be used to obtain required certifications. Although the notation must be clear and well-defined, it should be also friendly enough to be used by people not expert in the language itself: it should use familiar and intuitive concepts (like events, states ...) and constructs (e.g. graphical, tabular, or mathematical). Moreover for real-time systems, a special attention must be devoted 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. Furthermore, since timing aspects are quite relevant, 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 crosses the road, the bar must have been completely lowered for 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 system analysis. In chapter 3 we will present a method for the specification of real-time systems, that provides all the above mentioned peculiar features.

Another use of formal methods in which the scientific community has spent much research effort is the formal analysis. In fact formal specification enables formal analysis, based on formal reasoning, in order to uncover as many errors as possible, improve the quality of the specification, and formally prove its correctness.

Many kinds of analysis have been proposed: some, called *light analysis*, check the well-formedness of specifications, looking for internal inconsistencies (for example inconsistent use of names and types, missing cases, undesired non determinism). Some methods [HJL96] strongly support this kind of analysis, that can be automatically supported and successfully applied to real world

(complex) cases.

Another kind of analysis is formal property verification, i.e. the logical derivation of requirements of the specified system. This kind of derivation is harder to perform and normally still requires human interaction. In a very abstract logical setting, this activity consists of analyzing, validating, and eventually proving the following implication [ZJ97]:

$$\mathsf{EnvModel} \wedge \mathsf{SensorModel} \wedge \mathsf{ActuatorModel} \wedge \mathsf{DesignSpec} \rightarrow \mathsf{UserReq}$$

i.e., the user requirements are ensured if the system 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.

It's widely recognized that 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. Past experiences have shown that many errors were still present in formal specifications written without tool support, and such errors were uncovered only using tools capable to perform some sort of analysis (also simply name checking, type consistency checking or other similar simple analysis). Most notable examples are many properties that were proved by hand and whose proofs were later checked by tools and discovered wrong, because of false or missing assumptions, or inaccurate application of inference rules or simply typing errors. We will return on this subject in Chapter 5, where we discuss property verification and its automatic support, of specifications written following the method presented in Chapter 3.

Besides formal analysis or verification, formal specifications can be used in many other ways: different uses have been devised in recent years, particularly appreciated by people from industry and new ones are continually proposed, trying to achieve a better reuse and improve the overall cost-effectiveness. For example specifications can be validated by simulation. Validation by means of a simulator or history checking [FM94] consists in checking possible behaviors of the specified system (often in a interactive way and sometimes thanks of a prototype) and being sure that they meet users' needs. Only correct behaviors must be allowed, while wrong or dangerous ones must be banned.

Formal specifications are useful also in the later stages of the system life-cycle: during the implementation phase they can act as guidelines, in order to speed up the first release and improve the quality of the final product. In chapter 6 we will present a method to derive an implementation of a real time system, from an abstract specification, with the assurance that temporal properties are preserved.

Formal specifications might be useful also during testing phase, since from them test cases can be derived. Several techniques [GH99, GLMZ96] have been proposed: thanks to these methods the designer can build or generate possible histories of the specified system and feed them in the implementation to check that behaviors do not differ.

Acknowledgments

I thank friends, Ph.D. fellows, office- and train-mates for their affectionate company during these years. For their help and loving guidance I'm particularly grateful to those I have worked with: Angelo Morzenti, Dino Mandrioli, Connie Heitmeyer, and Elvinia Riccobene. Thanks especially to Angelo Morzenti, my teacher and adviser. Thank you !

Chapter 2

The temporal logic TRIO

In this chapter we briefly present a survey over some formal methods, their main features and their capability to deal with real-time. After a brief introduction in which we argument a possible classification, we present some formal methods, describing in details time Petri Nets (TPNs). Then in section 2.2 we present TRIO, a temporal logic particular suitable to specify and analyze real-time systems. In section 2.3 we introduce the dual language approach and in section 2.3.1 we show how it can be applied combining TRIO with an operational method like time Petri Nets.

2.1 Survey on Formal Methods for Real-Time

The number of formal methods is continually increasing. New and more powerful ones are proposed. To date as November 1999, the WWW page of the formal methods virtual library (http://archive.comlab.ox.ac.uk/formal-methods.html) lists 81 individual formal languages, tools or methods. In such great variety of notations, it's even difficult to suggest a classification and several taxonomies have been proposed and are equally acceptable. However, keep in mind that a classification has often only the goal of clarifying the terms and some concepts, since every split that can be made will find a method that does not fit in that classification. We propose, explain, and use two main classifications: the first one refers to the style and the intent of the modeling: we can distinguish *operational* and *descriptive* methods. The second classification refers to the notation used: *graphical* or *logic and algebra based* languages.

- *operational* methods tend to describe the system as it works, normally modeling the system as a (finite or even infinite) set of states and transitions. An example are finite state machines.
- *declarative* methods prefer to model system properties or behavior using logic formula or mathematical expressions.
- *graphical* methods use a graphical notation to express entities and/or relationships among entities.
- *algebra or logic based* methods are based on mathematical logics or algebra and express system entities using formula or equations.

Although there are some graphical declarative languages [DKMS+94] and some logic based operational methods, for the sake of brevity, we will consider only two groups: graphical operational languages and declarative logic based method. Furthermore we present only some issues that regard the subjects tackled by this thesis, referring the reader to [HM96] for further information.

2.1.1 Graphical Operational Languages

Examples of graphical operational formal methods are Statecharts[Har87] and Petri nets. Statecharts are based on the state machine model, and exploit the naturalness and simplicity of that model. The explosion of the number of states (typical of finite state machines) is avoided using constructors that allow the design of large systems as combination of smaller state machines. Statecharts are also supported by the tool STATEMATE.

Petri Nets

Petri Net theory was one of the first formalisms to deal with concurrency, nondeterminism and causal connections between events. We assume that the reader already knows the classic classic Petri Net model [Rei85]. Such model has been extended to deal with real-time. Two basic timed versions of Petri Nets have been developed in the past: *time Petri Nets* [MF76] and *timed Petri Nets* [Ram73].

Timed Petri Nets are derived from classical Petri Nets by associating a firing finite duration (a delay) with each transition of the net. The transition is disabled from occurring for the delay period, but is fired immediately after becoming enabled. These nets are used mainly in performance evaluation.

Time Petri Nets (TPNs) are more general than timed Petri Nets. A timed Petri net can be simulated by a TPN, but not vice versa. Time Petri nets differ from traditional Petri nets in that every transition v is associated with a pair of values, usually denoted by $[m_v, M_v]$, belonging to the temporal domain (with $0 \leq m_v \leq M_v \leq \infty$). These are called, respectively, the *lower and upper bound* of v, whereas the pair $[m_v, M_v]$ is called v's *time interval*. Intuitively, the meaning of the pair $[m_v, M_v]$ is that, once v is enabled by the presence of at least one token in each place of its preset, it *can not fire before a time m_v elapsed* (we call this property LB, since it imposes a lowerbound of the firing time of v) and it must fire within M_v unless in the meanwhile it is disabled by the firing of another transition in conflict with it (we refer to this property as UB, since it is related to the upperbound of v). As in traditional Petri nets, tokens are *uniquely* generated and consumed by transition firings. In particular, any firing of a transition consumes one and only one distinct token from each place in its preset (we call this property IU, for input unicity), and introduces one and only one token into each place of its postset; that token can contribute to no more than a single transition firing (we call this property OU, for output unicity).

Formally, a time Petri net is defined as a 5-tuple: $N=<P_N, T_N, F_N, \Theta_N, m_N>$, where

- P_N, T_N, F_N are the set of places, transitions, and arcs of the net; for any transition $v \in T_N$ (resp., place $p \in P_N$)we denote its preset and postset as ${}^\bullet v$ and $v^\bullet$ (resp., ${}^\bullet p$ and $p^\bullet$);

- Θ_N is a function assigning to each transition of the net its *time interval*: Θ_N: $T_N \rightarrow R_\infty^+ \times R_\infty^+$, where $R_\infty^+ = \{x \mid x \in R \wedge x \geq 0\} \cup \{\infty\}$ is the set of non negative reals enriched with the *infinite* value; for each $v \in T_N$, $\Theta_N(v)=<m_v, M_v>$ (also denoted as $[m_v, M_v]$ to conform with the literature on the subject) is a pair of nonnegative real values such that $0 \leq m_v \leq M_v \leq \infty$.

- m_N, the initial marking of the net, is a function of type m_N: $P_N \rightarrow N$ that assigns to each place of the net its initial marking, i.e. for each place $p \in P_N$ specifies the number ≥ 0 of tokens initially present in it.

TPNs will follow us in many other parts of this thesis: in section 2.3.1 we will introduce a logic model for them, in section 3.5.5 we will generalize the temporal relationship between events that TPNs model, in Chapter 4 we will use them to present an approach based on the non-standard analysis and dealing with zero-time transitions, and finally chapter 6 will discuss how to use TPNs to obtain an implementation starting from an abstract specification.

2.1.2 Logics and Algebras

Logics and algebras provide the most abstract approach to the analysis of real-time systems. These approaches typically consist of several elements. One element is a high level, formal specification language in which the requirements that the system must satisfy can be specified. A second element is a proof system (or finite state decision procedures) in which the correctness of the system relative to the specification can be verified.

The current rigorous approaches to real-time systems include Real-Time Temporal Logics.

Real-Time Temporal Logic

The following discussion is taken from [Ost89]. Temporal logic has its origins in philosophy, where it was used to analyze the structure or topology of time. In recent years, it has found application in computer science, especially in the areas of software verification and knowledge-based systems.

In physics and mathematics, time has traditionally been represented as just another variable. First order predicate calculus is used to reason about expressions containing the time variable, and there is thus apparently no need for a special temporal logic.

Philosophers found it useful to introduce special temporal operators, such as $\Box$ ("henceforth") and $\Diamond$ ("eventually"), for the analysis of temporal connectives in language. The new formalism was soon seen as a potentially valuable tool for analyzing the topology of time. For example, various types of semantics can be given to the temporal operators depending on whether time is linear, parallel or branching. Another question that may be asked is whether time is discrete or continuous.

Some of the different types of temporal semantics include:

- Interval semantics [CM96, DKMS+94, RMSM+96, CHR91]. The semantics is based on intervals of time, thought of as representing finite chunks of system behavior.
- Point semantics, in which temporal formulas are interpreted as requiring some system behavior with respect to a certain reference point in time. Past operators refer to the time prior to the reference point. Future operators refer to the time after the reference point.
  Point semantics may be further divided into two classes.
  - Linear semantics [MP83]. In linear semantics, each moment has only one possible future corresponding to the history of the development of the system.
  - Branching semantics [EH86]. In branching time semantics, time has a tree-like nature in which, at each instant, time may split into alternative courses representing different choices made by a system.

2.1.3 Remarks

Operational notations (such as state-transition systems, Petri nets, ...) are very attractive as modeling languages: they include or easily express most of the intuitive, real-world notions like states, events, cause and effect relationships, and transitions from a state to another. However, System Requirement Analysis (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.

As already outlined in the introduction, a special attention must be devoted 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 following section we present TRIO, a real-time logic particularly suitable to easily express temporal notions.

2.2 TRIO

TRIO [GMM90, MMG92] is a first order logic augmented with temporal operators that allow 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*: *Dist* is defined in such a way that if A is a formula and t is a term of the temporal type, then *Dist*(A, d) is a formula meaning that A holds at an instant d time units (t.u.) in the future (if d > 0) or in the past (if d<0) or at the current time (if d=0). Several derived temporal operators may be defined starting from *Dist*, using the propositional connectives, first-order quantification, and conditions on the temporal argument of Dist. A sample thereof is given in Table 2.1, together with short intuitive explanations, whenever needed.

Notice that, for the operators expressing a duration over a time interval (for example *Lasts*), 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 we listed above. 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.

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

Besides temporal dependent (TD) predicates, TRIO introduces temporal dependent variables with domain D, as variables whose value changes in D over time. For instance, TD variables are suitable to model enumerate variables and continuously changing variables, 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.

TRIO has been given a model-theoretical semantics in [GMM90] in a fairly standard way. In [FMM94] we defined a sound and (relatively) complete axiomatic system which is reported in [FGM95], together with some useful meta-theorems. In this axiomatic system the meta-theorems usually found in ordinary predicate calculus can be proved: we mention, among others, the Deduction Theorem, the Generalization theorem, and the Existential instantiation theorem [End72]. We also recall here an important and intuitive meta-theorem, frequently used in TRIO derivations: the Temporal Generalization theorem. It assets that if $\Gamma \vdash \alpha$ and every formula of Γ is of

[1] The definition of *Becomes* in other previous works was $UpToNow(\neg F) \wedge F$

Futr(F, d)	$d \geq \wedge Dist(F, d)$	future
Past(F, d)	$d \geq \wedge Dist(F, -d)$	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
$Until(A_1, A_2)$	$\exists t \left(t > 0 \wedge \begin{matrix} Futr(A_2, t) \wedge \\ Lasts(A_1, t) \end{matrix} \right)$	A_1 holds until A_2 becomes true
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 always held in the past
SomP(A)	$\exists d(d < 0 \wedge Dist(F, d))$	F held sometimes in the past
Som(A)	$\exists d Dist(F, d)$	Sometimes F held or will hold
UpToNow(F)	$\exists d(d > 0 \wedge Lasted(F, d)$	F held for a nonzero time interval that ended at the current instant
Becomes(F)	$UpToNow(\neg F) \wedge (F \vee NowOn(F))$ [1]	F holds at the current instant or starting from the current instant but it did not hold for a non zero interval that preceded the current instant
LastTime(F,t)	$Past(F, t) \wedge Lasted(\neg F, t)$	F occurred for the last time t units ago

Table 2.1: TRIO derived operators

the type $Alw(\Gamma)$, then $\Gamma \vdash Alw(\alpha)$, i.e., if the hypotheses under which a property is proved are not restricted to the present but hold at any time, then the derived property is also always true.

Note that the underlying time is assumed to be linear and the logic easily accommodates both discrete and dense models of time. In the present thesis, we assume as model of time the set of real numbers, which makes the time domain continuous and unlimited both in the past and in the future.

To construct specification of complex systems in a systematic and modular way, TRIO was enriched with concepts and constructs from object-oriented methodologies and with a meaningful graphic notation (in literature we sometime refer to this extension as TRIO+ [MP94]). The basic construct is of course, the construct of *classes*. A class is a set of axioms describing that particular entity and the set of variables and predicates used by that entity (divided in outputs, inputs and internal variables). Classes my by simple or structured, may be generic and may be organized in inheritance hierarchies.

For a complete example using TRIO with classes see [GM96].

TRIO has been adopted in a variety of industrial projects [GLMZ96, BCC+98, CCPMM99, CCPC+99], thus demonstrating its applicability to real-life industrial applications. TRIO is also provided with a tool suite for analysis and verification based on simulation [FM94] and testing [MMM95].

In chapter 3 we will present an original extension of TRIO, that introduces intuitive notions like events, states, periodicity, cause-effect relation between events, and states changing caused by events occurrences.

2.3 Dual Language Approach

In section 2.1 we have compared operational methods and descriptive languages, emphasizing their weaknesses and merits. Typically operational specifications are easier to develop, but too

much oriented towards the implementation, whereas descriptive specifications are more abstract and less biased by the implementation, but require greater skill and ingenuity. As presented in [Ost89], the so called *dual language* approach tries to combine an operation method to specify a system and its behavior (for example using a state machine) and a logic language to express its properties and requirements. In this section we show how a descriptive formal language as TRIO can combined with an operational language such as Time Petri nets.

2.3.1 Axiomatization in TRIO of Time Petri nets

In preceding papers ([FMM91] and, in a simplified version, in [FMM94]) we gave a TRIO axiomatization of timed Petri nets, which we briefly report in the following. For the sake of brevity and simplicity, unless otherwise specified we adopt the following conventions in writing axioms describing Petri net semantics: all axioms are preceded by an implicit *Alw* operator; identifiers denoting transitions (e.g., r, s, u, v) and places (e.g., p, q) are constant names, while identifiers for time distances (e.g., d, e) are variables; free variables in axioms are implicitly universally quantified at the outermost level. Thanks to the *Generalization* and *Temporal Generalization* metatheorems the same is true also for the formulas derived from such axioms.

We divide the axioms describing a TPN into general axioms, describing properties that hold for all nets independent of their topology, and topology dependent axioms.

2.3.2 Basic predicates and general axioms

Since the semantics of timed Petri nets admits multiple simultaneous firings of a single transition, we define a time dependent predicate *nFire*(v, n), with $n \geq 0$, whose meaning is that at the current time transition v fires n times (n=0 iff v does not fire). Of course, the number of transition firings in a given time instant is unique, and there always exists a nonnegative number of firings for each transition: this is expressed, for each transition v, by the two axioms UFN(v) and NNF(v), reported below.

UFN(v) (Unique Firing Number of v):

$$\neg\exists m \exists n (m \neq n \wedge nFire(v, m) \wedge nFire(v, n))$$

NNF(v) (Non-Negative Firing of v):

$$\exists n (n \geq 0 \wedge nFire(v, n))$$

To refer individually to the i-th firing of a transition v we introduce a time dependent derived predicate *fireth*(v, i) meaning that there is an i-th firing of transition v at the current time; since the intended meaning is that v does fire, we require that $i > 0$. The predicates *fireth* is defined in terms of *nFire* as follows.

DEF(*fireth*):

$$fireth(v, i) \stackrel{def}{=} (i > 0) \wedge \exists n (\geq i \wedge nFire(v, n))$$

To model the timing and topological features of a TPN, we introduce the time dependent predicate *tokenF*(s, i, p, v, j, d) meaning that the token produced at the current instant by the i-th firing of transition s enters place p and will be consumed by the j-th firing of transition v after d time units (this implies $p \in s^\bullet \cap^\bullet v$). *tokenF* is the key predicate in the present axiomatization of TPNs, since each one of its occurrences uniquely identifies a single token produced and consumed in the net. *tokenF* is asserted at the time instant of the firing of the first transition argument, s; for reasons of simplicity and symmetry of the formulas, it is useful to introduce a

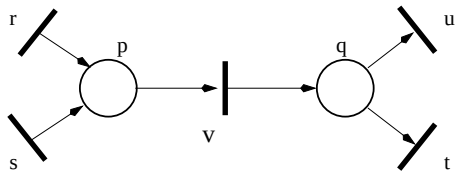


Figure 2.1: A simple net

dual predicate, *tokenP*, having the same meaning but referring to the firing time of the second transition. Predicate *tokenP* is therefore derived from *tokenF* by the following definition. [2]

DEF(tokenP):

$$tokenP(r,i,p,s,j,d) \stackrel{def}{=} Past(tokenF(r,i,p,s,j,d)$$

tokenF implies the firing of the involved transitions, as expressed by the following axiom [3].

FI(r, p, s) (Future Implication):

$$tokenF(r,i,p,s,j,d) \rightarrow fireth(r,i) \wedge Futr(fireth(s,j),d)$$

2.3.3 Topology-dependent axioms

These specify, for each transition v of the net, the above informally described properties of upper and lower bound, and of input and output unicity referred to that transition, in a form that depends on the net topology (i.e., on the places of $^\bullet$v and v$^\bullet$, and on the transitions in their respective presets and postsets). The general form of the topology-dependent axioms is reported in [FGM95]: we illustrate here the axioms LB(v), UB(v), IU(v), and OU(v) specifying such properties for transition v in the simple net fragment of Figure 2.1, to give the reader an intuitive understanding of how they can be expressed as TRIO axioms.

LB(v):

$$fireth(v,i) \rightarrow \exists d(d \geq m_v \wedge \exists j(tokenP(r,j,p,v,i,d) \vee tokenP(s,j,p,v,i,d)$$

UB(v):

$$fireth(r,i) \rightarrow \exists d(d \leq M_v \wedge \exists j(tokenP(r,i,p,v,j,d) \wedge$$
$$fireth(s,i) \rightarrow \exists d(d \leq M_v \wedge \exists j(tokenF(s,i,p,v,i,d)$$

IU(v):

$$tokenP(x,i,p,v,j,d) \wedge tokenP(y,k,p,v,j,e) \rightarrow x = y \wedge i = k \wedge d = e$$

(with x and y variables ranging on the set of transitions).

OU(v):

[2] From the above definition of tokenP the following property,
FP(r, p, s): tokenF(r, i, p, s, j, d) $\leftrightarrow$Futr(tokenP(r, i, p, s, j, d), d)
expressed by a formula symmetrical to DEF(tokenP), can be immediately derived.

[3] From FI(r, p, s) and the definition of tokenP a symmetrical property,
PI(r, p, s) (Past Implication): tokenP(r, i, p, s, j, d) $\rightarrow$ fireth(s, j) $\wedge$ Past(fireth(r, i), d)
can be immediately derived.

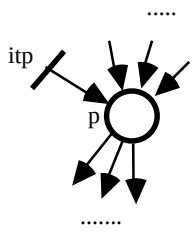


Figure 2.2: The dummy transition that "builds" the initial marking.

$$tokenF(v,i,q,x,j,d) \wedge tokenF(v,i,q,y,k,e) \rightarrow x = y \wedge i = k \wedge d = e$$

(with x and y variables ranging on the set of transitions).

Unlike traditional Petri nets, the instantaneous marking of a TPN does not adequately characterize its overall state: the "age" of each token (i.e., the length of the time interval elapsed from the time of its creation to the present time) is significant too. To uniquely identify tokens, we refer to the events of their production and consumption (i.e., to the firings of transitions), and to the places where they are inserted, or from which they are deleted. The temporal semantics of the nets is ultimately provided by imposing constraints on the distance in time among transition firings. The notion of *instantaneous marking* is formalized in [FMM91, FMM94] as a *derived concept* on the basis of transition firing axioms. We do not report such formalization here because in the present work we focus our attention on the transition firings, which in our approach constitute the observable events of interest.

We however formalize the initial marking of the net, because it is an essential part of the net definition. Let us assume that in the initial state of the net all the tokens have just been created, so that their age is zero: this is the most frequently assumption adopted in the literature and, as it will be apparent from the following, generalizations under this respect are straightforward. As shown in Figure 2.2, for each place p in the net, we introduce a special extra transition called *itp* (initializing transition for p) with $itp^{\bullet}$=p, ${}^{\bullet}itp = \emptyset$. If p's initial marking is k, then the following axiom holds (notice that no *Alw* operator is implicitly assumed in this axiom):

IM(p):

$$nFire(itp,k) \wedge AlwF(nFire(itp,0)) \wedge \forall x AlwP(nFire(x,0))$$

where variable x ranges over the set T_N of the transitions of net N (including itp). Axiom IM(p) states that itp only fires k times at the current instant (and never before or after now) and that no transition ever fired before.

In the remainder of the work, especially in chapter 6, for a given timed Petri net N we call Axt(N) the collection of all topological axioms of N, and Axg the set of general axioms as presented in section 2.3.2. The set Ax(N) = Axg ∪ Axt(N) completely axiomatizes the temporal features of net N. Furthermore, for any timed Petri net N it will be useful to define a TRIO theory $\mathcal{N}$, whose proper axioms, denoted as Ax(N) are precisely the general and topological axioms characterizing the net N. Therefore for any TRIO formula φ constructed from predicates *nFire* and *tokenF*, if $\vdash_{\mathcal{N}} \varphi$ then every execution of net N satisfies the property described by φ .

As an example of a timing property consider, with reference to the net fragment of Figure 2.1, the formula

$$fireth(r,i) \rightarrow WithinF(\exists j(fireth(u,j) \otimes fireth(t,j)), M_v + max(M_u, M_t))$$

can be derived (using UB(v), UB(u), UB(t), FI(v, q, t), FI(v, q, u), and the definition of *WithinF*). It asserts that any firing of transition r will always be followed by a corresponding firing of either t or u within $M_v + max(M_u, M_t)$ t.u..

2.4 Conclusions

In this chapter we have presented the bases of the thesis. We have introduced some formal methods, divided in two main classes: graphical operational methods and declarative logic-based languages. In particular section 2.1.1 has presented time Petri Nets and section 2.2 has been dedicated to the presentation of the temporal logic TRIO. In section 2.3.1 we have seen the dual language approach, combining time Petri Nets and TRIO.

Chapter 3

High level notions for real-time systems

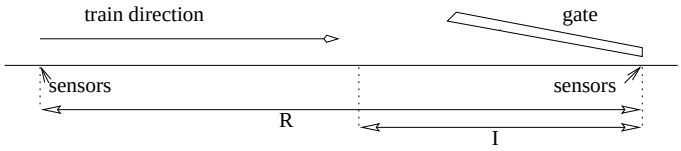
In the introduction we have argued advantages and objectives of applying formal methods to requirement analysis of real time systems. In the previous chapter we have presented several formal methods and the state of the art about them (in particular TRIO). In the present chapter we propose a new framework for system requirements analysis, based on TRIO but 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. Thanks to the ability of these constructs 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.

In section 3.1 we introduce the notion of event (as a predicate that holds at isolated time points), in 3.2 the notion of interval predicate (a predicate that holds, or does not hold, for non-empty temporal intervals), and then in section 3.3 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. We generalize these definitions to formulas and to time dependent variables (in section 3.4) 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. Note that the use of such high level entities, although it is not mandatory, really simplifies the modeling and the reasoning about real-time systems, giving a precise and formal semantic to vague terms like *event* and many others

In the section 3.5 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 the first sections of this chapter, 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 3.5.2 we introduce a direct way to define new derived entities from other previously defined ones. In section 3.5.3 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 3.5.4 we introduce periodic events. In section 3.5.5 we discuss means to model cause- effect relations between events. Last section presents some related works.

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 [HJL93] 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 [HM96]. 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 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 bar is operated by commands *goUp* and *goDown*; its 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, when it is moving up, by means of a *down* command; however its motion cannot be interrupted when it is downward. It takes the bar γ time units to reach the closed (respectively, open) position starting from an open (respectively, closed) state. If the bar motion is inverted after it has been moving up for exactly t time units, then it takes again t time units to reach the closed position.

3.1 Point-based predicates: Events

We call *point-based* or *events* the predicates that hold in isolated time points, and are false elsewhere. Their behavior is visualized in Figure 3.1. Their formal definition in terms of TRIO is as follows.

Definition 3.1 point-based predicate: *a TRIO predicate E is called point-based or event if and only if*:

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

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 3.3. Some formal languages solve the problem by banning events having null duration. See [GMM99], briefly reported in section 4.3, 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 [BGJ91] might be modeled as events.

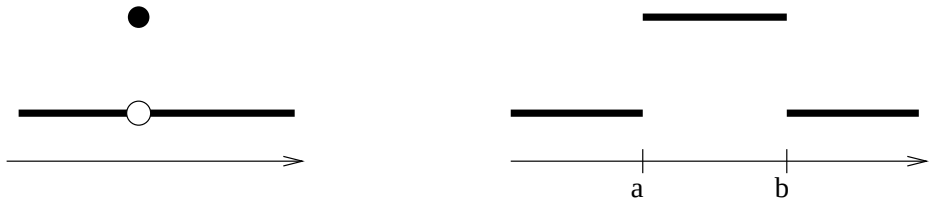


Figure 3.1: event and interval predicate

Example 3.1 In our case study, GRC, we have used events to represent commands given to the bar: up, down. Informally, when the bar receives an up command and it is closed, it starts opening. If it is opening (but not open yet) and it receive a down command, it starts closing. We think of these commands as control pulses having a very small duration, which we consider null. ■

3.2 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 3.1.

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

Definition 3.2 interval-based predicate: *a TRIO predicate I is interval-based if and only if:*

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

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 3.1). 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 3.2): 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 3.3 ***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 [GM99]. 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 3.3.3 and in section 3.4.3, when other kinds of considerations will allow us to make a choice.

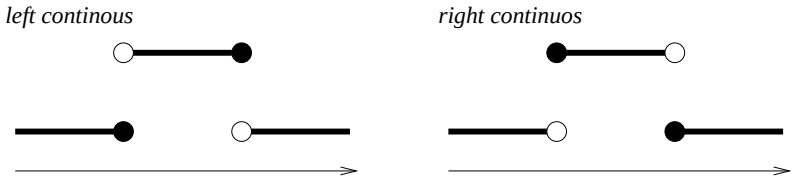


Figure 3.2: left and right continuous interval predicates

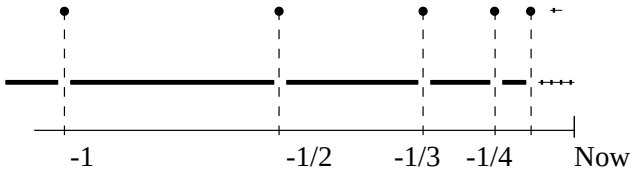
3.3 Non-Zeno requirement for predicates

In this section we define the *non-Zeno* or *finite variability* requirement [AL94] for a TRIO 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 and allowing Zeno predicates would be a source of incompleteness, as shown by the following example.

Example 3.2 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. ■

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 3.4 non-Zeno requirement for TRIO predicate: *a TRIO predicate A is non-Zeno iff:*

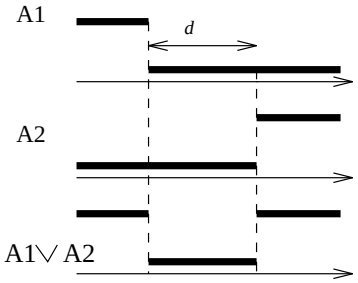
$$(UpToNow(\neg A) \vee UpToNow(A)) \wedge (NowOn(\neg A) \vee NowOn(A))$$

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".

Note Previous definitions for non-Zeno TRIO predicate, were based on counters [MMP+96]. The definition proposed here is equivalent but simpler.

The predicate E in the previous example (example 3.3) does not satisfy the non-Zeno requirement. In the current instant neither $UpToNow(\neg E)$ nor $UpToNow(E)$ hold, because there is no ε such that $Lasted(E, \varepsilon)$ or $Lasted(\neg E, \varepsilon)$.

Note 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 3.6 reports on related works adopting solutions similar to ours.



3.3.1 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 to obtain definitions of the entities of practical interest, as in the following propositions, whose proofs are in [GM99].

Proposition 3.5 non-Zeno event: *a TRIO predicate E is a non-Zeno event if and only if:*

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

Proposition 3.6 non-Zeno interval predicate: *a TRIO predicate A is a non-Zeno interval predicate if and only if:*

$$\begin{array}{rcl} (UpToNow(A) \wedge NowOn(A) \wedge A) & \vee & (UpToNow(\neg A) \wedge NowOn(\neg A) \wedge \neg A) \\ & \vee & \\ (UpToNow(\neg A) \wedge NowOn(A)) & \vee & (UpToNow(A) \wedge NowOn(\neg A)) \end{array}$$

Proposition 3.7 non-Zeno interval left continuous predicate: *A is a non-Zeno left continuous interval predicate if and only if :*

$$\begin{array}{rcl} (UpToNow(A) \wedge A & \vee & UpToNow(\neg A) \wedge \neg A) \\ & \wedge & \\ (NowOn(A) & \vee & NowOn(\neg A)) \end{array}$$

To summarize the definitions presented so far, we can picture the set of TRIO predicates as in Figure 3.3.

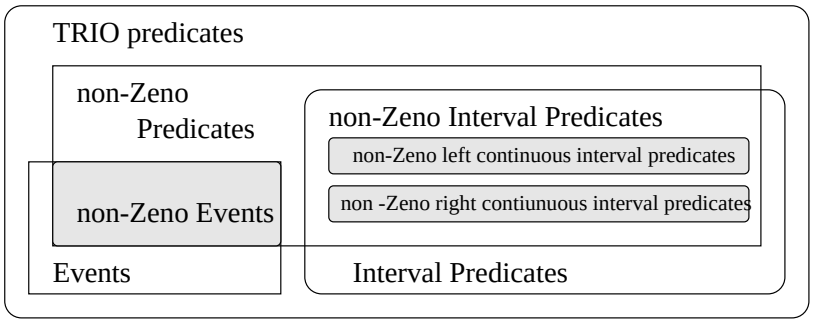


Figure 3.3: a subset system for TRIO Formula

$\wedge$	f	i	lci	rci	e
e	e	e	e	e	e
rci	f	f	f	rci	
lci	f	f	lci		
i	f	f			
f	f				

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

$\vee$	f	i	lci	rci	e
e	f	f	i	i	e
rci	f	f	i	rci	
lci	f	f	lci		
i	f	f			
f	f				

Table 3.1: closure properties of boolean operators

3.3.2 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:

Definition 3.8 non-Zeno event formula : *a TRIO 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.

3.3.3 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 results of the application of the propositional operators $\wedge$, $\neg$, and $\vee$ to operands of the various types is summarized by Table 3.1, where f stands for a generic formula, *rci* (*lci*) for right (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 *Remarks on particular applications of the rules.* If E is an event, and F a generic formula, $E \wedge F$ is still an event, hence conditioned events (as defined in SCR [HJL96]) 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 [CH97]). Fortunately the class of left- and 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 on this subject again in section 3.4.3.

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

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

Table 3.2: 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 3.2. 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 TRIO predicates, we can immediately be sure that the formula $Becomes(F) \wedge G$ is an event.

3.4 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 5.2.2).

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 3.9 point variable: *a variable v is a point variable with default value* d *if :*

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

Point variables might model data flows of LUSTRE [HLR92] and SIGNAL [BGJ91]. 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.

Example 3.3 In the GRC case study we consider the event *"a train enters region I"*. As we have more than one track, more than one train can enter the region I at the same time. We can use the following point integer variables with default value 0: *RI* models the number of trains entering region R, *II* those entering region I, *IO* those exiting region I (and therefore region R as well). ■

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 3.10 interval variable: *x is a interval variable with domain D if and only if, for every* $a \in D$*:*

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

Event counters

A particular kind of interval variable is that of *event counters*, i.e., integer variables having as value the number of occurrences of a given event. Given an event E we denote its counter as $\#E$. The definition of event counters will be given in section 3.5.3.

Continuity

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

Definition 3.11 left (right) continuity: *a variable x with domain D is a left (right) continuous interval variable if and only if, for every* $a \in D$:

$$\begin{array}{l} x = a \ \rightarrow \ UpToNow(x = a) \\ (x = a \ \rightarrow \ NowOn(x = a)) \end{array}$$

3.4.1 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 3.12 non-Zeno variable in a countable domain: *x in a countable domain is non-Zeno iff:*

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

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 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

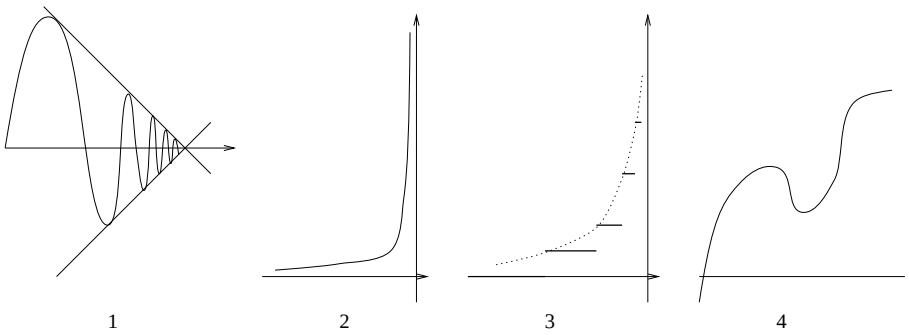


Figure 3.4: examples of Zeno and non-Zeno behaviors

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 x in an uncountable domain D to be piecewise analytic when considered as a function of time[1].

More formally, we define a variable x 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 x 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 3.13 non-Zeno variable: *a variable* x *in an uncountable domain is non-Zeno iff:*

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

Notice that if the variable domain is countable this definition reduces to definition 3.12. Indeed, the only analytic functions with value in a countable domain are the constant functions. Then the variable must be piecewise constant (see [GM99]).

Definition 3.13 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.

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 3.4), $(int)\frac{1}{t}$ (function 3), $sin\frac{1}{t} * t$ (function 1, continuous but Zeno).

Next we provide some more rationale for definition 3.13. We argued that the definition 3.12 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 3.4. Consider for example a time dependent variable x equal to the function $sin\frac{1}{t} * t$ in a left interval of the origin. This function is pictured in the figure 3.4(1) and it is continuous. Nonetheless the formula $x = 0$ 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 $sin\frac{1}{t}e^{-1/x^2}$: it is very regular, even C_∞, (but not analytic) and indeed it does not satisfy

[1] In extreme summary, an analytic function is very regular: it can be expanded in a power series, for example a Taylor series. See [CJ74] for a deeper insight.

our intuitive requirements for non-Zenoness, because it changes sign an infinite number of times in every interval surrounding the origin.

The following theorems provide further evidence that only non-Zeno formulas are obtained starting from non-Zeno variables.

Theorem 3.14 *if the time dependent variable x is non-Zeno then the formula x=0 is non-Zeno*

Proof. See the identity principle for olomorph function [CJ74]: the zeros of an analytic function are isolated (unless, of course, f vanishes identically) □

Theorem 3.15 *if the time dependent variable x is non-Zeno, then ∀ a the formula x = 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. □

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.

3.4.2 Non-Zeno point and interval variable

The following two propositions, whose proof is reported in [GM99], combine the definition of non-Zeno and of point and interval variable.

Proposition 3.16 non-Zeno point variable: *x is a non-Zeno point variable iff exists d such that:*

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

We call d the default value of x.

Proposition 3.17 non-Zeno interval variable: *x is a non-Zeno interval variable iff it is piecewise constant:*

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

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 3.18 non-Zeno left (right) continuous interval variable: *x is a non-Zeno and left (right) continuous interval variable if and only if:*

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

3.4.3 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 (+, -, /, and *) whereas this is not the case for generic interval variables. [2]

These properties, together with those seen in section 3.3.3, 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 3.6 for a brief comparison with other languages.

3.4.4 Conclusive remarks

Table 3.3 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 modeling 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 way the engineer will be able 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.

3.5 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, 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 3.5.2 we introduce a direct way to define new derived entities from other previously defined ones. In section 3.5.3 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 3.5.4 we introduce periodic events. In the last subsection we discuss means to model cause- effect relations between events.

3.5.1 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

[2] 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.

	Predicate	Variable
p	$E \rightarrow \begin{pmatrix} UpToNow(\neg E) \\ NowOn(\neg E) \end{pmatrix} \wedge$	$v \neq d \rightarrow \begin{pmatrix} UpToNow(v = d) \\ NowOn(v = d) \end{pmatrix} \wedge$
i	$I \rightarrow \quad (UpToNow(I) \vee NowOn(I))$ $\wedge$ $\neg I \rightarrow (UpToNow(\neg I) \vee NowOn(\neg I))$	$\forall a\, x = a \rightarrow \begin{pmatrix} UpToNow(x = a) \\ NowOn(x = a) \end{pmatrix} \vee$
lci	$I \rightarrow UpToNow(I)$ $\wedge$ $\neg I \rightarrow UpToNow(\neg I)$	$\forall a\, x = a \rightarrow UpToNow(x = a)$
nZ	$UpToNow(\neg A) \vee UpToNow(A)$ $\wedge$ $NowOn(\neg A) \vee NowOn(A)$	$\exists f \exists g \begin{pmatrix} f \in AF_\circ \wedge g \in AF_\circ \wedge \\ \exists d \forall t \left(0 < t < d \rightarrow \begin{pmatrix} futr(x,t) = f(t) \\ past(x,t) = g(t) \end{pmatrix} \wedge \right) \end{pmatrix}$
nZp	$UpToNow(\neg E) \wedge NowOn(\neg E)$	$UpToNow(v = d) \wedge NowOn(v = d)$
nZi	$UpToNow(I) \quad \wedge \quad NowOn(I) \quad \wedge I \quad \vee$ $UpToNow(\neg I) \quad \wedge \quad NowOn(\neg I) \wedge \neg I \quad \vee$ $UpToNow(\neg I) \quad \wedge \quad NowOn(I) \quad \vee$ $UpToNow(I) \quad \wedge \quad NowOn(\neg I)$	$\exists a \exists b \begin{pmatrix} UpToNow(x = a) \quad \wedge \\ NowOn(x = b) \quad \wedge \\ (x = a \vee x = b) \end{pmatrix}$
$nZlci$	$\wedge \begin{pmatrix} UpToNow(A) \quad \wedge \quad A \quad \vee \\ UpToNow(\neg A) \quad \wedge \quad \neg A \\ (NowOn(A) \vee NowOn(\neg A)) \end{pmatrix}$	$\exists a \exists b \begin{pmatrix} UpToNow(x = a) \wedge x = a \\ NowOn(x = b) \end{pmatrix}$

Table 3.3: all the definitions

of sensors and actuators. This view is consistent with several models proposed in the literature, see for instance Parnas' Four Variables model [PM95] and Jackson & Zave's model [ZJ97].

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.

3.5.2 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 a defining 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 3.3.3 and Tables 3.1 and 3.2 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 3.3) or directly the rules given in section 3.4.3. In the tool that we have built the constraint about *expression* is automatically generated (as TCC: Type Correctness Conditions) and the user is in charge of proving it, possibly with the assistance of the tool itself.

Example 3.4 In our case study we have defined these two events:

$$\begin{array}{ll} stopMovingUp & \leftrightarrow Lasted_{ee}(gate = movingUp, \gamma) \\ stopMovingDown & \leftrightarrow \left(\begin{array}{c} Lasted_{ee}(gate = movingDown, \gamma) \\ \vee \\ \exists t \left(\begin{array}{c} Lasted_{ee}(gate = movingDown, t) \wedge \\ Past(Lasted_{ee}(gate = movingUp, t), t) \wedge \\ Past(Becomes(gate = movingUp), 2 * t) \end{array} \right) \end{array} \right) \end{array}$$

stopMovingUp occurs when the gate reaches the open position after moving up for γ time units from a closed position (recall that the downward motion of a bar cannot be interrupted by an *up* command, while an upward motion can be interrupted by a *down* command). The other event, *stopMovingDown*, occurs after the gate has been moving down for γ time units (in case the bar was completely open when it received the *down* command), or because t time units ago an upward motion lasting since t time units before was interrupted by a *down* command that caused the bar to reverse its motion to *movingDown*.

We have defined also the following 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 defined directly also some outputs of the DUC. *down* is a controlled (output) 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.

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

■

3.5.3 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 [HJL96].

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

Definition 3.19 Change_relation: *for a interval variable with domain D, given a set of events E*[3], we define a relation over $D \times E \times D$ and we call it **change_relation**

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 $change\_relation(x, e, y)$, 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 $change\_relation(x, e_1, y)$ and $change\_relation(x, e_2, z)$, then, when e_1 and e_2 occur simultaneously, the variable may nondeterministically turn from x into y or z; if $e_1 = e_2$ (for brevity = e), and if 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 s, by the following two TRIO axioms. Clause 1 above (describing the case when the variable keeps its value) is formalized by Axiom 3.20:

Axiom 3.20 continue

$$UpToNow(s = old) \wedge \neg\exists e, new(change\_relazion(old, e, new) \wedge e)) \rightarrow NowOn(s = old)$$

[3]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 clause 2 (describing the cases when the variable may change its value) is formalized by Axiom 3.21:

Axiom 3.21 change

$$UpToNow(s = old) \wedge \exists e_1, new_1\ (change\_relation(old, e_1, new_1) \wedge e_1) \\ \rightarrow \exists e_2, new_2\ (change\_relation(old, e_2, new_2) \wedge e_2 \wedge NowOn(s = new_2))$$

In Axiom 3.21 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.

Using tables

The use of tables in specifications is known to provide great benefits, and tables are particularly suitable to specify relations [Par92]. Some notations intensively use tables, since tables offer a concise and clear way to specify software and system requirements. SCR [HJL96] 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). A very intuitive way to specify the above described relation is by means of tables like the following one:

UpToNow	event	NowOn
v_1	e_{11}	v_{11}
	e_{12}	v_{12}
v_2	e_{21}	v_{21}
		v_{22}
	...	...

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 3.22 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 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):

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

Definition 3.22 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 3.23 Change for a deterministic change_relation: *for a deterministic relation axiom 3.21 (change) is equivalent to the formula:*

$$UpToNow(s = old) \wedge change\_relation(old, e, new) \wedge e \rightarrow NowOn(s = new)$$

while the first axiom (continue) remains unchanged. Proposition 3.23 has been proved with the TRIO prover based on PVS.

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

Example 3.5 In our case study, the behavior of the bar can be described 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	*stopMovingDown*	*closed*
open	*down*	*movingDown*

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

■

Note *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 *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

[4]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 [GM99].

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.

3.5.4 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 3.24 ***Periodic Event:*** An event A is called periodic with the period d iff:

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

Propositions

1. A is actually an event
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. $A \rightarrow Lasts(\neg A, d)$, $A \rightarrow Futr(a, d)$, and $A \rightarrow NextTime(A, d)$: A is periodic in the future
5. $A \rightarrow \forall k Dist(A, k \cdot d)$: A repeats itself every d time units

Definition 3.25 ***Semi-periodic Event*** A is a periodic event in the future, with period d if and only if

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

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

Notice that for semi-periodic events $Som(A)$ cannot be proven, i.e., it is not guaranteed that the event occurs sometime.

3.5.5 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.[5]

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

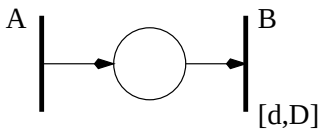
Notice that the above cases of cause-effect relation are *deterministic* as, once the cause (event A) occurs, then the occurrence time of the effect (event B) is precisely determined.

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 3.26 *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 [AH96, MMT91] and Timed Petri Net (TPN) in [MF76] (presented in section 2.1.1). Remember that in TPNs 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 [GM99] by showing that they both model the relation introduced in Definition 3.26.

Notice that simple, apparently obvious formalizations are easily proved incorrect. For instance, in [GM99] we show that the following pair of axioms

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

do not capture the one-to-one binding between occurrences of A and B[6].

Using special predicates to formalize the relationship

The approach of using special predicates to model the relationship for TPNs was given in section 2.3.1. The same approach can be generalized 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 and occurrence of B t time units later. The following axioms characterize predicate *ACausesB*.

[5] 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: $(A \rightarrow Futr(B, d)) \wedge (B \rightarrow Past(A, d))$.

[6] 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 [Koy89].

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 [GM99]).

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 [GM99] 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 3.27, reported in [GM99], showing that it is immaterial from a mathematical viewpoint.

Theorem 3.27 *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 3.26 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 3.27 expresses the concept stated in Definition 3.26 with very simple relations between event counters. Thanks to their simplicity (they are just linear inequalities) these relations can be used very easily and effectively 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 3.26, introduced for simple events, can be extended to events with multiple simultaneous occurrences. Theorem 3.27 maintains its validity also in this generalized framework.

A further, interesting generalization of the one-to-one relation introduced in Definition 3.26 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 3.27 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	$Dist(AoneTooneB(t_1), t_1) \wedge$ $Dist(AoneTooneB(t_2), t_2) \quad \rightarrow t_1 = t_2$

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 3.6 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 a train entering region *R*, entering region *I*, and exiting region *I*, and that they are multiple events).

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

■

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 3.26, 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.

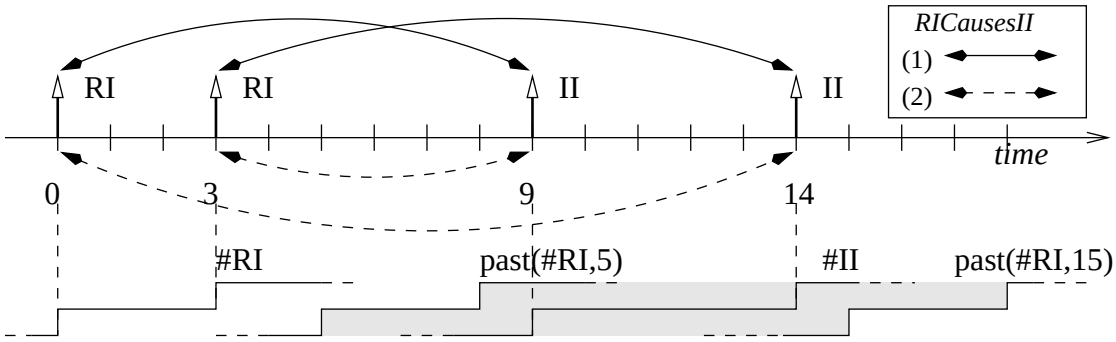


Figure 3.5: Models for axiomatizations based on special predicates or on counters.

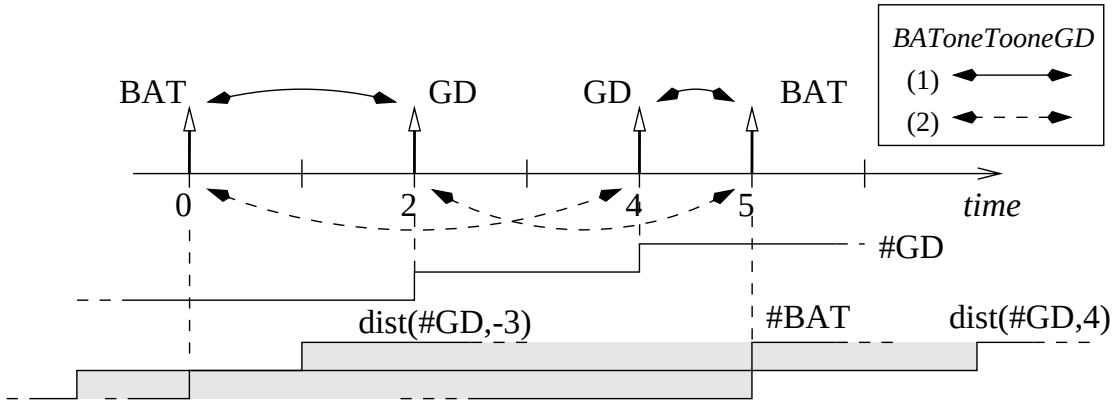


Figure 3.6: BAT and GD

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 3.5: 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 3.5, 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 3.6, where two bank account transactions take place at time 0 and 5, and goods delivery occur at time 2 and 4. Figure 3.6 shows that there is only one model based on counters #*GD* and #*BAT*, and there are two models based on the predicate *BATone-TooneGD*.

(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.)

3.6 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 chapter, mainly: non-Zenoness and finite variability, continuity of interval variables, and abstract relations among events.

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 [YMW97]). However the same assumption is taken by some other formalisms using continuous time, like timed CSP [RR88] and ASTRAL [CPGK97]. 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. Such 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 behavior as sequence of events or states [CH85, MMT91], finite variability is defined requiring that time "eventually increases above any bound" [Sha93]. In 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.

The same assumption, in a different form, can be found in interval temporal logics, see for instance the Duration Calculus [CH97] and, for a formal definition (but without using the logic itself), RTGIL [MRK+97]. Note that interval temporal logics deal only with piecewise constant variables. For this kind of variables we have given formal definitions using TRIO itself. This has been necessary to the encoding of TRIO in PVS, where an intuitive abstract definition must be substituted by a definition in terms of the logic itself and its encoding (i.e. TRIO in our case) or at least using the language of PVS (as in [Sha93])[7]. Encoding the non- Zeno requirement in TRIO and then in PVS, allowed us to prove several interesting theorems, and reuse them in proofs of system properties. Another original contribution of this work on this subject is a formal definition of non Zeno requirement for variables in uncountable domains.

Continuity of interval entities

In [CH85] 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) [CH97] 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 3.3.3. RTGIL [MRK+97] simply assumes variables to be right continuous, supporting that decision with arguments similar to ours.

[7] PVS, however, does not have a predefined theory about sequences or limits. Therefore many definitions of non-Zenoness using those contructs, could not be tranlsated into PVS.

Abstract relations among events

The idea of using counters to model relations between events was already presented in [CH85]. 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 [YMW97] 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 [MF76] and in MMT timed Automata [MMT91] (whose embedding in PVS is done in [AH96]). We have shown how the use of counters can be extended to specify this relation in a very simple and effective way.

3.7 Conclusions

In this chapter we have 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, etc.

This framework is particularly well suited to supporting the activity of System Requirements analysis, a preliminary activity of crucial importance in the preliminary phases of the development of highly critical systems, that requires modeling the environment together with the Device Under Construction, stating 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 [GLMZ96, BCC+95]. 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 [AGM97].

Chapter 4

Dealing with Zero-Time Transitions: a "non standard" approach

In the modeling of time-dependent systems it is often useful to use the abstraction of *zero-time transitions*, i.e., changes of system state that occur in a time that can be neglected with respect to the whole dynamics of system evolution. Such an abstraction, however, sometimes generates critical situations in the formal system analysis. This may lead to limitations or unnatural use of such formal analysis. In this section we present an approach that keeps the intuitive appealing of the zero-time transition abstraction yet maintaining simplicity and generality in its use. The approach is based on considering zero-time transitions as occurring in an infinitesimal, yet non-null time. The adopted notation is borrowed from non-standard analysis. The approach is illustrated through Petri nets as a case of state machines and TRIO as a case of logic-based assertion language, but it can be easily applied to any formal system dealing with states, time, and transitions.

4.1 Introduction

Several formalisms have been proposed recently for the modeling and analysis of time-critical systems. In many cases systems to be analyzed are described by some abstract machine and their properties are formalized through suitable formulas. Abstract machines are characterized by some notion of *state* and by *transitions* from one state to another. In such formalizations it is often useful to adopt the abstraction of *zero-time transitions*, i.e., transitions whose duration is so short that it can be neglected w.r.t. the whole system evolution.

Allowing transitions to occur in zero-time is certainly intuitively appealing; it exposes however to some risks in mathematical formalization. The main problem arises from the fact that it is quite natural to describe system state evolution by formalizing its state as a total function of the time variable: $s(t)$ denotes system's state at time t. By this way, the effect of a transition tr is described as a state transformation that leads system's state from s_1 at time t_1 (at the beginning of the transition) to s_2 at time t_2 (at the end of the transition). If we allow tr to have a null duration, however, we obtain that $t_1 = t_2$ and, therefore, at t_1 the system is *both* in state s_1 and in state s_2: such a claim contradicts the intuitive assertion that at a given instant the system is in exactly one state. It also exposes to the risk of formal contradictions if, e.g., one describes system state as the property that some node is marked or not.

To overcome this difficulty several approaches have been followed in the literature:

- In ASTRAL [CPGK97] zero-time transitions are simply excluded.
- At the other extreme, in Esterel [BC84] all transitions are assumed to take zero time. This is due to the typically synchronous approach on which Esterel is based: the abstraction

provided by the model assumes that a whole time unit elapses and that at its end a *finite* sequence of state transitions occurs. As with all synchronous abstract machines time is intrinsically a discrete set [1].

- In [HL96], instead, time *must* be a dense set. System evolution is described as an alternating sequence of trajectories and actions. A trajectory corresponds to a time interval where the state is constant or changes continuously with time; actions are instantaneous transitions that change the system state. Thus, the system state is a piecewise constant or continuous function of time.

- In other cases [Ost89], [BD91], [Cer93] time is modeled as a particular system variable and its value is explicitly updated by special transitions (e.g. tick in [Ost89], which imposes a discrete time domain) which are interleaved with other state transformations. This approach sometimes imposes rather unnatural formalizations of system properties, and makes their proof much longer and unintuitive. For instance, it could happen that in system description two states s_1 and s_2 have the same value for the "time variable", which prevents the classical, simple modeling of system state as a function of time, and hinders the use of familiar locutions such as "the system state at time t". Also, in the case of discrete time domains, the unexperienced user must be emphatically warned that "the next value of system variable v" is not necessarily "the value at time t+1".

- In [FMM94], reported in section 2.3.1, we have provided an axiomatization of timed Petri nets which allows zero-time transitions. In the general case, however, such an axiom system must deal with the possibility of several firings of the same transition in the same instant: this imposes a fairly cumbersome notation and requires a convention to define a single state (marking) of the net at a time t when several transitions fire simultaneously. In [SS96] the authors show that problems arise even when modeling time in Petri nets by means of token time stamps: in presence of instantaneous events, they propose to add to time stamps an index denoting the order of production of simultaneously generated tokens. This solution is similar to that proposed in [FMM94] and shares the same weaknesses in terms of naturalness and generality.

To summarize, all approaches proposed so far had to pay a price either in terms of generality, or in terms of naturalness in the expression and proof of system properties, or in terms of heaviness of the mathematical notation.

In this section we present a novel approach which conjugates intuition with mathematical rigor and generality. Going back to the original intuitive meaning of zero-time transitions we consider such transitions as occurring in an infinitesimal - yet non-null - time; in the traditional continuous mathematics terminology "a zero-time transition actually takes a non-null time whose measure is smaller than any finite positive number".

We fully formalize this approach within the framework of *non-standard analysis* [Rob61, Rob96], which provides a simple and intuitive notation to formalize infinitesimal calculus. We instantiate our approach with reference to timed Petri nets and to the logic language TRIO which we are using for our research in the field of real-time systems [FMM94]. We will show, however, that our approach is absolutely general and can be applied as well to any other abstract machine and assertion language. Furthermore, despite the fact that we deal with infinitesimal numbers, our approach can be applied both to dense and to discrete time domains.

The work is organized as follows. Section 4.2 provides a short summary of non-standard analysis; Section 4.3 provides an axiom system for timed Petri nets based on a minimal subset of the TRIO language and assuming a time domain augmented with infinitesimal numbers. Section

[1] A typical application of this model is the synthesis of hardware circuits. Not surprisingly their design is based on a synchronous model where combinatorial gates (and, or, not, ...) are modeled as zero-time transitions: obviously, it is the designer responsibility to verify that, in practice, all switchings corresponding to combinatorial evaluation occur within a single machine cycle so that the zero-time abstraction is correct

4.4 provides a few examples of property proofs in the new axiomatization and shows that these new proofs are considerably simpler than those derived with previous approaches.

For the sake of shortness we limit ourselves to the essential aspects of the proposed approach; the skipped details, however, can be easily filled out.

4.2 A summary of non-standard analysis

In this section we introduce the main concepts of the modern theory of infinitesimals founded by A. Robinson [Rob61, Rob96], the non-standard analysis (NSA in brief). We provide only the minimum background that is needed to explain our application of this theory.

The main idea that facilitates practical application of NSA is due to E. Nelson [Nel77]; he defined a theory, called Internal Set Theory (IST), which includes a typical axiomatization of arithmetics (say, ZFC, the Zermelo-Fraenkel set theory with the axiom of Choice [Coh66]) and extends it through the predicate *standard* (briefly *st*), which is left deliberately undefined, plus three additional axiom schemes. Thanks to the new *st* predicate introduced by IST, we can say whether a number (of the usual numeric sets such as the reals R and the naturals N) is either standard or not. Every concrete number one could write or a computer could generate is standard. Thus numbers such as $1, \pi, 1/100$, are standard.

The predicate standard is used to introduce the concept of infinitesimal in R in the following way: x is infinitesimal if $x \geq 0$ and x is smaller than any positive standard number (smaller then any number we can write or calculate). 0 is infinitesimal; in fact, it is the only infinitesimal standard number. Close to 0 there are the non standard infinitesimal numbers (infinitesimal and greater than zero). They are not the only non standard numbers. R includes many other non standard numbers, that are the result of adding and subtracting infinitesimal amounts to standard numbers. There are also unlimited non standard numbers, i.e., the inverses of infinitesimal non standard numbers, greater than every standard number.

Now we formalize the above concepts in R through first order predicate formulas, where $\forall x^{st} A(x)$ is an abbreviation for $\forall x(st(x) \rightarrow A(x))$:

infinitesimal(ε) is defined as $\forall^{st}$ x (x>0 $\rightarrow |\varepsilon| \leq$ x)

nsinfinitesimal(ε) is defined as $\forall^{st}$ x (x>0 $\rightarrow |\varepsilon| \leq$ x) $\wedge\neg$st(ε)

infinitesimal+(ε) is defined as $\forall^{st}$ x (x>0 $\rightarrow$ 0< $\varepsilon \leq$x) $\wedge\neg$st(ε)

Formulas in which the predicate *st* does not occur are called *internal* formulas. whereas formulas using the standard predicate are *external*. The definitions given above are all external formulas, while formulas of classical arithmetic are internal. Given an internal *sentence* (a formula with no free variables) A, the *relativization* of A to the standard sets, denoted as A^{st}, is obtained from A by restricting all quantifications to standard values (i.e., by substituting every occurrence of $\forall$x by $\forall^{st}x$). A fundamental meta-theorem of IST (hereinafter called the Transfer Theorem) asserts that $A^{st} \leftrightarrow A$; hence all theorems of conventional mathematics also hold in IST when relativized to the standard sets, and, conversely, to prove an internal theorem it suffices to prove its relativization to the standard sets. Another fundamental result of IST ensures that it is a *conservative extension* of ZFC, that is, every internal sentence that can be proved in IST can also be proved in ZFC.

The results of the usual operations (*, +, -, and /) between standard and non standard numbers are driven by the so called Leibniz rules [DD95]. The following tables express some of these rules using the symbol $\emptyset$ for an infinitesimal, £ for a limited number (i.e., a number that is not larger than any standard number).

+	$\emptyset$	£
£	£	£
$\emptyset$	$\emptyset$	

×	$\emptyset$	£
£	$\emptyset$	£
$\emptyset$	$\emptyset$	

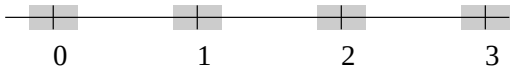


Figure 4.1: An intuitive display of integer numbers augmented with non-standard neighbors.

From these tables we can derive the intuitive rules: "The sum of two infinitesimal numbers is a infinitesimal number, the product of a limited number by an infinitesimal number is infinitesimal, etc ...."

Here we do not express in our axiom system these rules (as well as operations between standard numbers) and assume other useful properties from the IST theory (e.g. there exists in R and in N an infinitesimal number, ...)

4.3 A non-standard axiom system for time Petri nets

In this section we provide an axiom system for time Petri nets (TPN) (see section 2.1.1) modifying the axiom system seen in section 2.3.1.

It will appear, however, that the method illustrated here can be applied as well to any formalism that is based on the notions of state and transition (Finite or infinite state machines) and to several logic-based assertion languages that allow dealing with time issues (e.g., [CPGK97], [Koy89], [Ost89]).

Let us first define a suitable time domain T enriched with non-standard numbers and let us denote the augmented domain as $\widetilde{T}$. For simplicity let us assume that the original time domain is a subset of the set of real numbers R. For instance, we could take as time domain T the set of integers: thus $\widetilde{T}$ would be the set of integers augmented with the non-standard numbers that are infinitely close to an integer number. Figure 4.1 suggests an intuitive graphical representation of such a set. In general, $\widetilde{T}$ can be visualized by surrounding each standard real element of T by a "cloud" of nonstandard reals that differ from it by an infinitesimal number.

Next we introduce the following basic predicates describing TPN behavior. This formalization is a simplified version of that already presented in section 2.3.1. We assume that nets are 1-bounded (with no more than one token in a place). This restriction –together with other minor assumptions– guaranteed *a priori* that no transition could fire twice in the same instant. Dealing with the general case would require the predicates seen in section 2.3.1.

- *fire*(v) means that the transition v fires now, i.e., at the current instant.
- *tokenF* (v, w, d) means that transition v fires now and the token produced by its firing will be consumed by transition w, d time units in the future. Symmetrically, the tokenP predicate is defined by
  tokenP(v, w, d) $\leftrightarrow$ *Past (tokenF (v, w, d), d).*

Also, we keep here a minor simplification that excludes that the same pair of transitions has more than one place in the intersection between pre- and post-sets. This assumption does not cause any loss of generality and only allows some simplification in the notation.

The essential features of our approach are the following:

1. There are no firings occurring *exactly* in null time: in general, if a lowerbound, upperbound pair $<m_v, M_v>$ is associated with a transition v, we will assume that v's firing may occur at a time distance t since its enabling with $m_v+\varepsilon_1 < t < M_v+\varepsilon_2$, ε_1, ε_2 being two *positive* infinitesimal numbers.

2. No transition can fire more than once exactly at the same instant; it can, however, fire at two instants whose distance is infinitesimal.

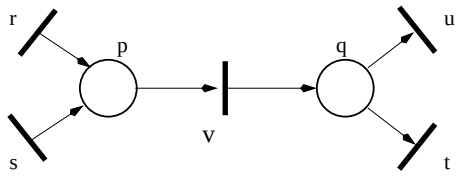


Figure 4.2: Simple time Petri Net

3. There is *exactly one* system state associated to every instant (having a standard numerical value or not) of the time domain.

We are now ready to give axioms formalizing the behavior of TPNs. Following the same schema as in section 2.3.1 we consider transitions of the types given in Figure 4.2 (already seen in Figure 2.1 at page 17).

We augment both the lower and the upper bound of every transition by an infinitesimal positive constant amount. This choice allows us to treat in the same manner zero-time transitions, transitions with lowerbound equal to the upperbound, and any other transition with upperbound > lowerbound. Thus the only requirement about m_v and M_v is $0 \leq m_v \leq M_v$.

For the fragment of Figure 2.1 the axiom related to v's lowerbound is

LB(v) =

$$fire(v) \rightarrow \exists d(> m_v \wedge tokenP(r, v, d) \vee tokeP(s, v, d))$$

which means that if v fires it consumes a token produced by r or s strictly more than m_v time units ago. If $m_v = 0$ this axiom excludes a -strictly- zero-time firing.

The axiom related to v's upperbound is:

UB(v) =

$$\begin{array}{rcl} fire(r) & \rightarrow & \exists d(d \leq M_v + \varepsilon \wedge tokenF(r, v, d)) \\ & \wedge & \\ fire(s) & \rightarrow & \exists d(d \leq M_v + \varepsilon \wedge tokenF(s, v, d)) \end{array}$$

where ε is a positive infinitesimal number. This is a short notation for:

$$\exists e \left(infinitesimal^{+}(e) \wedge \left(\begin{array}{l} fire(r) \rightarrow \exists d(d \leq M_v + \varepsilon \wedge tokenF(r, v, d)) \\ \wedge \\ fire(s) \rightarrow \exists d(d \leq M_v + \varepsilon \wedge tokenF(s, v, d)) \end{array} \right) \right)$$

Notice that the above axioms are the same as [FMM94] with the only addition of infinitesimal numbers.

The UB axiom is slightly more complex when two transitions compete to consume a token from a single place, as do transitions u and w in Figure 2.1. Let M be the least of the upperbounds of u and w, i.e., $M = min(M_u, M_w)$. The axiomatization of UB imposes the firing of either u or w within M time units after v.

UB(u), UB(w):

$$fire(v) \rightarrow \exists d(d \leq M + \varepsilon \wedge (tokenF(v, u, d) \vee tokenF(v, w, d)))$$

Finally we add an axiom stating token unicity (with x and y variables ranging on the set of transitions):

IU(v):

$$tokenP(x, v, d) \wedge tokenP(y, v, e) \rightarrow x{=}y \wedge d{=}e$$

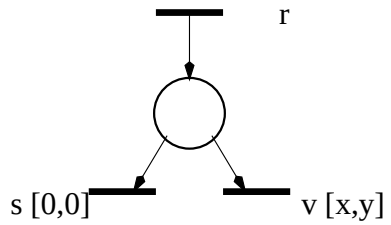


Figure 4.3: Two transitions in mutual exclusion.

OU(v):

$$tokenF(v, x, d) \wedge tokenF(v, y, e) \rightarrow x{=}y \wedge d{=}e$$

As a result we obtained an axiom system for TPNs with the same simplicity as for 1-bounded TPNs which applies however to general TPNs.

The examples given in the next section show the usefulness of the new axiomatization w.r.t. other approaches.

4.4 Proving system properties through the non-standard axiom system

In this section we provide a few examples of use of the new axiom system to prove system properties. Comparisons with previous approaches show how the proposed method joins naturalness with generality.

Example 4.1 We show that having increased by an infinitesimal quantity the lowerbound of a transition does not alter the order of firings.

Let us consider the net fragment in Figure 4.3, where x is any *standard* positive real number and y any real number $\geq$x. Then the following property holds

$$Alw(\neg fire(v)) \tag{4.1}$$

i.e., despite the (infinitesimal) increase in the upperbound of s, transition v will never fire.

This property was illustrated and proved in [FMM91], using a different axiomatization: there we could not avoid simultaneous transition firings, hence both the formalization of the behavior of the net and, as a consequence, the proof the property were much less intuitive and transparent. We were compelled to use the predicate *fireth*(v, i) to state that transition v fires *for the i-th time at the current instant*, and therefore we formalized the property as $Alw(\neg\exists i\ fireth(v, i))$. Similarly, in that axiomatization *tokenP*(*r*,*j*, *v*,*i*, *d*) would mean that transition *r* fires now (at the current instant) for the *j*-th time and the token produced by this firing will be consumed after *d* time units by the *i*-th firing of transition v. We report the proof based on the axiomatization of [FMM91] in the Appendix, and invite the reader to compare it with the new proof we display next. The latter is much more terse, though similar in structure, thanks to the use of simpler predicates and the absence of quantifications over the number of simultaneous transition firings. ■

Proof. The proof of (4.1) follows.

Axiom UB for transition s is:

UB(s): fire(r) $\rightarrow \exists$d (d $\leq \varepsilon \wedge$ (tokenF(r, s, d) $\vee$ tokenF(r, v, d)))

Let us assume, by contradiction, that transition v fires. Then, we can construct the following derivation.

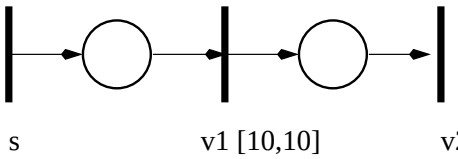


Figure 4.4: Two transitions firing at the same time.

1	Fire(v)	Hyp
2	∃d(d>x ∧ tokenP(r, v, d))	1, LB(v): Lower Bound axiom of v
3	D>x ∧ tokenP(r, v, D)	2, EI: Existential Instantiation: D for d
4	D>x ∧ Past(tokenF(r, v, D), D)	3, def: tokenP(x,y,d) =Past(tokenF(x,y,d),d)
5	D>x ∧ Past(fire(r), D)	4, def: tokenF(r,v,d) → fire(r)
6	D>x ∧ Past(∃e(e ≤ ε∧ (tokenF(r, s, e) ∨ tokenF(r, v, e)))), D)	5, UB(s) Upper Bound axiom f r s
7	D>x ∧∃e(e ≤ ε∧ Past(tokenF(r, s, e) ∨ tokenF(r, v, e), D))	6, th: Past(∃x A(x),d) = ∃x Past(A(x),d)
8	∃e(D>x ∧ e ≤ ε∧ Past((tokenF(r, s, e) ∨ tokenF(r, v, e)) ∧ tokenF(r, v, D), D))	7,4 AI And Introduction
9	(tokenF(r, s, e) ∨ tokenF(r, v, e)) ∧ tokenF(r, v, D) → D=e	OU(r) Output Unicity for r
10	∃e(D> x ∧ e ≤ ε∧ Past(D = e ,D))	8,9, MP
11	∃e(D> x ∧ e ≤ ε∧ D = e)	10, th: Past(A,x)→ A, if A is time independent
12	∃e(x< e ≤ ε)	11, AE And Elimination

Proposition 12 is false, since x is a positive standard real number, while ε is less than any positive standard. By contradiction, the initial assumption is therefore false.

□

Example 4.2 Consider the net fragment given in the Figure 4.4. We want to prove that

$$fire(s) \rightarrow WithinF(fire(v2), 10) \tag{4.2}$$

In this case it is immediate to realize that (4.2) cannot be derived as a theorem in our non-standard system. In fact the axioms UB given in Section 4.2 formalize the fact that, once s fires, $v2$ will fire in a right neighborhood of the instant at 10 time units after the firing of s, whereas (4.2) requires a firing of $v2$ within *exactly* 10 time units. In such cases, the user has the responsibility to state precisely whether timing properties to be proved must hold *exactly or up to an infinitesimal approximation*.

In this case, for instance, the "right" formula to be proved should be

$$fire(s) \rightarrow WithinF(fire(v2), 10 + \varepsilon) \tag{4.3}$$

where, for ease of reading, we use the short notation WithinF(fire(v), 10+ε) as an abbreviation for $\exists e$(WithinF(fire(v), 10 + e) ∧ infinitesimal+(e)).

Once it is understood that the wished property of the net of Figure 4.4 is (4.3) rather than (4.2), its proof with the new axiom system becomes a trivial exercise by exploiting the fact that the sum of two infinitesimals is infinitesimal. ■

Example 4.3 Consider the net fragment given in the Figure 4.5. It is interesting to note that the following property can be easily proved through a simple induction

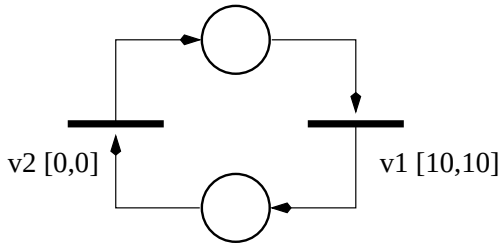


Figure 4.5: A transition loop.

$$fire(v1) \rightarrow \forall^{st}k(Futr(fire(v2), k * 10 + \varepsilon)) \tag{4.4}$$

■

Proof. From
1. *fire*(v2) → *Futr*(*fire*(v1), 10 + ε_1) and
2. *fire*(v1) → *Futr*(*fire*(v2), ε_2)
we find:
3. *fire*(v2) → *Futr*(*fire*(v2), 10 + ε_1+ε_2)
from which we derive:
4. *fire*(v2) → *Futr*(*fire*(v2), $10n + n \cdot \varepsilon_1 + n \cdot \varepsilon_2$) from which the thesis (4.4) follows (since n is standard and by Leibniz rules), taking $\varepsilon = n \cdot \varepsilon_1 + n \cdot \varepsilon_2$ □

Note Notice that the order of quantifications is intended as

$\exists e \forall^{st}$n (*infinitesimal*+(*e*) ∧(*fire*(*v*2)→ *Futr*(*fire*(v2), n(10 + e))).

From this we derive, thanks to basic properties of predicate calculus,

$\forall^{st} n \exists e(infinitesimal + (e) \wedge$(*fire*(*v*2) → *Futr*(*fire*(v2), n(10 +e))).

Then the Transfer theorem of IST allows us to derive

$\forall n \exists e(infinitesimal + (e) \wedge$ (*fire*(*v*2) → *Futr*(*fire*(v2), n(10 + e))),

whereas the formula with the other quantifier alternation, ∃e∀n(...), does not hold. This remark perfectly matches the intuition that, if we want to execute the loop of Figure 4.5 "an extremely large number of times" keeping the firing times "close to multiples of 10" we can always find a sufficiently short firing time for the single transition firings to fulfill the requirement; this property, however, does not generalize to "an infinite number of times".

4.5 Conclusions

In this chapter we have presented a new axiomatic approach that allows dealing with zero-time transitions in a way that is both intuitive and general. The approach is based on considering exact time bounds that are associated with transition firings as *approximations* of time measures up to infinitesimal numbers. As a consequence the user must apply some care in specifying system properties by clearly distinguishing whether some time values are exact or approximated numbers (in most practical cases it will turn out that we are dealing with approximate quantities).

Our short experience with the use of TRIO in a non-standard framework shows that extending the approach from the very basics presented in this chapter to the complete language (dealing with several derived operators, with large specifications and more complex proofs, ...) can proceed quite smoothly.

The approach has been formalized for timed Petri nets and the TRIO logic language but it can be applied as well to any abstract machine and logic assertion language.

For instance, our approach can provide a sound and complete explanation of the "arbitrary small constant " that is introduced in [HL96] as a "technicality to take into account of possible critical races": a close inspection shows that such a constant is but an infinitesimal positive quantity.

Furthermore, in those approaches, such as [BD91], [Cer93], and [Ost89], where time is formalized as a state variable updated by special "tick" transitions, time flow could be made implicit, as it is in traditional dynamic system theory, by associating a positive, possibly infinitesimal duration to every "normal" (non-tick) transition. This would permit the unification of the above approaches with other ones, such as [CPGK97], where time advances independently but a non-zero duration is associated with every transition.

Chapter 5

Verification of TRIO specifications

In the introduction we have already argued the importance of property verification and its support by means of automatic or semi-automatic tools. Although tool support wouldn't be necessary to the property verification, many experiences have shown how verification done by hand is error prone and of doubt value. For this reason formal verification and automatic verification are considered indissoluble. As a matter of fact every method for the verification is somehow now supported by tools, and much effort and time is spent to find more powerful algorithms and more capable programs. Although formal verification still suffers from both theoretical problems (e.g., problems of decidability) and practical problems (time and space necessary for the proofs), there has been recent progress in using model checkers and theorem provers to verify properties, and these techniques are applied in many industrial fields (for example hardware verification)[CW96]. In section 5.1 we present two main approaches for property verification: namely model checking and theorem proving. In section 5.2 we explain our motivations and why we chose the theorem prover PVS for our purposes, and we present our approach. In section 5.3 we present some related works.

5.1 Automatic property verification

Two well-established approaches to automatic (or semi-automatic) verification are model checking and theorem proving. They go one step beyond specification; these formal methods are used to analyze a system for desired properties, eventually to prove that from the specification requirements follow.

Model Checking.

Model checking is a technique that relies on building an finite model of a system and checking that a desired property holds in that model. Roughly speaking the check is performed as an exhaustive state space search which is guaranteed to terminate since the model is finite. The technical challenge in model checking is in devising algorithms and data structures that allow us to handle large search spaces [Cla91]. Model checking has been used primarily in hardware and protocol verification; the current trend is to apply this technique to analyzing specifications of software systems [HKL+98].

In contrast to theorem proving, model checking is completely automatic and fast, sometimes producing an answer in a matter of minutes. Model checkers' strongest point is that it produces counterexamples, which usually represent subtle errors in design, and thus can be used to aid in debugging. For this reason model checking is more successfully used more to find errors (for refusing wrong specifications) than to prove the complete correctness.

The main disadvantage of model checking is the state explosion problem. Many approaches have tried to to represent states or state transition efficiently: most used are the Bryant's ordered

binary decision diagrams (BDDs)[Bry85].

Although model checkers today are routinely expected to handle systems with a very large number of states, they are not able to deal with dense domain, unless using some abstractions.

Theorem Provers

Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. This logic is given by a formal system, which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system. Steps in the proof appeal to the axioms and rules, and possibly derived definitions and intermediate lemmas. While proofs can be constructed by hand, here, we focus only on machine-assisted theorem proving. Theorem provers are increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs.

Theorem provers can be roughly classified in a spectrum from highly automated, general-purpose programs to interactive systems with special-purpose capabilities. The automated systems have been useful as general search procedures and have had noteworthy success in solving various combinatorial problems. The interactive systems have been more suitable for the systematic formal development of mathematics and in mechanizing formal methods. In contrast to model checking, theorem proving can deal directly with infinite state spaces. It relies on techniques like structural induction to prove over infinite domains. Interactive theorem provers, by definition, require interaction with a human, so the theorem proving process is slow and often error-prone. In the process of finding the proof, however, the human user often gains invaluable insight into the system or the property being proved.

Since we consider time continuous (as real number), and since TRIO and TRIO specifications are normally complex and use mathematical expressions (like sum ...), we felt model checker not suitable to our goal and we addressed our attention towards theorem provers, with the purpose of building an interactive tool to support TRIO proofs otherwise done by hand.

5.2 Automatic verification of TRIO specifications

The first choice we had to take in providing automatic support to proofs in TRIO was that of a convenient formal theory: an encoding of TRIO formulas and the desired reasoning mechanisms (inference rules) in the language of the automatic tool. In [FMM94] we introduced a Hilbert like proof system, based on the use of modus ponens as the only inference rule, which is known to be well suited for studying the properties of a logic, but not for constructing readable proofs or for automatic theorem proving. To the purpose of automation, two principal kinds of proof system are used in practice: clausal form coupled with the resolution rule [WOLB92], and Gentzen-like systems [Pra65].

Resolution-based procedures find a proof by contradiction, deriving from the premises and the negation of the goal a huge number of consequences, until a contradiction is found. To improve efficiency, formulas are expressed in a very simple and rigid way, as clauses. This reduces the readability, and prevents the user from understanding the proofs or the reasons of their failure. We believe that this way it is not adequate to support validation and verification, where interaction with the user is fundamental. Gentzen systems, instead, favor the combination of a simple interaction with the prover (to direct the proof in the more complex cases) with automated solving of simpler subgoals by means of decision procedures. Gentzen systems include a set of inference rules that naturally correspond to the meaning of every operator. For instance, the sequent

$$\frac{\Delta \vdash \Gamma, A \quad \Delta \vdash \Gamma, B}{\Delta \vdash \Gamma, A \wedge B}$$

(where A and B are formulas, Δ and Γ are set of formulas and $\Delta \vdash \Gamma$ means that Γ is deducible from Δ) means that the formula $A \wedge B$ is deducible from a set of hypothesis, if and only if so are both A and B. This presentation is easily understandable, which helps significantly in designing and examining proofs. Besides, these inference rules can be easily used by a prover to decompose a proof into a tree of subgoals. For instance, the rule above can be used to reduce the deduction of $A \wedge B$ to those of A and B. Furthermore, if a subgoal fails because a counterexample can be found for it, the same counterexample falsifies also the original goal. For the above reasons, we chose a Gentzen-like axiomatization. The most available theorem prover we found, with a Gentzen-like deductive system and powerful decision procedures for arithmetic, was PVS [ORSvH95], that we chose for our first experiments [AGM97]. We kept working until we have obtained the tool we are going to present in this chapter.

Our tool has three components: a set of theories, containing the necessary definitions to encode in PVS the TRIO variables, operators, and definitions; 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.

5.2.1 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}
```

5.2.2 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 [Pau94], 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 very rich language and type system, and a very 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 has very powerful decision procedures over arithmetic and a very powerful prover. The semantic approach (with a dedicated pretty printer) was adopted for Duration Calculus in [SS94] and also in [DS97]. See section 5.3 and the web page of PVS (`http://pvs.csl.sri.com`) for further reference.

Some approaches try to combine the most valuable features of syntactic and semantic encodings: among these we mention Timed Automata [AH97]; in a previous, preliminary work [AGM97], 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

We have chosen to encode every time dependent term with values in a domain `D` as a function from time to `D`. This is implemented in PVS through a parametric theory:

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

The domain[1] `D` can be for example the integer set, and in this case we will have time dependent integer variables. The domain might also be a more complex type, such as a tuple, a record, or an abstract data-type or a function: in such cases we will have a time dependent tuple, a time dependent record and so on. A particular case is that of time dependent arrays, defined as time dependent functions from an integer range to a domain.

Several other proposals [DS97, SS94] adopted similar definitions, expressing time dependent entities as function from time, making this a rather standard way to represent temporal dependency in semantic encodings. As noted in [HRS98], considering system time dependent entities as functions from time to their domain, is used in conventional dynamic systems theory [Lue79], and such models are known to engineers in general, often through their graphical representation by timing diagrams.

Another interesting case is when `D` is the boolean set: then we obtain TD terms of boolean type, representing the TD predicates. As a matter of fact, the ability of PVS to deal in the same way with entity of any type, boolean included, allowed us to use unique definitions for predicates and for variables in a generic domain. For example there is a unique definition for point based entities, that can be either events (with boolean domain as defined in section 3.1) or point variables in a generic domain (as in section 3.4). Therefore, TD formulas are simply defined 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 an operator *op* on the domains $\mathtt{D}_1, \ldots, \mathtt{D}_n$ with result in the domain `D`:

[1] 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 [Rus97, ROS98]

```
op [D1,...,Dn] -> D
```

We would like to extend *op* to time dependent terms of the same type. Let a_1,...,a_n be temporal variables in those domains, i.e.,

```
a1: TD_Term[D1],..., an: TD_Term[Dn]
```

We define the extension of *op*, denoted as $\overline{op}$, as the operator producing as result a time dependent term *a* with domain `D` (i.e. `a: TD_Term[D]`), as follows:

```
op (a1,...an) : a(t) = op(a1(t),...,an(t)).
```

Hence the result of the application of $\overline{op}$ is still a function from time to the domain D, and this function has in every instant the value of the application of the operator *op* among the values of the arguments in that instant. We consider this the natural extension of *op* and we assign to $\overline{op}$ the same name as *op*. For example we have defined the sum for temporal dependent real numbers (or a subset thereof), such a *x* and *y*:

```
+(x,y)(t) : real = x(t)+y(t)
```

Similar definitions are provided for other mathematical and logical operations on time dependent entities. For example the definition of logical conjunction AND is as follows.

```
AND(A,B)(t) : bool = A(t) AND B(t)
```

Useful closure properties of those operators are proven. For example the fact that the sum of two time dependent integer is still an 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.

Temporal Operators

Besides usual domain operators, TRIO introduces several temporal operators. The basic temporal operator is *Dist*. In TRIO *Dist* is used to refer to values of variables in the future or in the past. For example, if *A* is a TRIO predicate, the formula *A* means that *A* holds now, whereas *Dist(A,d)* means either *A in d time units* (if d is positive) or *A d time units ago* (if d is negative). Therefore *Dist(A,d)* can be encoded as a 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)
```

[2]

The encoding of all the others 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:

```
Alw(A) : bool = FORALL (tt : Time) : A(tt)
Som(A) : bool = EXISTS (tt : Time) : A(tt)
```

Other derived temporal operators have as result a TRIO formula (i.e., a time dependent boolean term). For example, if A and B are TRIO formulas, then $\text{Lasts}_{ee}(A, d)$ and $\text{Until}_{ee}(A, B)$ are defined as follows.

[2] In PVS `lambda` expressions denote functions. For example the function which adds 3 to an integer may be written (see [OSRSC98]):
`LAMBDA (x:int): x+3`
and its type is function from integer to integer: `[int->int]`

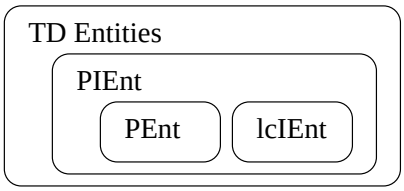


Figure 5.1: TD types in PVS

```
Lasts_ee(A,dur): TD_term[bool] =
  FA![3](it : {it:Time | 0 < it AND it < dur}):Dist(A,it)
```

which means A is true for dur time units in the future, and

```
Until_ee(A,B) : TD_term[bool] =
   EX! pt : Futr(B,pt) AND Lasts_ee(A,pt)
```

which means A is true in the future until B is true.

Definitions for entities with bound behavior

Relying on the definitions of TRIO variables and temporal operators, we defined the non Zeno requirement and introduced point and interval variables. The way followed is slightly different from that taken in Chapter 3. The main difference is that we have not introduced in PVS the concept of analytic function, because PVS has no predefined libraries for such a deep mathematical notion [Dut96], and building a complete library would have been far beyond the scope of the present work. Instead we have exploited simple but very general sufficient conditions (some of which mentioned in section), ensuring that variables with a simple temporal behavior (e.g., a ramp or a sinusoid) are non-Zeno.

Furthermore, as already noted, we provided unique definitions for predicates, formulas, and variables, using the parametric theories of PVS.

Another difference of our encoding with respect to the presentation of section 5.2.2 is that in PVS we have grouped non-Zeno point and interval variables in one type (piecewise constant variables) called `PIEnt,` with the aim to prove and use properties in common between these two types of behavior. The relation among types we have defined in PVS, is shown in Figure 5.1.

The definition for `PIEnt` is the following:

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

where `H` is a TRIO variable in a domain `D` and `l` and `r` are variables in `D`. Type `PIEnt` contains point and interval variables, as subtypes of time dependent variables.

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

```
PEnt(q) : NONEMPTY_TYPE =
{ PI : PIEnt | Som(UpToNow(PI=q)
       AND FORALL v : Alw(UpToNow(PI=v) <=> NowOn(PI=v))}
```

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

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

[3] `FA!` stays for `FORALL`,and `EX!` for `EXISTS`

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 may introduce either proof rules clashing with the original ones, or miss important inference rules. The proof of soundness and completeness of our encoding, i.e. that every PVS rule is valid in TRIO, and every TRIO axiom can be proved in PVS, is reported in [Jef95].

5.2.3 Proofs and Strategies

The primary purpose of our method and tool is to provide support to specification, validation, and verification of real-time systems using TRIO, especially in the early stages of the life-cycle. The main goal is to discover as many errors as possible from the beginning, because detection and correction of errors in the later stages is time consuming and costly. We believe that the best way to do that is to write a formal abstract specification and attempt proofs of desirable properties or requirements. We, among many others, have found that in this way errors in the specification and in the system can be easily discovered, even minor misunderstanding and inaccuracies. These faults can be corrected, to eventually obtain a completely correct specification and a set of critical requirements formally proven. For this reason proofs are very important and our tool (and PVS, of course), besides a very expressive language, pays a lot of attention in supporting the construction of readable proofs starting from a specification and provides a rich set of basic commands and more powerful strategies. We have extended 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: direc extensions of PVS strategies, and new strategies specifically aimed at dealing with time.

Extension of PVS strategies

First type are strategies defined as natural extension of PVS strategies. For example PVS' proof strategy `ground` takes the proof $\Gamma \vdash A \wedge B$[4] and transforms it to the pair of proofs $\Gamma \vdash A$ and $\Gamma \vdash B$.

A generalization of this strategy, called `trio-ground`, can be found in our system and, for example, takes a sequent $\Gamma \vdash Dist(A \wedge B, t)$ and makes the user prove two sequents: $\Gamma \vdash Dist(A, t)$ and $\Gamma \vdash Dist(B, t)$

In PVS *inference rules* 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: $\Gamma_1 \vdash \Delta_1 \ldots \Gamma_n \vdash \Delta_n$. In this notation the previous example of application of the rule `trio-ground` might be written:

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

Our tool of course supports all other rules based on propositional reasoning, including e.g. the case analysis, (using `case` and `trio-split` strategies).

[4] The proof $\Gamma \vdash A$, where Γ is a set of formulas and A is a formula, means the proof of A starting from (or supposing true) the formulas of the set Γ.

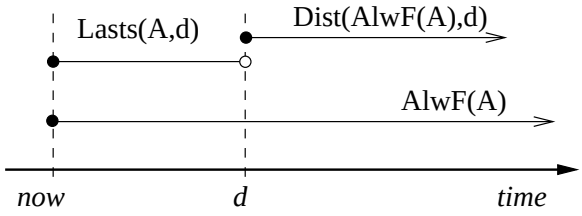


Figure 5.2: application of `(merge-temp-ops)`

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 strategy, named `(merge-temp-ops)` that is useful to merge temporal intervals. An instance application of the strategy is in the following derivation rule, illustrated in Figure 5.2 [5].

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

Temporal induction

Temporal induction deserves particular attention among time related strategies. The relation proposed in 3.5.3 defines the behavior of interval variables only step by step, or instant by instant. For an interval variable it specifies the next value according to the current value and the now happening events. How can we move from this step by step view to a reasoning about temporal intervals? This problem is similar to the well known problem of the mathematical induction.[6]

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 5.1 Temporal induction: *for every left continuous TRIO 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.*

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

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

Lemma 5.2 *If* v *is a time dependent variable whose behavior is defined by its change_relation as in section 3.5.3, then*

$$v = x \wedge \forall e \forall y(change\_relation(x, e, y) \rightarrow Lasts(\neg e, d)) \rightarrow Lasts(v = x, d)$$

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

[5] Notice that we assume *Lasts* and *Lasted* to denote intervals that are closed at the left and open at the right end, i.e., *Lasts* is defined as $Lasts_{ie}$ and *Lasted* as $Lasted_{ie}$.

[6] By the induction a statement about *i-th* state and *i+1-th* state can be generalized *for all i.*

GRC

For the GRC the previous lemma ensures the following property:

Corollary 5.3

$$gate = closed \wedge Lasts(\neg up, pt) \rightarrow Lasts(gate = closed, pt)$$

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.

5.2.4 The pretty printer

Besides a set of powerful strategies, to support proof, we have implemented a pretty printer for TRIO formulas and PVS sequents. As already mentioned, in semantic encodings, proofs may look very different from the original proofs in the source logic. In our case, using a semantic encoding with explicit time, current time shows up in the proofs, whereas a peculiarity of TRIO is to hide it. To hide time we have built a pretty printer that can be activated during proofs, and substitutes the original PVS display routines. During the proofs the pretty printer takes over the printing of the sequent, restructuring formulas to remove undesired information. Note that the pretty printer does not modify any PVS theory or reasoning algorithm, but only the standard PVS printing algorithm that is normally activated during proofs. Here we report a few examples that show its functionalities.

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`):

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

Hiding explicit time

For formula stated not at the current time, but at a given distance (say `t!1` time units[7]) 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)`

[7] 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`.

Temporal translation

Sometimes proofs are derived more easily using as point of view not 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.

5.2.5 Proofs for our case study

Our main goal was to test our system with the two desired and well known properties for GRC: safety and utility, here reported again. The safety is formalized by the following theorem:

Theorem 5.4 *Safety:*
$CTI > 0 \rightarrow gate = closed$

simply stating that if the number of trains in the critical region I is greater than 0, the bar is closed.

Here we report a sketch of the proof of this property. The first lemma used to prove safety, is the following:

Lemma 5.5 *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 reach the closed position), then at the end of that interval the bar will be closed. Intuitively, the Lemma holds because the bar takes no more than γ time units to become 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 5.3. 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 5.6 *$CTPI_\gamma$_is_safe :*
$CTI > 0 \rightarrow Lasted_{ie}(CTPI_\gamma > 0, \gamma)$

meaning that if there is a train in I, $CTPI_\gamma$ has been positive for γ time units, i.e. $CTPI_\gamma$ is at least equal to the number of trains that after γ time units will be in the region I. Therefore $CTPI_\gamma$ can be safely used to foresee the number of trains in I. This lemma is proved using the definition of $CTPI_\gamma$ and *CTI* (see page 33) and the temporal relations between *RI*, *II*, and *IO* (see page 40) 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 then 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 specification should satisfy this second requirement, *utility*, stated as follows:

Theorem 5.7 *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 conjoins in the premise of the utility property 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_open*) 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 prooved through intensive use of case analysis and temporal induction.

Note *Comparison with the previous approach.* Comparing our current approach with the previous one [AGM97], proofs are now significantly simpler and shorter. We needed 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 of each command, counting a complex command as the number of the 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 weight 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.

5.3 Related work

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 [SS94] 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). 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 [Sha93] systems evolve step by step and every step has a time associated with it. The same approach has been followed by [Hoo94], that extends Hoare triples with timing constraint in every assumption (pre-condition) and commitment (post-condition) pair. This approach is oriented to the verification of sequential systems like, for instance, computer programs. Recently PVS has been used also in [DS97], where the requirement analysis of a real avionic control system is conducted without relying on any formal language besides PVS; however the state variables used there are inspired to DC and data flows to LUSTRE [HLR92] and SIGNAL [BGJ91]. 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 function 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 [AH97]), 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 [DS97], 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.

5.4 Conclusions

In this chapter we presented an encoding of the TRIO logic and of the mentioned (in chapter 3) 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 [GLMZ96, BCC+95]. 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 [AGM97].

Although three tool components (theories, pretty printer, and strategies) are supposed to work together, they might be separately used. A user might use only definitions of derived entities, adopt explicit time and therefore use only PVS original strategies. Other users may use their own definitions and our pretty printer to hide the time in time dependent entities (provided that time dependent entities are defined as functions from time to their domain).

Chapter 6

Design of time critical systems through refinement

In section 2.3 we have already discussed the combined use of a operational language with a declarative language to specify and verify real-time systems in the so called dual-language approach (combining a temporal logic as TRIO and a operational graphical formalism as time Petri Nets). In this chapter we show how the same approach can be applied to the refinement of systems, defining what is exactly refinement and giving some (proved correct) refinements rules. The chapter is organized as follows. In section 6.1 we argument the usefulness of the refinement in designing and developing time critical systems, then in section 6.2 we formally define the notion of implementation for time critical systems in terms of provability of properties described abstractly at the specification level. We characterize this notion in terms of formulas of the temporal logic TRIO and operational models of time Petri nets, and provide a method (in section 6.3) to prove that two given nets are in the implementation relation. Refinement steps are often used as a means to derive in a systematic way the system design starting from its abstract specification. In section 6.4 we present a method to refine a system using a few simple but powerful rules. In section 6.5 we formally prove the correctness of the proposed refinement rules.

6.1 Introduction

Real time systems are required to manage their resources in a way that predictably satisfies some given timing constraints. Such systems are often embedded in critical applications such as patient monitoring systems, plant supervision systems, traffic control systems: their correctness is of primary importance, since their failure can have enormous costs and lead to unrecoverable damages. In the past years, the research on formal methods for the specification and verification of real time systems has been particularly active, especially in the field of temporal logic, resulting in the proposal of several specification formalisms and verification methods.

The proposed models are however rarely employed in the industrial development of such systems, where informal and semi-formal methods are still largely prevalent. One of the reasons for this unsatisfactory state of the art is that the systematic or algorithmic analysis techniques are very complex so that they cannot be scaled up to realistic systems. For instance the algorithms proposed for system verification and validation are often exponential in the size of the specification [AH90, FM92].

Often, however, the final specification of a (real time) system or its high level design are derived through a sequence of refinement steps. In each of these steps, starting from an abstract system description to be considered as its specification, one derives an *implementation*, i.e., a more detailed version that includes elements deriving from design choices but retains the required properties. If these repeated refinement steps are conducted in a systematic, careful way,

the verification activity needs not be repeated from scratch for each implementation step, since the system can be analyzed incrementally. The overall cost of the verification can thus be kept at a reasonable level by reusing in each step the results already obtained in the preceding phases.

In this chapter we address the problem of reducing the overall specification and design effort for real time system developed through a sequence of refinement steps. We report here the application of these ideas to the case when the real time system is abstractly modeled with time Petri nets (TPNs for short, already presented in section 2.1.1, a kind of Petri net where each transition is associated with a firing time interval describing its earliest and latest firing time after enabling) and its timing requirements are described by means of formulas written in TRIO (a temporal logic providing a metric on time distances, particularly suitable for specifying real-time systems). In this framework we formally define the notion of implementation among two time Petri nets: a net I acting as an implementation implements a net S acting as its specification, if it satisfies all the timing properties that are guaranteed by S. In previous works [FMM91, FMM94], as presented in section 2.3.1 at page 16, we defined an axiomatic system for TRIO and an axiomatization of time Petri nets that adequately copes with the salient features of this operational formalism, such as nondeterministic behavior, multiple simultaneous transition firings, zero-time and infinite-time transitions, and unbounded accumulation of tokens in places. Based on such axiomatization, in the present work we formally characterize properties of TPNs as TRIO theorems describing timing relations among their transition firings, and the notion of implementation among TPNs S and I in terms of a TRIO meta-theorem asserting that the theorems holding in the specification net can be proved (under a suitable translation) also in the implementation net. Furthermore we provide methods, based on sufficient conditions, to prove in several significant cases that two given nets are in the implementation relation.

The proof of implementation among time Petri nets, however, may not be performed algorithmically and it may require a certain amount of skill and ingenuity, because no general guideline can be provided for it and the TRIO axiomatization is not complete (we recall that the language includes arithmetic over the temporal domain). In a companion paper [FGP93] we introduced a set of refinement rules for time Petri nets which allow a designer to substitute a place or transition with a net fragment in such a way that the resulting net is a implementation of the original one. Here we provide systematic methods to formally prove the correctness of such refinement rules. Once the correctness of a rule has been formally proven, it can be applied to particular nets systematically or even automatically [PG92], since only the topological relations among net elements and the algebraic relations among the time bounds of the involved transitions must be checked.

By adopting this design method based on sequences of refinement steps, one can obtain nets that satisfy by construction all properties specified by the initial abstract version of the system. Moreover, the properties *inherited by refinement* can further be used as lemmas in the more detailed analysis of the final version of the system. In this way, when analyzing an implementation obtained through refinement, the analysis effort can be greatly reduced by performing the proof of intermediate lemmas on the first, more abstract and simple versions of the system.

The notion of refinement of Petri nets has already been studied in the literature [Vog86, SM83, Mül85] but, to the best of our knowledge, with reference only to classical (without time) Petri nets. Here, and in [FGP93], we propose new techniques specifically devoted to real time systems, thus employing time Petri nets and a temporal logic with a metric on time such as TRIO. We characterize properties that we wish to be preserved in implementations in a syntactic way, i.e., by means of TRIO formulas, whereas other approaches [FGP93, Vog89, Vog92, vGG90] adopt more semantic characterizations based on execution traces and behaviors. Under this respect, [DDGJ89] adopts a treatment closer to ours. because it uses the temporal logic MCTL (a modular extension of CTL) to describe properties of Petri net modules; this emphasis on modularity is also a major feature of the above-mentioned work on Petri nets [Vog92].

Other contributions that appeared recently in the literature are less closely related to the present work, in that they study the refinement operation in a different or broader context than (timed) Petri nets, referring to generic state-transition systems or to reactive systems without explicit and quantitative real-time constraints. Our definitions of implementation and refinement,

based on the ability to ensure at lower levels in the specification/implementation hierarchy the properties that are specified at the highest specification level, follows similar notions introduced in [Aiz89]. The approach proposed in [AL91] deals generically with any state-based machine, but does not take into account real-time aspects, since it considers just untimed sequences of machine states and focuses on safety and liveness properties; [AL91] also differs from our work in that the notion of correct refinement is defined in terms of mappings among states or behaviors, while we refer to properties explicitly expressed through logic formulas. Other approaches to the refinement operation [LA92, vGG90] greatly emphasize compositionality and modularity, both in the system structure and in the proof of its properties. The ideas on incremental analysis of refined systems presented in this chapter are strongly related to the notions of compositionality and incrementality reported in [YY91].

6.2 Implementation relation among TPNs

In this section we characterize the notion of implementation relation among TPNs. An implementation must satisfy, by its very definition, all the requirements expressed in the specification, hence we say that a time Petri net I implements a time Petri net S acting as a specification if all the properties satisfied by net S are also ensured by net I. To make this precise we must therefore provide the following items of information: 1. what kind of properties of TPNs we require implementations to preserve; 2. how these properties are to be ensured by the implementation net (in other words, if net S has property π , which is the property ψ that must hold in I?); 3. the precise conditions, according to the above items, under which two given nets S and I are in the implementation relation. We answer these questions by an informal description in the paragraphs below, and provide formal definitions subsequently.

1. In our view the properties ensured by a time Petri net are the temporal relations among transition firings that are verified in *every* execution of the net. We therefore assume that transition firings are the only observable events of the net: the state of a TPN, as usually defined in terms of place marking and age of the tokens, is assumed to be an internal feature, not accessible to the observer. Properties of this kind are naturally described by means of TRIO formulas. These assumptions are formally stated in Definition 6.1 (*Observable property*) below.

2. An obvious notion of implementation would require that the properties to be satisfied by net I are exactly the same that are ensured by S; this implies that every transition of net S corresponds to exactly one transition of I that *implements* it. Although this is a plausible choice, we consider it too restrictive, since we would like to consider also the case, frequently encountered in practice, where a single action a in the high-level system description is implemented by several, actions a_1,... a_n, that in the low-level description can occur in a mutually exclusive fashion depending on some condition that for abstraction purposes is ignored in the high-level version. Therefore in our approach every transition t of net S corresponds, in net I, to a set of transitions t_1,... t_n, $n \geq 1$, as specified in Definition 6.2 (*Event function*); furthermore, if net S satisfies a formula asserting that t fires, then in net I a formula must hold, asserting that one of t_1,... t_n fires, as stated in Definition 6.3 (*Property function*).

3. The precise definition of the implementation relation follows directly from the above items: nets S and I are in the implementation relation if the formulas describing properties of S as in 1. above, translated as described in 2. above, are satisfied by net I. This is precisely what stated in Definition 6.4 (*Implementation relation*),

Next we define formally the formulas describing an observable property of a TPN, the correspondence among transitions in TPNs associated in the implementation relation, and the implementation relation itself.

Definition 6.1 Observable formula and property. *Given a time Petri Net N, an observable formula for N is a TRIO formula constructed on the time dependent predicate* nFire *(and on the derived predicate* fireth*) applied to transitions in T_N, plus the usual arithmetic predicates and functions on the temporal domain. An observable* property φ *is an observable formula that can be derived as a theorem for theory $\mathcal{N}$, i.e., such that $\vdash_{\mathcal{N}} \varphi$ holds.*

The definition of observable property refers only to the *nFire* predicate because we assume that transition firings are the only observable events in the net. The other fundamental predicate in the axiomatization of time Petri nets, namely predicate *tokenF*, is related to the topology of the net and models the cause-effect relation among transition firings: such information concerns the mechanism of token production and consumption, so we consider it immaterial for the implementation relation.

Definition 6.2 Event function. *Given two TPNs S and I, an event function from I to S is any onto function $\lambda : T_I \rightarrow T_S$ from the transition of I to the transitions of S.*

An event function from net I to net S specifies which transition of I represents transitions of S in a (possible) implementation relation. Notice that λ may be partial, since the net I may add details (transitions) that are not present in S, but it is required to be onto, because *every* property of S must be ensured by I, which implies that every transition of S is represented by some transition in I; furthermore, λ is not required to be one to one: a single transition of S may well be represented by more than one transition of I (e.g., when an action of S is implemented by two exclusive actions of I).

Definition 6.3 Property function. *Given two TPNs S and I and an event function λ from I to S, a property function Λ (uniquely determined by λ) from $\mathcal{S}$ to $\mathcal{I}$ is a function that translates observable formulas of S into observable formulas of I, according to the following requirements* [1]

i $\Lambda(nFire(v,n)) = \exists n_1 ... \exists n_s (n_1 + ... + n_s = n \wedge nFire(v_1, n_1) \wedge ... \wedge nFire(v_s, n_s))$ where $\{v_1, ... v_s\} = \{t \in T_I \mid \lambda(t) = v\}$ is the set of transitions net I into which v is refined;

ii $\Lambda(P(t_1...t_n)) = P(t_1...t_n)$ for any other atomic formula (i.e., when P is '=' or '<');

iii $\Lambda(\alpha \rightarrow \beta) = \Lambda(\alpha) \rightarrow \Lambda(\beta)$

iv $\Lambda(\neg\alpha) = \neg\Lambda(\alpha)$

v $\Lambda(Dist(\alpha, t) = Dist(\Lambda(\alpha), t)$

vi $\Lambda(\forall x\ \alpha) = \forall x\ \Lambda(\alpha)$

Note (*On function* Λ). Notice that from clause i. of Definition 3 above, it follows that
$\Lambda(nFire(v,0)) = nFire(v_1, 0) \wedge ... \wedge nFire(v_s, 0)$ if $\{v_1, ... v_s\} = \{t \in T_I \mid \lambda(t) = v\}$;
furthermore, if $\{v_1\} = \{t \in T_I \mid \lambda(t) = v\}$, that is, v_1 is the only transition in net I corresponding to transition v in net S, then
Λ (*nFire*(v, k)) = *nFire*(v_1, k) for any integer number *k*.
The above properties simplify the translation of formulas via Λ in case a transition does not fire or the correspondence among transitions is one-to-one.

The above introduced notations allow us to formally characterize the implementation relation among TPNs.

[1] Since any TPN has only a finite set of transitions, we assume without loss of generality that only transtion constants appear in observable formulas; if a formula contains a variable representing a transition (necessarily quantified, since we consider only sentences), the quantification can be translated into a finite conjunction or disjunction of formulas where only transition constants occur.

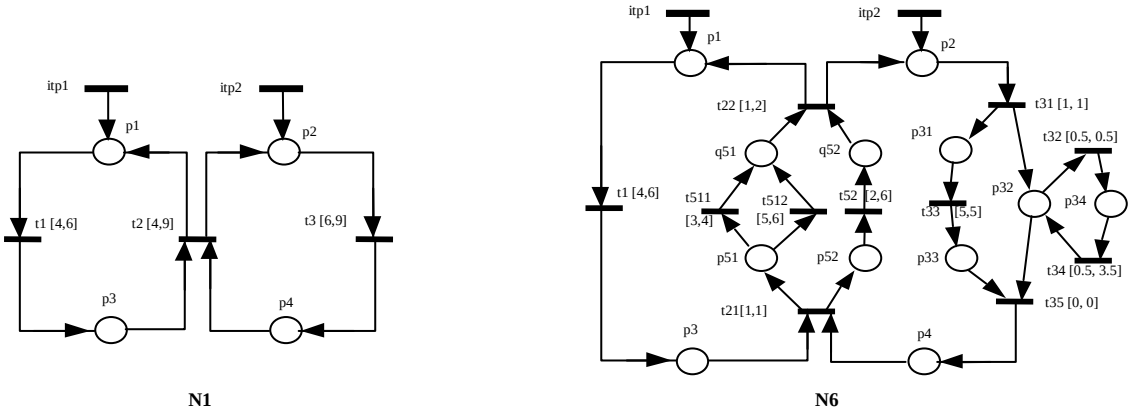


Figure 6.1: A simple TPN (N1) and a more complex net (N6) implementing it.

Definition 6.4 Implementation relation among TPNs. *Given two TPNs S and I and an event function λ from I to S (and accordingly a property function Λ from $\mathcal{S}$ to $\mathcal{I}$), we say that I implements S* through λ *iff, for each observable formula φ of S, $\vdash_{\mathcal{S}} \varphi$ implies $\vdash_{\mathcal{I}} \Lambda(\varphi)$, i.e., the translation by Λ of every observable property of S is an observable property of I. We say that I implements S iff there exists an event function λ from I to S such that I implements S through λ*

It can be easily shown that the implementation relation as defined above is transitive: given nets N_1, N_2, and N_3, if N_2 implements N_1 through event function λ_1 and N_3 implements N_2 through function λ_2, then N_3 implements N_1 through the event function $\lambda_1 \circ \lambda_2$. This property can be the base for a design methodology where the implementation is obtained in terms of a sequence of *refinement* steps.

Example 6.1 As a trivial example of implementation relation among time Petri nets, let us consider two TPNs S and I having the same topology and initial marking (i.e., T_S=T_I, P_S=P_I, F_S=F_I, and m_S=m_I) and such that the time interval of every transition in I is stricter than that of the same transition in S (i.e., $\forall$ v $\in T_I$, if we let Θ_I(v)=[m_{vI}, M_{vI}] and Θ_S(v)=[m_{vS}, M_{vS}] then $m_{vI} \geq m_{vS}$ and $M_{vI} \leq M_{vS}$). Then it can be easily proven that net I implements net S. In fact, every axiom $\alpha_S \in$ Ax(S) for net S is logically implied by the corresponding axiom $\alpha_I \in$ Ax(I) in I, i.e., $\vdash \alpha_I \rightarrow \alpha_S$, because of the stricter time bounds. Then net I implements net S through the identity event function, so that Λ is the identity function on the observable formulas of net S. In fact, for any observable formula φ , $\vdash_{\mathcal{S}} \varphi$ implies $\vdash_{\mathcal{I}} \varphi$ thanks to the following property of the TRIO axiomatic calculus (and in fact of any first order calculus): if $\Gamma, \alpha \vdash \varphi$ and $\vdash \beta \rightarrow \alpha$ then $\Gamma, \beta \vdash \varphi$. ■

Example 6.2 As a more concrete example[2] of implementation, consider the net fragments N_1 and N_6 shown in Figure 6.1. N_1 models a simple rendezvous between a producer and a consumer. The producer gets data (e.g., temperature, pressure) from an external device, in 4 to 6 time units (transition t1), and then, in 4 to 9 t.u., communicates the acquired data to the consumer who is responsible for the elaboration (transition t2). The elaboration by the consumer takes 6 to 9 t.u. (transition t3). Initially, the producer is ready for the acquisition and the consumer is ready for elaboration (places p1 and p2 are initially marked: notice initializing transitions itp1 and itp2). In the next section we will prove that the net N_6, where several elements modeling design choices have been added to those of N_1, implements it, with event function λ defined as $\lambda(t_1) = t_1$, $\lambda(t_{22}) = t_2$, $\lambda(t_{35}) = t_3$, and $\lambda(t)=\perp$ for every other transition $t \in T_{N6}$. ■

[2]The example is borrowed, with modifications, from [FGP93].

6.3 A method for proving implementation

Proving the existence of an implementation relation among two arbitrary TPNs can be complex and difficult. First of all, the choice of the correct event function may be non trivial, since the space of such function is rather large. Furthermore, given an event function, proving the implementation relation requires the proof of a meta-theorem on the derivability of a set of formulas, and no precise guideline on how to structure such proof can be given if no additional information is available on the relation among the two given nets. This situation is further complicated by the fact that (timed) Petri nets are rather unstructured mathematical objects: in general they are just bipartite graphs having no *a priori* constraint on the adjacency relation among nodes.

In the following we present a method for proving implementation among TPNs that can be easily applied when the implementation mechanism is intuitively clear, as it is in the case of an implementation net obtained through systematic transformation of a given specification net. The method is based on the idea that for each observable property π of specification net S there exists in the axiomatization of the implementation net I a proof of $\Lambda(\pi)$ that mirrors the proof of π in Ax(S). We therefore introduce a *proof* function Δ that translates formulas of theory $\mathcal{S}$ (referring to the transitions of S) included in the derivation of $\vdash_{\mathcal{S}} \pi$ into formulas of theory $\mathcal{I}$ (referring to transitions of I) in such a way that if the derivation of $\vdash_{\mathcal{S}} \pi$ consists of the formulas π_0, π_1... π_n (where $\pi_n = \pi$), then the proof of $\vdash_{\mathcal{I}} \Lambda(\pi)$ includes formulas $\Delta(\pi_0)$, ... $\Delta(\pi_1)$, ... $\Delta(\pi_n) = \Delta(\pi)$ plus possibly other ones. The proof of π for net S uses axioms of Ax(S) that describe the cause-effect relations among transition firings and therefore include occurrences of the *tokenF* predicate. The proof translation function must therefore be defined on any formula of theory $\mathcal{S}$, not only on observable formulas. Finally, notice that the requirements expressed above for the proof translation function Δ imply that $\Delta(\pi) = \Lambda(\pi)$, i.e., Δ must be an extension of the property function Λ . As it will be apparent in the following, Δ is therefore essentially characterized by the way it translates the tokenF predicate.

The systematic translation of the derivation of $\vdash_{\mathcal{S}} \varphi$ into the derivation of $\vdash_{\mathcal{I}} \Lambda$ (φ) is formalized by the notion of *proof* (translation) function, to be defined next.

Definition 6.5 Proof function. *Given two TPNs S and I, with functions λ and Λ as in Definition 3, a proof (translation) function Δ from $\mathcal{S}$ to $\mathcal{I}$ is a function that translates any well formed formula (wff) of $\mathcal{S}$ into a wff of $\mathcal{I}$ satisfying the following conditions.*

i. Δ is an extension of Λ (i.e., it is equal to Λ where they are both defined), and

ii. Δ is compositional with respect to the structure of the formulas:

$\Delta(\alpha \rightarrow \beta) = \Delta(\alpha) \rightarrow \Delta(\beta)$; $\Delta(\neg\alpha) = \neg\Delta(\alpha)$

$\Delta(Dist(\alpha, t)) = Dist(\Delta(\alpha), t)$; $\Delta(\forall x\ \alpha) = \forall x\ \Delta(\alpha)$.

The proof function is extended to sets of formulas in an obvious way: for a set Γ of wffs of theory $\mathcal{S}$, we define $\Delta(\Gamma)$ as the set $\{\Delta(\gamma) \mid \gamma \in \Gamma\}$ including precisely the translation according to Δ of all wffs of Γ.

The following proposition illustrates some (immediately proved) properties of the proof function.

Proposition 6.6 *If Δ is any proof function from wffs of a TRIO theory $\mathcal{S}$ to wffs of a TRIO theory $\mathcal{I}$, then*

a. Δ is congruous with modus ponens, i.e., $\Delta(\alpha)$, $\Delta(\alpha \rightarrow \beta) \vdash_{\mathcal{I}} \Delta(\beta)$;

b. TRIO axioms are maintained by Δ, i.e., the translation of each TRIO axiom of theory $\mathcal{S}$ is a theorem of theory $\mathcal{I}$. n

The following meta-theorem provides the basis for our principal method of proving implementation among TPNs.

Theorem 6.7 (*meta-theorem*) *Let S and I be two TPNs, $\mathcal{S}$ and $\mathcal{I}$ the two TRIO theories describing them, Δ a proof function from $\mathcal{S}$ to $\mathcal{I}$. Then for every wff φ of $\mathcal{S}$, $\vdash_{\mathcal{S}} \varphi$ implies $\Delta(Ax(S)) \vdash_{\mathcal{I}} \Delta(\varphi)$.*

Proof. The proof of the meta-theorem is by induction on the length of the derivation of φ in $\mathcal{S}$. Let $\varphi_o, \varphi_1, ..., \varphi_n = \varphi$ be a derivation of φ in $\mathcal{S}$; then for each i, with $0 \leq i \leq n$, $\Delta(\text{Ax(S)}) \vdash_{\mathcal{I}} \Delta(\varphi_i)$.

Base step. If φ_o is a TRIO axiom then $\vdash_{\mathcal{I}} \Delta(\varphi_o)$ by Proposition 6.6(b);

If $\varphi_o \in$ Ax(S), then $\Delta(\varphi_o) \in \Delta(\text{Ax(S)})$, hence $\Delta(\text{Ax(S)}) \vdash_{\mathcal{I}} \Delta(\varphi_o)$.

Induction step. Let us assume that $\Delta(\text{Ax(S)}) \vdash_{\mathcal{I}} \Delta(\varphi_j)$ for each $j < i$; then

if $\varphi_i \in$ Ax(S) or φ_i is a TRIO axiom, then $\Delta(\text{Ax(S)}) \vdash_{\mathcal{I}} \Delta(\varphi_i)$ as in the base case;

if φ_i is obtained by modus ponens from two preceding formulas φ_h and $\varphi_k = \varphi_h \rightarrow \varphi_i$, with h, $k < i$, then by the induction hypothesis $\Delta(\text{Ax(S)}) \vdash_{\mathcal{I}} \Delta(\varphi_k)$ and $\Delta(\text{Ax(S)}) \vdash_{\mathcal{I}} \Delta(\varphi_h \rightarrow \varphi_i)$ and the thesis follows from Proposition 6.6(a). □

The preceding meta-theorem shows that a proof translation function provides, as its name suggests, a way to obtain, from a proof of φ in $\mathcal{S}$, a proof of $\Delta(\varphi)$ in $\mathcal{I}$ from Δ(Ax(S)).

Given two sets of wffs Γ and Ψ, with a slight abuse of notation we shall write in the following $\Gamma \vdash \Delta(\Psi)$ meaning that $\Gamma \vdash \Delta(\psi)$ for each $\psi \in \Psi$ or, equivalently, that $\Gamma \vdash \bigwedge_{\psi \in \Psi} \Delta(\Psi)$.

Corollary 6.8 (to meta-theorem 6.7). *In the same hypotheses as in meta-theorem 6.7, if $\vdash_{\mathcal{I}} \Delta(Ax(S))$ (i.e., all axioms of $\mathcal{S}$, translated into $\mathcal{I}$, are theorems) then net I implements S (i.e., for each observable formula φ, $\vdash_{\mathcal{S}} \varphi$ implies $\vdash_{\mathcal{I}} \Lambda(\varphi)$).*

Corollary 6.8 provides a sufficient condition for implementation: to prove that a TPN I implements a TPN S through event function λ, it suffices to find a proof function Δ such that the axioms of net S, translated through Δ, are theorems of the theory $\mathcal{I}$ of TPN I.

Example 6.3 We apply corollary 6.8 to the TPNs N1 of Figure 6.1 and N2 of Figure 6.2; N2 implements N1 through event function defined as follows: λ(t22)=t2, t21 $\notin$ Dom(λ), and λ(x)=x $\forall$ x $\in T_{N2}$-t21, t22. Implementation can be proved by taking Λ defined in the obvious way, and Δ extending Λ as follows: for all i, j, d,

$$\Delta(tokenF(t_1, i, p_3, t_2, j, d)) = \exists d_1 \exists d_2 \left(d_1 + d_2 = d \wedge \exists k \left(\begin{array}{c} tokenF(t_1, i, p_3, t_{21}, k, d_1) \\ \wedge \\ Futr(tokenF(t_{21}, k, p_5, t_{22}, j, d_2,), d_1) \end{array} \right) \right)$$

$\Delta(tokenF(t3, i, p4, t2, j, d))$ is defined similarly, and

$\Delta(tokenF(u, i, p, v, j, d)) = tokenF(\lambda^{-1}(u), i, p, \lambda^{-1}(v), j, d)$ for any other pair of transitions u and v[3].

Intuitively, the proof function describes how any execution of net N1 can be simulated by an execution of net N2. In net N1 a token produced by a firing of transition *t1* (or *t3*) can be consumed *d* time units later by a firing of transition *t2*, then in net N2 a token produced by a firing of *t1* (or *t3*) can be consumed by a firing of *t21* occurring *d1* after time units and this firing produces a token that can be consumed by a firing of *t22* *d2* time units later, with *d1*+ *d2*= *d*. All other events of token production and consumption taking place in net N1 can be simulated directly by net N2. The adequacy of the above defined proof function to prove that N2 implements N1 will be discussed, in a more general setting, in section 6.4. ■

Note *On initial marking.* The above described method for proving implementation can consider also the initial marking of the TPNs, despite the fact that it refers exclusively to the transitions and their firings. The reason for this is precisely the fact that the initial marking of places is defined, in section 2.3.1, in terms of suitable firings of special initializing transitions. In general, if place p in net S is initially marked, then an axiom of $\mathcal{S}$ called IM(p) describes the firings of the initializing transition itp_s. If a second net I implements S then it must have a transition ipt_i, corresponding to itp_s, which fires in such a way that Λ(IM(p)) is a theorem of $\mathcal{I}$.

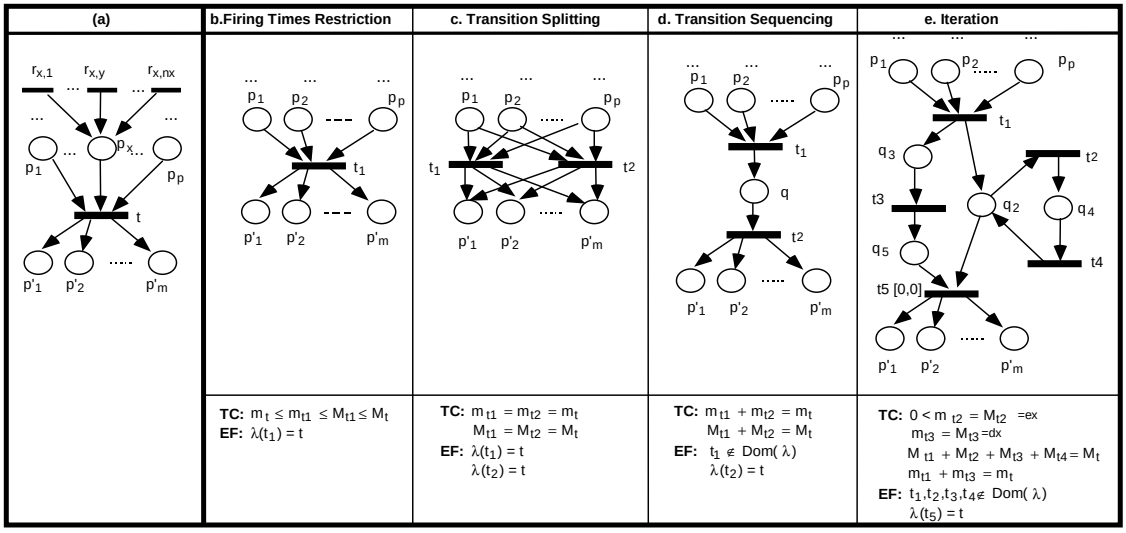


Table 6.1: Transition refinement rules

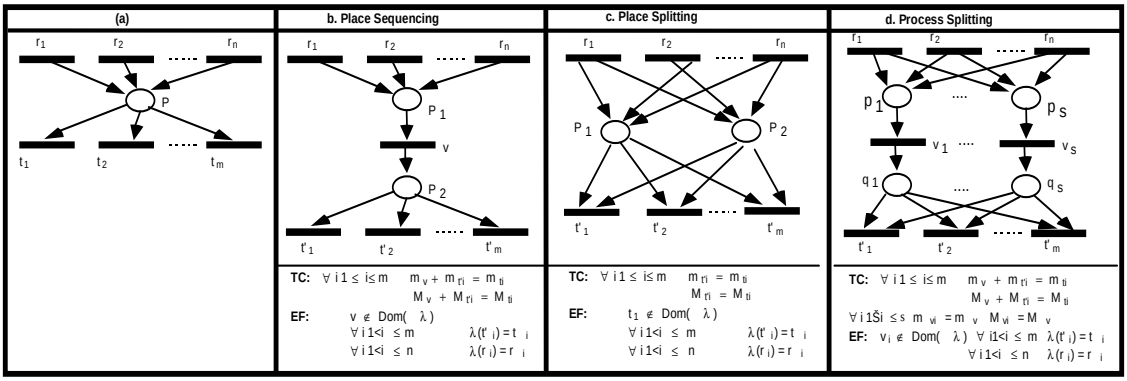


Table 6.2: Place refinement rules

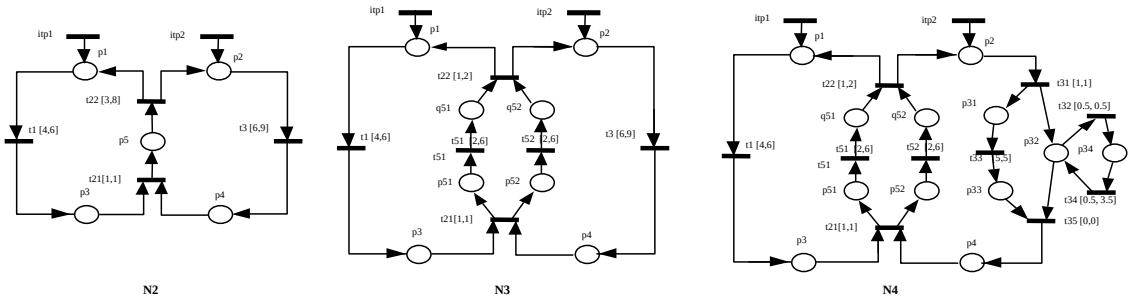


Figure 6.2: A sequence of refinement steps.

6.4 Implementation through refinement

An incremental approach to the specification and design of time critical systems through time Petri nets and TRIO can avoid many of the difficulties in proving implementation outlined in section 6.3 and in general it can greatly reduce the overall development effort. In the following we present a set of *refinement rules* that, given a TPN, permit the substitution of one of its components, be it a transition or a place, with a net fragment composed of a combination of new places and transitions in such a way that the new net retains the properties of the initial net and is therefore an implementation. The correctness of each of these refinement rules can be proved in a general way, independently of particular net instances, under the hypothesis that the firing time intervals of the newly introduced transitions satisfy suitable constraints with respect to those of the transitions composing the original net.

An incomplete sample of such refinement rules is represented in Tables 6.1 and 6.2. Each table reports in the first column the net fragment where the substitution is performed and in the other columns the net resulting from the rule application. For each rule, the table also provides timing constraints (TC) among the time bounds of the involved transitions, and the event function (EF) for such transitions (since the rest of the net is unchanged, λ(v)=v for every other transition).

Table 6.1 displays *transition* refinement rules, where a single transition t in net S is replaced in net I by a net fragment; for all such rules (except the transition splitting rule of column c) it is required that in net S there is no transition conflicting with t. Table 6.1 displays *process* refinement rules, where the refined component in net S is a place p; for all such rules (except the process splitting rule of column c) it is required that no transitions t_1,...,t_m of $p^\bullet$ is in the postset of any other place.

Table 6.1(d) describes the *transition sequencing* rule, that substitutes a given transition t with a sequence of two transitions t_1 and t_2 and a place p among them representing the start and the end of the action modeled by the original transition t. We apply this rule to transition t2 of net N_1 in Figure 6.1 obtaining the net N_2 of Figure 6.2. The action representing the communication between the producer and the consumer is detailed in two actions (transitions t21 and t22) representing the start and the end of the communication, respectively. N_2 implements N_1 through the event function λ_1:

λ_1(t1) = t1; λ_1(t3) = t3; λ_1(t22) = t2, λ_1(t21) = $\perp$.

Table 6.2(d) presents the *process splitting* rule: a place p and the transitions t_1, ... t_m in its postset are replaced by a set of transitions v_1, ... v_s having p_1, ... p_s as preset places and q_1, ... q_s as postset places, with each of the q_i's being in the preset of every transition of t_1', ... t_m'. Intuitively, the application of this rule introduces a set of s processes that evolve in parallel, being synchronized at the start and at the end of their evolution. This rule, applied to N_2 in Figure 6.2, yields the net N_3. Place p5, representing the communication being held, is refined into two processes (p51 and p52 with transitions t51 and t52) in net N_3 to model the actions executed in parallel during the communication by the producer and the consumer. N_3 implements N_2

[3]Notice that in this case λ^{-1} is defined because λ is one-to-one.

through event function λ_2:

$$\lambda_2(tx) = tx \qquad \forall tx \in \{t_1, t_{22}, t_3, t_{21}\}, \qquad \lambda_2(t_{51}) = \lambda_2(t_{52}) = \perp.$$

Table 6.2(b) shows the *place sequencing* rule: a place p and the transitions $t_1, \ldots t_m$ in its postset are replaced by places p_1 and p_2, transition v and a set of transitions t'_i each corresponding to one of the transitions $t_i \in p^\bullet$.

Table 6.2(c) shows the *place splitting* rule: a place p is replaced by places p_1 and p_2, with the same preset and postset as p.

Table 6.1(e) presents the *iteration* rule. It consists of substituting a transition t with a set of transition and places modeling repeated firings of a transition with a time upper bound equal to that of the original transition t. Applying this rule to transition t3 of N_3 of Figure 6.2 leads to the net N_4. The consumer's behavior is described by two processes: an iteration executing the computation, controlled by a time-out. Now the event function is:

$$\lambda_3(t_{35}) = t_3 \qquad \lambda_3(tx) = tx \qquad \forall tx \in \{t_1, t_{22}, t_{51}, t_{21}\}$$

$$\lambda_3(t_{31}) = \lambda_3(t_{32}) = \lambda_3(t_{33}) = \lambda_3(t_{34}) = \perp$$

Table 6.1(c) presents the *transition splitting* rule, where a transition t is split into two transitions t_1 and t_2 having the same preset and postset and the same firing time interval as t. This rule can be applied to model that the original action is implemented by two alternatives. The net N_5, obtained by applying this rule to transition t51 in the net of N_4 of Figure 6.2, has the same topology as the net N_6 in Figure 6.1, with different time intervals for transitions t511 and t512, which are associated with the interval [2,6] as the original transition t51. Transition t51 of net N_4, representing actions performed by the producer, can be further refined into two transitions (t511,t512) representing two exclusive actions (e.g., an "if" inside the code of the producer). The net N_5 implements N_4 with the event function:

$$\lambda_4(t_{511}) = \lambda_4(t_{512}) = t_{51} \qquad \lambda_4(t_x) = t_x \ \ for\ the\ other\ transitions$$

Table 6.1(a) describes, finally, the *firing times restriction* rule: a given transition t is substituted by a new transition t_1 having a restricted firing time interval. This rule formalizes the concepts presented in Example 6.2. Applying this rule to the transitions t511 and t512 just obtained yields the net N_6 of Figure 6.1, where t511 and t512 represent actions having different timings. N_6 implements N_5 with the identity event function: $\lambda(tx) = tx$.

In conclusion, the net N_6 implements the net N_1 through the event function $\lambda = \lambda_5 \circ \lambda_4 \circ \lambda_3 \circ \lambda_2 \circ \lambda_1$

6.5 Proving correctness of refinement rules

In this section we show how Corollary 6.8, which suggests a method for proving the implementation among TPNs, can also be used for the proof of correctness of the refinement rules.

To provide such proof, we must show that every net *b... e* in Tables 6.1 and 6.2, obtained applying a refinement rule, implements the specification net, the net *a* in those tables. From this point on we call net *a* also net S, because it acts as specification net, whereas nets *b.. e* will be called net I, because they are implementation nets.

First, we define for every rule a different *proof* translation function Δ, that translates every formula for the specification net S in a formula for the refined net I. Function Δ must be an extension of the *property* function Λ. Λ is characterized in Definition 6.3 from the *event* function

λ described in the tables. The proof function Δ must be defined on every predicate of net S, included tokenF; however, since Δ is required to be extension of Λ , it is essentially characterized by the way it translates tokenF predicate.

Then, according to Corollary 6.8, to prove that the refined net is an implementation we show that the axioms of net S, translated through Δ, are theorems of theory $\mathcal{I}$, i.e. $\vdash_{\mathcal{I}} \Delta(\mathrm{Ax(S)})$.

In following sections we propose for the refinement rules one possible definition (the more intuitive one) of the proof function Δ, and we explain briefly the intuitive meaning of the translation. The functions thus defined are adequate to prove the correctness of refinement rules; complete proofs of correctness are reported in [FGM95].

Notice that for simplicity we drew only in table 6.4 the transitions in the preset of each of the places $p_1 ... p_p$ that are in the preset of transition t. We will assume that each place p_x, with $x \in [1..p]$, has n_x transitions in its preset. In the following of sections we will call $r_{x,y}$, with $x \in [1..p]$ and, for each x, for $y \in [1..n_x]$ the y-th transition in the preset of the x-th place in the preset of transition t.

Firing times restriction rule

In this case the Δ function is very simple.

$$\Delta(tokenF(u,i,p,v,j,d)) = \\ = tokenF(\lambda^{-1}(u),i,p,\lambda^{-1}(v),j,d) = tokenF(u,i,p,v,j,d) \text{ if } u \neq r_{x,y} \text{ or } v \neq t;$$

$$\Delta(tokenF(r_{x,y},i,p_x,t,j,d)) = tokenF(r_{x,y},i,p_x,t1,j,d)$$

A token produced by $r_{x,y}$ and consumed by t is translated into a token produced again by $r_{x,y}$ and consumed by t1.

Transition sequencing rule

The proof translation function applied to the *tokenF* predicate is defined as follows.

$$\Delta(tokenF(u,i,p,v,j,d)) = tokenF(\lambda^{-1}(u),i,p,\lambda^{-1}(v),j,d) \text{ if } u \neq r_{x,y} \text{ or } v \neq t;$$

$$\Delta(tokenF(r_{x,y},i,p_x,t,j,d)) = \\ = \exists h \exists d'(tokenF(r_{x,y},i,p_x,t1,h,d') \wedge Futr(tokenF(t1,h,q,t2,j,d-d'),d'))$$

In words, the token produced by the transition $r_{x,y}$ and consumed by t after d time units corresponds in net I to two tokens, one produced by $r_{x,y}$ and consumed by t1 after d' and the other produced by the same firing of t1 and consumed by t2 after d time units.

Transition splitting rule

The translation of the tokenF predicate in this case is not immediately defined as in the preceding rules. All trivial translations (which for brevity we do not discuss here: the interest reader is referred to [Gar95]) are not adequate to prove the correctness of the rule, either because the sentences obtained by translation of the axioms of net S are either too restrictive on the behavior of net I (and hence false) or because they happen to be totally unrelated with it. It turns out that to permit proof of correctness of the transition splitting rule the Δ function should be defined in such a way that it sets a one-to-one correspondence between the firings of transition t on one side and those of transitions t1 or t2 on the other side.

One definition of function Δ on the tokenF predicates that satisfies this constraint is the following.

$$\Delta(tokenF(u,i,p,v,j,d)) = tokenF(u,i,p,v,j,d) \text{ if u}\neq\text{t and v}\neq\text{t;}$$

$$\Delta(tokenF(u,i,p,t,j,d)) = \\ = \exists n_1 \left(Futr(nFire(t,n_1),d) \wedge \left(\begin{array}{c} j \leq n_1 \wedge tokenF(u,i,p,t_1,j,d) \\ \wedge \\ j > n_1 \wedge tokenF(u,i,p,t_2,j-n_1,d) \end{array} \right) \right)$$

$$\Delta(tokenF(t,i,p,v,j,d)) =$$
$$= \exists n_1 \left(Futr(nFire(t,n_1),d) \wedge \left(\begin{array}{c} i \leq n_1 \wedge tokenF(t_1,i,p,v,j,d) \\ \wedge \\ i > n_1 \wedge tokenF(t_2,i-n_1,p,v,j,d) \end{array} \right) \right)$$

This translation states, arbitrarily but without loss of generality, that the firings of transition t1 correspond to the ones of t having a lowest index, while those of t2 correspond to those with highest index.

Iteration rule

The iteration rule can be proven correct by defining the Δ function on the tokenF predicate as follows.

$\Delta(tokenF(u,i,p,v,j,d)) = tokenF(\lambda^{-1}(u),i,p,\lambda^{-1}(v),j,d)$ if u$\neq$r$_{x,y}$ or v$\neq$t;

$$\Delta(tokenF(r_{x,y},i,p_x,t,j,d)) =$$
$$\exists h' \exists d' \exists h'' \exists d'' \left(\begin{array}{c} tokenF(r_{x,y},i,p_x,t_1,h',d') \wedge \\ Futre(tokenF(t_1,h',q_1,t_3,h'',d''),d') \\ Futr(tokenF(t_3,h'',q_5,t_5,j,d-d'-d''),d'+d'') \end{array} \right)$$

In words: a token produced in net S by the transition r$_{x,y}$ and consumed by t after d time units is corresponds, in net I, to three tokens, the first one produced by r$_{x,y}$ and consumed by t1 after d', the second one produced by the same firing of t1 and consumed by t3 after d'' time units and the third one produced by the same firing of t3 and consumed after d time units from the transition t5 firing.

Place sequencing rule

This rule, shown in table 6.2(b) is very similar to the transition sequencing rule of table 6.1(d). In fact, it could be proven correct using corollary 6.8 as in the above discussed cases, defining Δ as follows.

$\Delta(tokenF(u,i,p,v,j,d)) = tokenF(\lambda^{-1}(u),i,p,\lambda^{-1}(v),j,d)$ if u$\neq r_x$ and v$\neq t_y$;

$$\Delta(tokenF(r_x,i,p,t_y,j,d)) =$$
$$= \exists h \exists d' (tokenF(r_x,i,p,s,h,d') \wedge Futr(tokenF(s,h,q,t_y,j,d-d'),d'))$$

We do not develop further the proof of correctness for the place sequencing rule, however, because place sequencing rule can be seen as a particular case of process splitting rule, which will be treated in the next section.

6.5.1 An extension to the proof method

The method for proving correctness of refinement rules, based on Corollary 6.8 and applied to several cases in the preceding section 6.5, does not apply to the place splitting and process splitting rules.

We illustrate this fact by referring, without loss of generality, to the simplest nets of Figure 6.3, where we assume the time bounds of transition t to be m$_t$=1 and M$_t$=4. We consider, as shown in Figure 6.4(a), an execution of net S where transition r fires at times 0, 2, and 5, while transition t fires at times 4, 6, and 8. The lines connecting transition firings above the time axis show that the tokens produced by r's firing at times 0, 2, and 5 are consumed respectively at times 4, 6, and 8 by a firing of transition t (notice that such correspondences, when the firing times are fixed, are the only possible ones). Figure 6.4(b) shows an execution of net I where transitions r and t fire exactly at the same times as the homonymic transitions in S; the lines above (resp., below) the time axis describe the production and consumption of tokens passing through place *p1* (resp.,

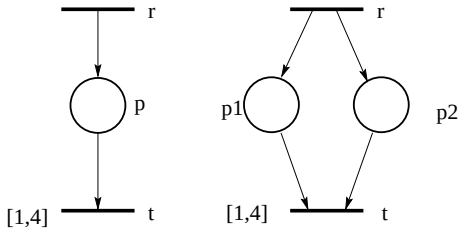


Figure 6.3: A simplest application of the place splitting refinement rule.

Δ	F	$\Delta(F)$
$\Delta 1$	*tokenF*(r, i, p, t, j, d)	*tokenF*(r, i, p1, t, j, d) $\wedge$ *tokenF*(r, i, p2, t, j, d)
$\Delta 2$	*tokenF*(r, i, p, t, j, d)	*tokenF*(r, i, p1, t, j, d)
$\Delta 3$	*tokenF*(r, i, p, t, j, d)	*tokenF*(r, i, p1, t, j, d) $\vee$ *tokenF*(r, i, p2, t, j, d)

Table 6.3: Proof functions for the place splitting rule

p2): for instance, the token introduced into *p1* by the firing of *r* at time 2 is consumed by the firing of *t* at time 8, while the token introduced by the same firing of *r* at time 2 into place *p2* is consumed by the firing of *t* at time 4.

First, we illustrate how the three simple (tentative) proof functions $\Delta 1$, $\Delta 2$, and $\Delta 3$ reported in Table 6.3 do not satisfy the requirement (ii) of Corollary 6.8 (i.e., not all axioms of Ax(S), translated through them, are theorems of net I). Let us consider the predicate *tokenF*(r, i, p, t, j, d) that relates firings of transitions *r* and *t*, with the token produced by *r*'s firing consumed *d* time units later by a firing of *t*.

Proof function $\Delta 1$, applied to axiom

UB_S(t):

$$fireth(r,i) \rightarrow \exists d(d \leq M_t \wedge \exists j\, tokenF(r,i,p,t,j,d))$$

returns the formula

$\Delta 1(\mathrm{UB}_S(\mathrm{t}))$:

$$fireth(r,i) \rightarrow \exists d(d \leq M_t \wedge \exists j\, (tokenF(r,i,p_1,t,j,d) \wedge tokenF(r,i,p_1,t,j,d)))$$

Which asserts that in net I two tokens introduced into places *p1* and *p2* by one given firing of transition *r* would always be consumed by the same firing of transition *t*. This is however false, as it is shown by all firings of transition *r* in Figure 6.4(b). Hence $\Delta 1$ is not adequate for proving correctness of the place splitting refinement rule because, in a sense, it imposes too strong a constraint on I's behavior.

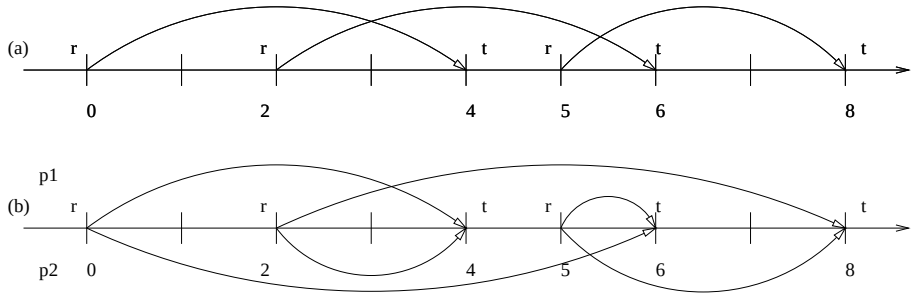


Figure 6.4: Token production and consumption in the two nets S (a) and I (b) of Figure 6.3.

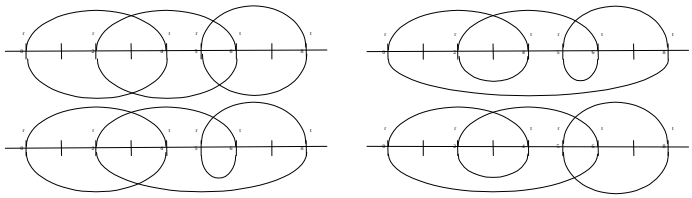


Figure 6.5: Some more executions of net I.

The proof function $\Delta 3$ suffers symmetrical problems, with axiom
OU_S(r): *tokenF*(r, i, p, t, j, d) $\wedge$ *tokenF*(r, i, p, t, k, e) $\rightarrow$ j=k $\wedge$ d=e
that, translated through $\Delta 3$, becomes
$\Delta 3(\mathrm{OU}_S(r))$:

$$\left(\begin{array}{c} (tokenF(r,i,p_1,t,j,d) \vee tokenF(r,i,p_2,t,j,d)) \\ \wedge \\ (tokenF(r,i,p_1,t,k,e) \vee tokenF(r,i,p_2,t,k,e)) \end{array} \right) \rightarrow j = k \wedge d = e$$

which implies

$$tokenF(r,i,p_1,t,j,d) \wedge tokenF(r,i,p_1,t,k,e) \rightarrow j = k \wedge d = e$$

again asserting that in net I two tokens introduced into places *p1* and *p2* by one given firing of transition *r* are necessarily consumed by the same firing of transition *t*.

As a final example, proof function $\Delta 2$ applied to UB_S(t) produces the following formula:

$$\Delta 2(UB_S(t)) = fireth(r,i) \rightarrow \exists d(d \leq M_t \wedge \exists j\, tokenF(r,i,p1,t,j,d))$$

asserting that any token introduced into p1 by a firing of transition *r* will always be consumed by a firing of transition *t* within *t*'s upperbound M_t, which is clearly false, as shown by *r*'s firing at time 2 in Figure 6.4(b).

We can understand why any attempt to prove correctness of the place splitting rule by using a proof function is bound to fail by considering once more the execution of net S depicted in Figure 6.4(a) and confronting it with that of net I in Figure 6.4(b). The use of a proof function Δ for proving correctness of a refinement rule is based on the idea that the translation of the *tokenF* predicate through Δ describes how the mechanism of token production and consumption by transitions of S is simulated in net I. Several applications of this idea were discussed in section 6.5. Applying this approach to the place splitting refinement rule amounts to trying to translate the production of a token by *r* and its consumption by *t* into the production and consumption of some tokens in net I. The example of net executions in Figure 6.4 shows, however, that this in not possible. In fact, in Figure 6.4(a) neither the tokens flowing through place *p1* nor those flowing through *p2* in net I behave like the tokens flowing through the unique place *p* in net S. (Notice that there are *other* executions of net I, some of which are shown in Figure 6.5, where either the tokens flowing through *p1* or through *p2* of net I or both follow the same pattern as those in place *p* of net S, but these are not the only possible executions.)

In conclusion, no proof function satisfying the hypotheses of Corollary 6.8 can be defined for the place splitting rule. A similar reasoning applies to the process splitting refinement rule, where the amount of nondeterminism introduced by the refinement step is even greater.

To overcome these difficulties we propose a proof method that extends the one defined by corollary 6.8 and adopted so far.

Theorem 6.9 *Let S, I, $\mathcal{S}$, $\mathcal{I}$, and Λ be defined as in Corollary 6.8. Assume that $\Delta 1$ and $\Delta 2$ are two proof functions from S to I such that*

(*) $\vdash_{\mathcal{I}} \Delta 1(Ax(S)) \vee \Delta 2(Ax(S))$.
Then net I implements net S.

Proof. Let φ be an observable property of net S, i.e., $\vdash_{\mathcal{S}} \varphi$. By meta-theorem 7, we have
(•) $\Delta 1(\mathrm{Ax(S)}) \vdash_{\mathcal{I}} \Delta 1(\varphi)$ and $\Delta 2(\mathrm{Ax(S)}) \vdash_{\mathcal{I}} \Delta 2(\varphi)$.
Let $\Delta 1(\mathrm{Ax(S)})=\phi_1, \ldots \phi_n$ and $\Delta 2(\mathrm{Ax(S)})= \psi_1, \ldots \psi_n$; then (•) can be rewritten as
$\phi_1, \ldots \phi_n \vdash_{\mathcal{I}} \Delta 1(\varphi)$ and $\psi_1, \ldots \psi_n \vdash_{\mathcal{I}} \Delta 2(\varphi)$.
Applying repeatedly the deduction theorem we get
$\vdash_{\mathcal{I}} \phi_1 \rightarrow \ldots \rightarrow \phi_n \rightarrow \Delta 1(\varphi)$ and $\vdash_{\mathcal{I}} \psi_1 \rightarrow \ldots \rightarrow \psi_n \rightarrow \Delta 2(\varphi)$
and, by the tautology $(\alpha \rightarrow \beta \rightarrow \Gamma) \rightarrow (\alpha \wedge \beta \rightarrow \Gamma)$,
$\vdash_{\mathcal{I}} \phi_1 \wedge \ldots \wedge \phi_n \rightarrow \Delta 1(\varphi)$ and $\vdash_{\mathcal{I}} \psi_1 \wedge \ldots \wedge \psi_n \rightarrow \Delta 2(\varphi)$.
Since $\Delta 1$ and $\Delta 2$ are extensions of Λ , it follows that
$\vdash_{\mathcal{I}} \phi_1 \wedge \ldots \wedge \phi_n \rightarrow \Lambda(\varphi)$ and $\vdash_{\mathcal{I}} \psi_1 \wedge \ldots \wedge \psi_n \rightarrow \Lambda(\varphi)$
from which, thanks to hypothesis (*) and Case Analysis we finally get the thesis, $\vdash_{\mathcal{I}} \Lambda(\varphi)$. □

Note Notice that using two proof functions $\Delta 1$ and $\Delta 2$ when applying Theorem 6.9 is *not* equivalent to using a single proof function Δ defined as $\Delta(\varphi) = \Delta 1(\varphi) \vee \Delta 2(\varphi)$. In fact, it can be easily verified that function Δ as just defined is not even a proof function, because $\Delta(\beta \wedge \gamma) \neq \Delta(\beta) \wedge \Delta(\gamma)$.

Theorem 6.9 can be immediately generalized to the case where, instead of two proof functions $\Delta 1$ and $\Delta 2$, a generic positive number n of proof functions $\Delta 1, \ldots \Delta n$ are used.

Corollary 6.10 *(to Theorem 6.9). In the same hypotheses as Theorem 6.9, assume that* n *proof functions* $\Delta 1, \ldots \Delta n$ *from* $\mathcal{S}$ *to* $\mathcal{I}$ *are given, such that*
$\vdash_{\mathcal{I}} \Delta 1(Ax(S)) \vee \Delta 2(Ax(S)) \vee \ldots \vee \Delta n(Ax(S))$.
Then net I implements net S.

The proof of Corollary 6.10 is a straightforward generalization of the proof of Theorem 6.9.

Theorem 6.9 and Corollary 6.10 can be used to prove the correctness of the place splitting and process splitting refinement rules. Again, for the sake of brevity we only present here the definition of the proof functions, and report the details of the proofs in [FGM95].

Place Splitting rule

The proof of correctness uses two proof functions $\Delta 1$ and $\Delta 1$. For transitions not in the preset nor in the postset of the refined place, i.e., for $u \neq r_h$, $\forall$ h $\in$ [1..n], or $v \neq t_k$, $\forall$ k $\in$ [1..m], they are defined as follows.

$$\Delta 1(tokenF(u,i,q,v,j,d)) = \Delta 2(tokenF(u,i,q,v,j,d)) = \\ = tokenF(\lambda^{-1}(u),i,q,\lambda^{-1}(v),j,d).$$

For transitions in the preset or postset of the refined place, $\Delta 1$ and $\Delta 2$ are reported below.
$\Delta 1(tokenF(r_h,i,p,t_k,j,d)) = tokenF(r_h,i,p_1,t_k^{'},j,d)$ $\forall$ h $\in$ [1..n], $\forall$ k $\in$ [1..m];
$\Delta 2(tokenF(r_h,i,p,t_k,j,d)) = tokenF(r_h,i,p_2,t_k^{'},j,d)$ $\forall$ h $\in$ [1..n], $\forall$ k $\in$ [1..m].

$\Delta 1$ and $\Delta 2$ are therefore equal except for the way they translate the production and consumption of tokens flowing through the refined place p: $\Delta 1$ asserts that in net I the token flows through place p_1, while $\Delta 2$ through place p_2.

Process Splitting rule

For the process splitting rule we can apply Corollary 6.10 with as many Δi's as there are branches in the refined net fragment. Assuming, as in Table 6.2(d), that this number is s, in the following we define Δ_h, for each h $\in$ [1..s], as follows. As in the preceding paragraph we treat differently the case of transitions in the preset and postset of the refined place.

$\Delta_h(tokenF(u,i,q,v,j,d)) = tokenF(\lambda^{-1}(u),i,q,\lambda^{-1}(v),j,d)$ for $u \neq r_k$, $\forall k \in [1..n]$, or $v \neq t_l$, $\forall l \in [1..m]$

$\Delta_h(tokenF(r,i,p,t_1,j,d)) =$

$\exists e \exists x(tokenF(r_k,i,p_h,v_h,x,e) \wedge Futr(tokenF(v_h,x,q_h,t_1,j,d-e),e))$ $\forall k \in$ [1..n], $\forall l \in$ [1..m].

Again the difference among the various Δ_h's is limited to the way they translate the production and consumption of tokens flowing through the refined place p: each different Δ_h, for h $\in$ [1..s], asserts that in net I the token traverses the branch with index h, flowing through places p_h and q_h.

6.6 Conclusions

The principal motivation for our research lies in the belief that formal methods and the related specification and verification techniques provide an adequate theoretical basis and a useful methodological support to the development of time critical systems. The use of formal specification and verification techniques can improve the reliability of time critical systems by permitting the production of unambiguous specification and by supporting the use of powerful tools to detect faults early in the development process.

In our view, one of the reasons why formal methods are still seldom applied in the industrial practice is that for most formalisms having high expressive power the verification procedures either cannot be performed mechanically or their computational cost is too high to permit their use in realistic projects. These problems can be addressed by adopting a development methodology based on successive refinements that reduces the overall design and verification effort by reusing, at each phase of the development, the results gathered in the preceding steps.

In the present chapter we discussed these issues referring to an operational formalism like time Petri nets used as abstract system models, and to the TRIO temporal logic used as a means to describe desired timing properties. We formally defined the notions of implementation and provided a general method to prove that a set of refinement rules are correct with respect to this notion of implementation. We also showed how the development of a system through a series of correct refinement steps greatly facilitates its verification by allowing the designer to prove with a reduced effort intermediate lemmas on the early versions of the system. The described refinement rules are now supported by Cabernet [PG92], a tool for the incremental specification and design of time critical systems based on time Petri nets developed at Politecnico di Milano.

We claim that the presented development method, with its related formal definitions and proofs, can be adapted, with suitable modifications, to other formalisms that combine a descriptive language for specifying timing requirements with an operational notation to model system structure, such as, for instance, timed transition systems [HMP91] or TTM/RTTL [Ost89].

As noted in the introduction, our logic-based characterization of the implementation relation differs from other works on refinement, where such definitions are based on behaviors and execution traces. It is therefore interesting and useful to provide an alternative, more *operational* view of our notion of implementation. The semantics of time Petri nets is often defined in terms of *observable time behaviors*: an observable time behavior (OTB for short) lists all the transition firings in a given execution of the net, each firing being associated with the time of its occurrence. According to this view, the firings are the only externally observable elements of the net, whose semantics is defined as the set of all possible OTBs. Then a natural definition of the implementation relation [FGP93] states that a net I implements net S iff the set of OTB of I is a *subset* of the set of OTB of S. It can be immediately noticed that this notion of refinement is equivalent to the one presented in this chapter, by considering that every OTB essentially identifies (i.e., is in one-to-one correspondence with) a model of the TRIO formulas that describe the net behavior. Therefore, requiring that the set of OTB (that is, of the models of the TRIO formulas) of net I be a subset of those of S (after a possible renaming of transitions) is equivalent to requiring that all formulas satisfied in the models of S be satisfied (again, after a possible renaming of the transitions) also in the models of I, as we do in the present work.

Yet another way of defining implementation among time Petri nets could be based on the notion (very common in the literature on TPNs) of *simulation*, where a relation among the states of two nets is introduced, such that if the two nets are in related states and one net executes a transition that takes it to another state, then the other, simulating net can execute the same transition (or a similar one) and enter a related state. Characterization of implementation among TPNs in terms of simulation is an interesting open research problem. Any work in this direction should take into account the fact that the notion of state for TPNs is more elaborate than for untimed Petri nets: as shown in [BD91], the state space of any nontrivial time Petri net has an uncountable infinite set of states, which can be grouped (under some easily-satisfied conditions) into a finite or denumerable set of state classes. It is to be expected that the definition of implementation in terms of simulation would require the specification net S and the implementation net I to have state graphs with some kind of structural relationship.

Chapter 7

Conclusions

In this thesis we have seen (in Chapter 2) formal methods for the specification and analysis of safety-critical real-time systems. In particular we have seen the graphical operational method time Petri Nets and then the descriptive logical language TRIO (in section 2.2). Still in that chapter we have presented the dual language approach, which combines the best of the two kinds of formalism.

Although these methods have already proven to be capable for application in real cases, this thesis suggests original extension and improvements with contributions both conceptual and theoretical, technical and methodological.

From the conceptual viewpoint, we have performed an in-depth analysis and formalization of notions that are often stated informally or assumed implicitly but are seldom adequately formalized, such as events, interval variables, non-Zenoness for several kind of entities, finite variability, cause-effect relations (in Chapter 3), zero-time and simultaneous transitions (in Chapter 4) and refinement, implementation, and temporal property preservation (Chapter 6). We believe that definitions and analysis of these concepts, although given in terms of TRIO and sometimes TPNs, can be applied to other methods as well.

In Chapter 3 we have provided 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.

In the framework of the dual language approach we have presented in Chapter 4 a new axiomatic system that allows dealing with zero-time transitions in a way that is both intuitive and general, formally (thanks to the non-standard analysis) defining notions like infinitesimal duration of a transition.

As an experimental counterpart to the conceptual contributions reported in Chapters 3 and 4, in Chapter 5 we implemented our framework as a tool suite inside a widely available, powerful theorem prover such as PVS. 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.

In Chapter 6 we have provided necessary definitions, useful theorems and applicable rules to deal with the refinement of specifications of real-time systems, in order to obtain an implementation starting from the most abstract specification.

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, when 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 [CENb, CENa].

Bibliography

[AGM97] Andrea Alborghetti, Angelo Gargantini, and Angelo Morzenti. Providing automated support to deductive analysis of time critical systems. In Mehdi Jazayeri and Helmut Schauer, editors, *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*, pages 211–226, Zurich, Switzerland, September 1997. Springer-Verlag.

[AH90] Rajeev Alur and Thomas A. Henzinger. Real-time logics: Complexity and expressiveness. In *Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401, Philadelphia, Pennsylvania, jun 1990. IEEE Computer Society Press.

[AH96] M. Archer and C. Heitmeyer. TAME: A specialized specification and verification system for timed automata. In Azer Bestavros, editor, *Work In Progress (WIP) Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, pages 3–6, Washington, DC, December 1996. The WIP Proceedings is available at urlhttp://www.cs.bu.edu/pub/ieee-rts/rtss96/wip/proceedings.

[AH97] Myla Archer and Constance Heitmeyer. Human-style theorem proving using PVS. In Elsa Gunter and Amy Felty, editors, *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs '97*, volume 1275 of *Lecture Notes in Computer Science*, pages 33–48, Murray Hill, NJ, August 1997. Springer-Verlag.

[Aiz89] Jacob I. Aizikowitz. Designing distributed services using refinement mappings. Technical Report TR89-1040, Cornell University, Computer Science Department, September 1989.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AL94] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 5(16):1543–1571, sep 1994.

[BC84] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In A. W. Roscoe S. D. Brookes and G. Winskel, editors, *Proceedings of the Seminar on Concurrency*, volume 197 of *LNCS*, pages 389–448, Pittsburgh, PA, July 1984. Springer.

[BCC+95] M. Basso, E. Ciapessoni, E. Crivelli, D. Mandrioli, A. Morzenti, E. Ratto, and P. San Pietro. Experimenting a logic-based approach to the specification and design of the control system of a pondage power plant. In *ICSE-17, Workshop on Industrial Application of Formal Methods*, Seattle, WA, 1995.

[BCC+98] M. Basso, E. Ciapessoni, E. Crivelli, D. Mandrioli, A. Morzenti, E. Ratto, and P. San Pietro. A logic-based approach to the specification and design of the control system of a pondage power plant. In C. Tully, editor, *Improving Software Practice: Case Experience*, Series in Software Based Systems, pages 79–96. Wiley, 1998.

[BD91] B. Berthomieu and M. Diaz. Modelling and verification of time dependent systems using Time Petri Nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, March 1991.

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, September 1991.

[Bry85] R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proceedings of the 22nd ACM/IEEE Design Automation Conference*, pages 688–694, Los Alamitos, Ca., USA, June 1985. IEEE Computer Society Press.

[CCPC+99] E. Ciapessoni, A. Coen-Porisini, E. Crivelli, D. Mandrioli, P. Mirandola, and A. Morzenti. From formal models to formally-based methods: an industrial experience. *ACM TOSEM - Transactions On Software Engineering and Methodologies*, 8(1):79–113, 1999.

[CCPMM99] Riccardo Capobianchi, Alberto Coen-Porisini, Dino Mandrioli, and Angelo Morzenti. A framework architecture for supervision and control systems. In *ACM Computing Surveys*. 1999.

[CENa] CENELEC (European Committe for Electrotechnical Standardization). *Railway applications - Safety related electronic systems for signalling*, ref. no. pren 50129 edition.

[CENb] CENELEC (European Committe for Electrotechnical Standardization). *Railway applications - Software for railway control and protection systems*, ref. no. pren 50128 edition.

[Cer93] A. Cerone. *A Net-based Approach for Specifying Real-Time Systems*. PhD thesis, Universita' degli Studi di Pisa, Dipartimento di Informatica, Pisa, Italy, 1993. TD-16/93.

[CH85] P. Caspi and N. Halbwachs. A functional model for describing and reasoning about time behaviour of computing systems. *Acta Informatica*, 22:595–627, 1985.

[CH97] Zhou Chaochen and Michael Reichhardt Hansen. Duration calculus: Logical foundations. *Formal Aspects of Computin*, 9(3):283 – 330, 1997.

[CHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[CJ74] Richard Courant and Fritz John. *Introduction to Calculus and Analysis*. Springer, 1974.

[Cla91] Edmund M. Clarke, Jr. Temporal logic model checking: Two techniques for avoiding the state explosion problem. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 1–1, Berlin, Germany, June 1991. Springer.

[CM96] A. Cau and B. Moszkowski. Using pvs for interval temporal logic proofs. part 1: The syntactic and semantic encoding. SERCentre Technical Monograph SERCentre Technical Monograph 14, De Montfort University, Leicester,UK, 1996.

[Coh66] Pual. J. Cohen. *Set Theory and the Continuum Hypothesis*. Benjamin, New York, 1966.

[CPGK97] Alberto Coen-Porisini, Carlo Ghezzi, and Richard A. Kemmerer. Specification of realtime systems using ASTRAL. Technical Report TRCS96-30, University of California, Santa Barbara. Computer Science., January 1997.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DD95] Francine Diener and Marc Diener, editors. *Non standard analysis in practice*. Universitext. Springer-Verlag, 1995. ISBN 3-540-60297-6.

[DDGJ89] W. Damm, G. Döhmen, V. Gerstner, and B. Josko. Modular verification of petri nets. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, pages 180–207, REX Workshop, Mook, May/June 1989.

[DKMS+94] L. K. Dillon, G. Kutty, P. M. Melliar-Smith, L. E. Moser, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3(2):131–165, April 1994.

[DS97] Bruno Dutertre and Victoria Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transactions on Software Engineering*, 23(5):267–278, May 1997.

[Dut96] Bruno Dutertre. Elements of mathematical analysis in PVS. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, volume 1125 of *Lecture Notes in Computer Science*, pages 141–156, Turku, Finland, August 1996. Springer-Verlag.

[EH86] E. Allen Emerson and Joseph Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33:151–178, 1986.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, Inc., Orlando, Florida, 1972.

[FGM95] M. Felder, A. Gargantini, and A. Morzenti. Theory of implementation and refinement in timed petri nets. Technical Report 95-044, Politecnico di Milano, Dipartimento di Elettronica e Informazione, 1995.

[FGP93] M. Felder, C. Ghezzi, and M. Pezzè. Analyzing refinements of state-based specifications: The case of TB nets. In Thomas Ostrand and Elaine Weyuker, editors, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 28–39, New York, NY, USA, June 1993. ACM Press.

[FM92] M. Felder and A. Morzenti. Validating real-time systems by executing logic specifications in trio. In *Proceedings of 14th International Conference on Software Engineering*, pages 199–211, 1992.

[FM94] Miguel Felder and Angelo Morzenti. Validating real-time systems by history-checking TRIO specifications. *ACM Transactions on Software Engineering and Methodology*, 3(4):308–339, October 1994.

[FMM91] M. Felder, D. Mandrioli, and A. Morzenti. Proving properties of real- time systems through logical specifications and petri net models. Technical Report 91-72, Politecnico di Milano Department of Electronic Engineering and Information Sciences, December 1991.

[FMM94] M. Felder, D. Mandrioli, and A. Morzenti. Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models. *IEEE Transactions on Software Engineering*, 20(2):127–141, February 1994.

[Gar95] Angelo Gargantini. A temporal logic-based approach to the implementation and refinement of timed petri nets. Master's thesis, Politecnico di Milano, Dipartimento di Elettronica, 1995.

[GH99] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC/FSE '99*, number 1687 in LNCS, Toulose, France, September 1999. Springer-Verlag.

[GLMZ96] Angelo Gargantini, Lilia Liberati, Angelo Morzenti, and Cristiano Zacchetti. Specifying, validating and testing a traffic management system in the TRIO environment. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 65, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

[GM96] A. Gargantini and A. Morzenti. TRIO specification of a steam boiler controller. *Lecture Notes in Computer Science*, 1165:218–232, 1996.

[GM99] Angelo Gargantini and Angelo Morzenti. Automated deductive analysis of time critical systems based on methodical formal specification. Technical Report 50, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1999.

[GMM90] Carlo Ghezzi, Dino Mandrioli, and Angelo Morzenti. TRIO: A logic language for executable specifications of real-time systems. *The Journal of Systems and Software*, 12(2):107–123, May 1990.

[GMM99] Angelo Gargantini, Dino Mandrioli, and Angelo Morzenti. Dealing with zero-time transitions in axiom systems. *INFCTRL: Information and Computation (formerly Information and Control)*, 150, 1999.

[Har87] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HJL93] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. 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.

[HJL96] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HKL+98] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), November 1998.

[HL96] Constance Heitmeyer and Nancy Lynch. Formal verification of real-time systems using timed automata. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*, chapter 5. Wiley, 1996.

[HLR92] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992.

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

[HMP91] Tom Henzinger, Zohar Manna, and Amir Pnueli. Temporal proof methodologies for real-time systems. In ACM, editor, *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages, January 21–23, 1991, Orlando, FL*, pages 353–366, New York, NY, USA, 1991. ACM Press.

[Hoo94] Jozef Hooman. Correctness of real time systems by construction. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 19–40, L'ubeck, Germany, September 1994. Springer-Verlag.

[HRS98] K. Hansen, A. Ravn, and V. Stavridou. From Safety Analysis to Software Requirements. *IEEE Transactions on Software Engineering*, 24(7):573–584, 1998.

[Jef95] Ralph D. Jeffords. An approach to encoding the TRIO logic in PVS. Technical report, Naval Research Laboratory, 1995.

[Koy89] Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Eindhoven University of Technology, Netherland, 1989.

[LA92] N. A. Lynch and H. Attiya. USING MAPPINGS TO improve timing PROPERTIES (replaces 412.d). Technical Memo MIT/LCS/TM-412, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1992.

[Lev95] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[Lue79] D. G. Luenberger. *Introduction to Dynamic Systems: Theory, Models, & Applications*. Wiley, 1979.

[MF76] P.M. Merlin and D.J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Transactions on Communication*, 24(9):1036–1043, September 1976.

[MMG92] A. Morzenti, D. Mandrioli, and C. Ghezzi. A Model Parametric Real-Time Logic. *ACM Transactions on Programming Languages and Systems*, 14(4):521–573, 1992.

[MMM95] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, November 1995.

[MMP+96] D. Mandrioli, A. Morzenti, M. Pezze', P. San Pietro, and S. Silva. A petri net and logic approach to the specification and verification of real-time systems. In C. Heitmeyer and D. Mandrioli, editors, *Formal Methods for Real-Time Computing*, volume 5 of *Trends in Software*, chapter 5. Wiley, 1996.

[MMT91] Michael Merritt, Francesmary Modugno, and Mark R. Tuttle. Time-constrained automata (extended abstract). In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 408–423, Amsterdam, The Netherlands, 26–29 August 1991. Springer-Verlag.

[MP83] Zohar Manna and Amir Pnueli. Verification of concurrent programs: A temporal proof system. In J. W. de Bakker and J. van Leeuwen, editors, *Foundations of Computer Science: Distributed Systems*, pages 163–255. Mathematisch Centrum, Amsterdam, 1983.

[MP94] Angelo Morzenti and Pierluigi San Pietro. Object-oriented logical specification of time-critical systems. *ACM Transactions on Software Engineering and Methodology*, 3(1):56–98, January 1994.

[MRK+97] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon. A graphical environment for the design of concurrent real-time systems. *ACM Transactions on Software Engineering and Methodology*, 6(1):31–79, January 1997.

[Mül85] K. Müller. Constructable Petri nets. *J. Inf. Process. Cybern. EIK*, 21:171–199, 1985.

[Nel77] E. Nelson. Internal set theory: a new approach to nonstandard analysis. *Bulletin American Mathematical Society*, 83(6):1165–1198, 1977.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[OSRSC98] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference*. Computer Science Laboratory, SRI International, Menlo Park, CA, September 1998.

[Ost89] J. S. Ostroff. *Temporal Logic for Real–Time Systems*. Advanced Software Development Series ed. J. Kramer. Research Studies Press - John Wiley, England, UK, 1989.

[Par92] D. L. Parnas. Tabular representation of relations. Technical Report 260, CRL, McMaster University, Canada, 1992.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer-Verlag LNCS 828, 1994.

[PG92] M. Pezzè and C. Ghezzi. Cabernet: A customizable environment for the secification and analysis of real-time systems. In *DECUS Europe Symposium*, 1992.

[PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

[Pra65] Dag Prawitz. *Natural Deduction: A Proof-Theoretic Study*, volume 3 of *Stockholm Studies in Philosophy*. Almqvist & Wiksell, Stockholm, 1965.

[Ram73] C. Ramchandani. *Performance Evaluation of Asynchronous Concurrent Systems by Timed Petri Nets*. PhD thesis, Massachusetts Institute of Technology, Cambridge, 1973.

[Rei85] Wolfgang Reisig. *Petri Nets (an Introduction)*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, Berlin-Heidelberg-New York, 1985.

[RMSM+96] Y. S. Ramakrishna, P. M. Melliar-Smith, L. E. Moser, L. K. Dillon, and G. Kutty. Interval logics and their decision procedures. part I: An interval logic. *Theoretical Computer Science*, 166(1–2):1–47, October 1996.

[Rob61] Abraham Robinson. Non-standard analysis. Number 64 in A, pages 432–440. Proc. Roy. Acad. Amsterdam, 1961.

[Rob96] Abraham Robinson. *Non-Standard Analysis*. Princeton University Press, 1996.

[ROS98] John Rushby, Sam Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[RR88] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, June 1988.

[Rus97] John Rushby. Subtypes for specifications. In Mehdi Jazayeri and Helmut Schauer, editors, *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*, pages 4–19, Zurich, Switzerland, September 1997. Springer-Verlag.

[Sha93] Natarajan Shankar. Verification of real-time systems using PVS. In Costas Courcoubetis, editor, *Computer-Aided Verification, CAV '93*, volume 697 of *Lecture Notes in Computer Science*, pages 280–291, Elounda, Greece, June/July 1993. Springer-Verlag.

[SM83] Ichiro Suzuki and Tadao Murata. A method for stepwise refinement and abstraction of Petri nets. *Journal of Computer and System Sciences*, 27(1):51–76, August 1983.

[SS94] Jens U. Skakkebæk and N. Shankar. Towards a Duration Calculus proof assistant in PVS. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 660–679, Lübeck, Germany, September 1994. Springer-Verlag.

[SS96] S.Schöf and M. Sampels. Fairness and instant reactions in distributed simulation. In *Proc. of the 8th Europen Simulation Symposium (ESS'96)*, volume 1, pages 96–100, Genoa, Italy, 1996. Society for Computer Simulation International.

[vGG90] R. J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness,* Mook, The Netherlands, May/June 1989, volume 430 of *Lecture Notes in Computer Science*, pages 267–300. Springer-Verlag, 1990.

[Vog86] W. Vogler. Behaviour preserving refinements of petri nets. In Gottfried Tinhofer and Gunther Schmidt, editors, *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 246 of *LNCS*, pages 82–93, Bernried, FRG, June 1986. Springer.

[Vog89] Walter Vogler. Failures semantics and deadlocking of modular Petri nets. *Acta Informatica*, 26(4):333–348, February 1989.

[Vog92] Walter Vogler. *Modular construction and partial order semantics of Petri nets*, volume 625 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1992.

[WOLB92] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. McGraw-Hill, New York, second edition, 1992.

[YMW97] Jin Yang, Aloysius K. Mok, and Farn Wang. Symbolic model checking for event-driven real-time systems. *ACM Transactions on Programming Languages and Systems*, 19(2):386–412, March 1997.

[YY91] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. Technical Report SERC-TR-81-P, Software Engineering Research Centre, March 1991.

[ZJ97] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.