Real-Time Systems: A Survey of Approaches to Formal Specification and Verification*

Carlo Ghezzi, Miguel Felder, Carlo Bellettini

Politecnico di Milano, Dipartimento di Elettronica e Informazione
P.za L. da Vinci 32, 20133 Milan, Italy

Abstract. This paper reviews past work done by our group in the area of formal specification for reactive, real-time systems. Different approaches are discussed, emphasizing their ability to verify formal specifications and systematically derive test cases for the implementation. The specification languages reviewed here are TB nets (a specification formalism belonging to the class of high-level Petri nets) and TRIO (a real-time temporal logic language).
Keywords and phrases Real-time systems, formal specification, requirement capture, Petri nets, high-level Petri nets, real-time temporal logic, analysis, testing, test-case generation.

1 Introduction

Real-time computer systems are increasingly used in the practical world. Moreover, they often constitute the kernel part of critical applications — such as aircraft avionics, nuclear power plant control, and patient monitoring — where the effect of failures can have serious effects or even unacceptable costs. These systems are generally characterized by complex interactions with the environment in which they operate and strict timing constraints to be met. They are real-time, since their behavior and their correctness depend on time: the effect of producing certain results too early or too late, with respect to the expected response time, may result in an error.

Existing semi-formal methods supporting specification, design, verification and validation of real-time systems ([16, 30, 15]) provide very limited support to high-quality software in the above domains. First, they often address only one or a limited set of the phases of the application development. Second, their semantics is informally defined, and therefore they provide no or partial support to analysis and execution. Most existing formal methods, on the other hand, are difficult to use, often lack facilities for handling real-time and for structuring large specifications, and provide limited tool support.

In the past, our group has been working in the area of formal specifications for reactive, real-time systems, with the goal of:

* This material is based upon work supported by the Esprit project IPTES, and by the Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo (CNR).

- **Understanding different specification paradigms.** Our belief is that "the" correct approach to the problem does not exist. Rather, we will eventually need to integrate different approaches in a specification support environment. In our work, we pursued research in two complementary directions: specifications based on an operational approach (namely, Petri nets) and specifications based on a descriptive approach (namely, real-time temporal logic).
- **Supporting verification of formal specification.** Requirements capture for a new application is a highly critical activity, which can have a far-reaching impact on the quality of the product. Requirements errors are often discovered very late, when the system has been delivered to the final user and is operational in the target environment. At this point, however, not only the cost of (part of) the development effort would be wasted, but also the cost of failures would bear on the cost of the application. Our goal has therefore been to ensure that specifications are verified before proceeding to implementation, so that errors are not inadvertently transferred from requirement specification down through the whole development cycle. In this paper, we use the term "verification" as an umbrella concept that captures all forms of assessment of a specification[2]. In particular, we will discuss two complementary forms of verification: static verification (which includes a whole range of possibilities, from static semantic checking to all forms of mathematical proofs) and dynamic verification (i.e., testing). We also discuss symbolic execution, which is somehow in the middle between the two approaches.
- **Supporting the specification activity through an integrated set of tools.** Formal methods are intrinsically more supportive of mechanical manipulation than informal methods. Tools may in fact be based on both the syntax (e.g., syntax-directed editing tools) and the semantics (e.g., semantic checkers) of the formalism. Tool support is essential to promote the use of formal methods among users.
- **Enhancing usability of the formalism.** Formal methods are very often based on languages that non-mathematicians find difficult to read and write. The syntax of the language is often awkward; no graphical description facilities are provided; no modularization and abstraction mechanisms are available to structure large specifications; and no application-specific concepts can be added to the language. Tool support is a first step towards improving usability, but more is needed. In the work done by our group, a layered approach was followed. The start point is a concise, clean, and mathematically defined notation. Other linguistic layers were then defined on top of the kernel in order to provide more expressive, user-oriented notations. Structuring mechanisms were also provided to allow specifications to be modularized according to the principles of abstraction and information hiding. In the case of the Petri net approach, we also defined a way to

[2] Note that other authors distinguish between verification and validation. Others distinguish between verification, intended as formal verification, and testing.

make the specification notation extensible, by providing a definitional device (based on graph grammars) through which new graphical specification notations may be added.
- **Supporting subsequent development steps (design, implementation, testing, etc.).** There are approaches where the formal specification is transformed into an implementation through predefined and partially automated transformation steps. In the work done by our group, this aspect has not been investigated so far. Work has been done in the derivation of test cases from the specification; such test cases can be used to verify an implementation.

This paper provides a comprehensive view of the work done by our group in the two aforementioned research directions. The discussion is mainly based on a survey of previously published work; most notably, [10, 11, 12, 14] for the Petri net based approach, and [13, 7, 21, 24] for the real-time logic based approach.

The paper is structured as follows. Section 2 surveys TB nets and their verification methods. TB nets are a class of high-level Petri nets. Their definition is provided in Section 2.1. Section 2.2 deals with CABERNET, an environment designed to support net-based specification and verification of real-time systems. Section 2.3 focuses on the verification facilities provided by CABERNET to support timing analysis, while Section 2.4 focuses on structuring mechanisms. Section 3 surveys the real-time logic language TRIO, its verification, and its support to implementation verification. In particular, the language is presented in Section 3.1. TRIO's formal semantics is discussed in Section 3.2. Section 3.3 deals with different kinds of TRIO verification and shows how the specification can provide support for implementation verification. Section 3.4 discusses how TRIO can support formal verification. Finally, Section 4 draws some conclusions and outlines future work. Section 3.5 outlines the extensions proposed to include structuring mechanisms in a real-time logic.

2 TB Nets: an operational specification language

Time Basic nets (TB nets) ([11]) are an extension of Petri nets ([31]). TB nets have been introduced in [11]. In this paper, we introduce them rather informally by using a slightly different notation than in [11].

2.1 The language

In TB nets, each token is associated with a *timestamp*, representing the time at which the token has been created by a firing. Each transition is associated with a *time-function*, which describes the relation between the timestamps of the tokens removed by a firing and the timestamps of the tokens produced by the firing.

Definition 1 (TB nets). A TB net is a 6-tuple $< P, T, \Theta; F, tf, m_0 >$ where

1. P, T, and F are, respectively, the sets of places, transitions, and arcs of a net. Given a transition t, the preset of t, i.e., the set of places connected with t by an arc entering t, is denoted by $^\bullet t$; the postset of t, i.e., the set of places connected with t by an arc exiting t, is denoted by $t^\bullet$.
2. Θ is a numeric set, whose elements are the timestamps that can be associated with the tokens. The timestamp of a token represents the time at which it has been created. For instance, Θ can be the set of natural numbers, or the set of non-negative real numbers. In the following, we assume $\Theta = \mathbb{R}^+$ (the set of non-negative real numbers; i.e., time is assumed to be continuous).
3. tf is a function that associates a function tf_t (called time-function) with each transition t. Let en denote a tuple of tokens, one for each place in the preset of transition t. Function tf_t associates with each tuple en a set of values θ ($\theta \subseteq \Theta$), such that each value in θ is not less than the maximum of the timestamps associated with the tokens belonging to tuple en. $\theta = tf_t(en)$ represents the set of possible times at which transition t can fire, if enabled by tuple en. When transition t fires, the firing time of t under tuple en is arbitraryly chosen among the set of values θ. The chosen firing time is the value of the timestamps of *all* the produced tokens.
4. m_0, the *initial marking*, is a function associating a (finite) multiset of tokens with each place. In general, we use function m to denote a generic marking of nets, i.e., $m(p)$ denotes the multiset of tokens associated with place p by marking m.

The set T is partitioned into two sets ST and WT, the set of *Strong Transitions* and *Weak Transitions*, respectively. If a transition belongs to ST, then, if it is enabled, it *must* fire within its maximum firing time, as defined by function tf, unless it is disabled by some other firing. Instead, if a transition in WT fires, it fires if it is enabled and before its maximum firing time has expired (i.e., a weak transition *can* fire, but it is not forced to fire). Actually the initial marking m_0 of a TB net must ensure that there exists no enabling $< en, t >$, with $t \in TS$, such that the maximum firing time of t under tuple en is less than the maximum of the timestamps associated with the tokens in m_0.

In [11] a deeper discussion is presented where the two kinds of transitions lead to the definition of different semantics for the TB nets. The first class is there referred to as *Strong Time Semantics* while the other is referred to as *Weak Time Semantics*. In [11] it is shown how Weak Time Semantics is closer to the original semantics of Petri nets, while the other is closer to the intuitive notion of time evolution.

In order to define the rule by which new markings of the net may be generated, starting from the initial marking m_0, we need to define the concepts of *enabling tuple*, *enabling*, *firing time*, and *enabling time*.

Definition 2 (Enabling tuple, enabling, firing time, enabling time). Given a transition t and a marking m, let en be a tuple of tokens, one for each input place of transition t, the transition t can fire at a time instant τ under tuple en if and only if :

1. $\tau \in tf_t(en)$;
2. τ is greater than or equal to the time associated with any other token in the marking;
3. τ is not greater than the maximum firing time of every other strong transition enabled in the marking

If $tf_t(en)$ is empty, there exists no time instant at which transition t can fire under tuple en; i.e., transition t is not enabled under tuple en. If $tf_t(en)$ is not empty, en is said to be an *enabling tuple* for transition t and the pair $x =< en, t >$ is said to be an *enabling*. The triple $y =< en, t, \tau >$ where $< en, t >$ is an enabling, $\tau \in tf_t(en)$, and t can fire at instant τ under tuple en is said to be a *firing*. τ is said to be the *firing time*. We refer to the maximum among the timestamps associated with tuple en as the *enabling time* of the enabling $< en, t >$.

The dynamic evolution of the net (its semantics) is defined by means of firing occurrences, which ultimately produce firing sequences.

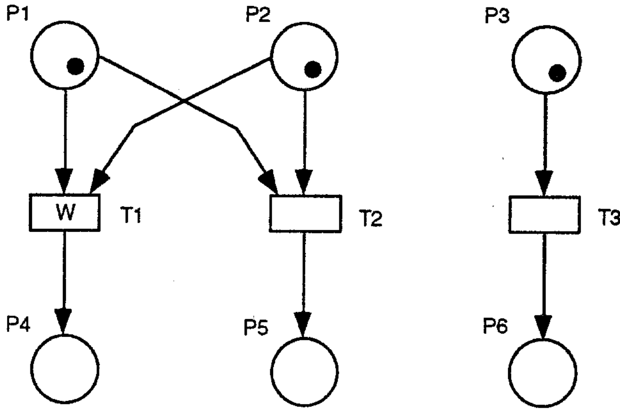
Definition 3 (Occurrence of a firing in a marking). Given a marking m and a firing $y =< en, t, \tau >$ such that en is contained in m, and t can fire at instant τ under tuple en, the firing occurrence of y in m produces a new marking m', that can be obtained from m by removing the tokens of the enabling tuple en from the places of $^\bullet t$, and producing a new token with timestamp τ in each of the places of $t^\bullet$. If x is a firing that produces marking m' from m we write $m[x > m'$.

Figure 1 shows a fragment of a TB net. Places $P2$ and $P3$ are marked with a token whose timestamp is 0. The timestamp associated to the token in place $P1$ is 1. Transitions $T2$ and $T3$ are strong; transition $T1$ is weak.

Function tf_{T1} states that transition $T1$ can fire at a time between the maximum time of the timestamps of the tokens in the place $P1$ and $P2$, and 5 time units after the value of the timestamp of token in $P2$. Transition $T1$ is weak and so it is possible that it does not fire also if the transition is enabled. In the example, this transition is enabled and the possible firing time is any value in [1,5].

Function tf_{T2} states that transition $T2$ can fire at a time between 8 time units after the maximum time of the timestamps of the tokens in the place $P1$ and $P2$, and 10 time units after the value of the timestamp of token in $P2$. In the example, this transition is enabled and the possible firing time is any value in [9,10].

Function tf_{T3} states that transition $T3$ can fire at a time between 3 and 15 time units after the value of the timestamp of token in $P3$. In the example, this transition is enabled and the possible firing time is any value in [3,10]. In fact, it cannot fire at a time greater than 10 unless transition T2 fires before (within 10).



$tf_{T1}(P1, P2) = \{\tau \mid max(P1, P2) \leq \tau \leq P2 + 5\}$
$tf_{T2}(P1, P2) = \{\tau \mid max(P1, P2) + 8 \leq \tau \leq P2 + 10\}$
$tf_{T3}(P3) = \{\tau \mid P3 + 3 \leq \tau \leq P3 + 15\}$
$m(P1) = \{1\};\ m(P2) = \{0\};\ m(P3) = \{0\}$

Fig. 1. A Simple TB net.

2.2 An introduction to the CABERNET environment

CABERNET ([29]) is an environment designed to support specification and verification of real-time systems based on high-level Petri nets[3].

CABERNET provides editing facilities to describe nets and execution facilities to animate them. By executing the net, it is possible to test a specification to detect specification errors. CABERNET provides different facilities to control execution, such as:

Execution mode: It is possible to execute a net with respect to the temporal constraints or as a pure net; i.e., ignoring timing information attached to both tokens and transitions.

Step mode: The user can direct the interpreter to proceed in single step mode, i.e., the interpreter waits for a command after each firing or each phase of a firing (e.g., identification and choice of the enabling, evaluation of the action, etc.).

Enabling choice: If required the user can select the enabling to fire. In the other case this is chosen nondeterministically.

Firing time choice: Once an enabling is chosen for firing, the firing time can be chosen according to different options:

---

[3] The acronym stands for Computer-Aided software engineering environment Based on ER NETs. ER nets are the kernel formalism of the environment. ER nets are Petri nets augmented with features to describe data, functionality, and control ([11]). TB nets are a timed formalism which is defined on top of ER nets.

Random: The firing time is chosen randomly in the set of possible firing times of the enabling.

Lowest/Highest: The lowest (respectively, highest) firing time for the enabling is chosen.

User choice: The user is prompted to choose one among the possible firing times.

CABERNET provides facilities for analyzing nets. In particular, it supports certain kind of verification of timing requirements, as we will show in Section 2.3. Finally, one of the goals of CABERNET is to support customization of the specification notation by means of a tool, called Meta-editor. The Meta-editor allows new graphical notations to be added to the environment, by defining not only their syntax, but also their semantics in terms of translation to the underlying kernel notation (ER nets).

In this way, the specification environment is open and extensible. User-oriented and application-oriented interfaces can be formally defined, and specifications written in the newly defined notations can be formally manipulated. We developed examples of this approach, such as the definition of Statecharts in terms of ER nets. The definition of the mapping between the external graphical language and the kernel notation is formally specified by means of graph grammars ([27]).

2.3 Verification of TB nets

The importance of executing formal specifications to validate requirements has been advocated, among others, by [19]. By executing formal specifications and observing the behavior of the specified system, one can check whether specifications capture the intended functional requirements or not. In other words, by executing requirements, we perform testing in the early phase of the development process. Although testing cannot prove the absence of errors, it is especially valuable as a mechanism for validating functional requirements.

The previous description of CABERNET shows how the available execution facilities provide some basic support to specification testing. In this section, we discuss two other forms of verification provided by CABERNET, which support verification of timing properties.

Symbolic execution. Symbolic execution is a well-known technique for analyzing sequential programs. It can be applied with different goals, such as verifying correctness of a particular path for all the input data which cause the execution of a particular path, synthesizing test data for a path.

In [4], symbolic execution is extended to deal with concurrent programs. In [12] symbolic execution is proposed as a method for analyzing a subset of TB nets, while [23] applies this proposal for validation of concurrent ADA^{TM} programs. Let us now briefly introduce the mechanism for symbolically execute a TB net.

Let us suppose that the tokens of the initial marking contain symbolic values for the timestamps. Let C (*Constraint*) be a boolean expression initialized by the user to describe the initial constraint on the timestamps. C is used to record the assumptions made on the timestamps and therefore plays the role of the Path Condition in the sequential case. Let ES (*Execution Sequence*) be a data structure recording a firing sequence. The pair $< C, ES >$ fully characterizes a symbolic execution. Below we describe a symbolic execution algorithm using $< C, ES >$.

At each transition firing, the symbolic interpreter incrementally update C, ES and the current symbolic marking m. The whole execution is based on the symbolic initial marking; that is, every symbolic value of the token timestamp is an expression derived from a sequence of elaborations starting from the symbolic values of the initial marking. The symbolic execution algorithm is decomposed in six steps. The algorithm is described below with reference to the example of Figure 1:

Step 0 : Initialization

The initial marking m_0 is defined by providing symbolic values for the timestamps of the tokens initially stored in the places, and by providing an initial constraint (C) on such timestamps. ES is initialized to NIL.

In the example, we can assume a symbolic marking in which places $P2$ and $P3$ contain a token whose associated timestamp τ_0 can assume any value between 0 and 10, and place $P1$ contains a token whose associated timestamp τ_1 can assume any value greater than τ_0 and less than $\tau_0 + 15$. Thus, the initial constraint is:

$C_0 = 0 \leq \tau_0 \leq 10 \ \wedge \ \tau_1 \geq \tau_0 \ \wedge \ \tau_1 \leq \tau_0 + 15$

Step 1 : Identification of the set of enabled transitions in the current marking

For each transition t potentially enabled by a tuple en (i.e., there is at least a token in each place of its preset), evaluate if there is some time value $\tau_{new} \in tf_t(en)$ that satisfies C and τ_{new} is greater than or equal to the last symbolic firing time and is less than the maximum firing time of any other enabled strong transition.

In the example the three transitions are all enabled. For instance, $T3$ is enabled because the expression that results by the conjunction of the following inequalities is satisfiable :

- C_0 (i.e., the previous constraint)
- $\tau_0 + 3 \leq \tau_{new} \leq \tau_0 + 15$ (i.e., $\tau_{new} \in tf_{T2}(en)$)
- $\tau_{new} \geq \tau_1$ (i.e., there is no other timestamp in the net that is greater than τ_{new})
- $\tau_{new} \leq \tau_0 + 10 \ \vee \ \tau_0 + 10 < \tau_1 \ \vee \ \tau_1 + 8 > \tau_0 + 10$ (i.e., either the firing is less than the maximum firing time of the other strong transition, or the other strong transition is not enabled.)

Step 2 : Selection of the enabling to fire

An *enabling* $< t, enab >$ of those found at the step 1 can be selected nondeterministically or according to the user's interaction.

Step 3 : Update of C

C is updated with the (possibly simplified) constraint built in the step 1 for the chosen enabling. If there is only one possible symbolic value of τ_{new} that satisfies the constraint, it is possible to substitute the variable τ_{new} with the symbolic expression. In most of cases, it is necessary to build a new symbolic value for the new variable.

In the example, if transition $T3$ is chosen, the new C will be:

$0 \leq \tau_0 \leq 10 \ \wedge \ \tau_0 \leq \tau_1 \leq \tau_0 + 15 \ \wedge \ \tau_0 + 3 \leq \tau_2 \leq \tau_0 + 15 \ \wedge$
$\tau_2 \geq \tau_1 \ \wedge \ (\tau_2 \leq \tau_0 + 10 \ \vee \ \tau_0 + 2 < \tau_1)$

Step 4 : Transition firing and marking update

Transition t of the enabling selected at step 2 is fired, i.e., the enabling tuple *enab* is removed from the places of $^\bullet t$ and new tokens are inserted into places of $t^\bullet$, bound to a symbolic expression that represents the possible firing times, or to a new symbolic value, as shown above.

In the example, the new marking will consists of a token (τ_0) in the place $P1$, a token (τ_1) in the place $P2$ and a token (τ_2) in the place $P6$.

Step 5 : Update of ES

Finally, ES should also be updated:

$ES_{new} := append(< t, enab >, ES_{old})$

Timing analysis. Reachability analysis is traditionally understood as finite enumeration of reachable states of some finite state model ([33]). It has been extended, however, to cope with infinite state models. This is the case of conventional Petri nets that can have an unbounded number of tokens in some places. In the case of infinite state models, analysis procedures have been derived, where states are grouped in (possibly infinite) sets and reachability analysis is then applied to such sets.

The techniques used for reachability analysis of (unbounded) Petri nets cannot be applied to the analysis of TB nets, since they group together the markings that differ only in the number of tokens in the marked places. In the case of TB nets, markings differ also in the timestamps associated with the tokens. For example, the number of states (markings) reachable in the TB net of Figure 1 by the firing of transition $T3$ is infinite, because the timestamp associated to the token in place $P6$ can be any real value in the interval $[3, 10]$.

[14] presents a technique for reachability analysis of TB nets, where each reachable state is symbolic, and represents a (possibly infinite) set of states of the TB net. That is, the technique is based on the symbolic execution presented in Section 2.3. In the general case, the analysis produces an infinite tree. However, [14] shows that the number of tree nodes to be examined for proving an interesting set of temporal properties is finite. Such properties may be classified in two sets: *bounded invariance* and *bounded response*.

A *bounded invariance* (or *timed safety*) property states an invariant property that must hold until a certain lower bound for time is reached. A *bounded response* (or *timed liveness*) property specifies that a certain property eventually holds, before an upper bound for time is reached.

The technique builds a *symbolic time reachability tree* (TRT). A node of the tree represents a symbolic state. A symbolic state is composed by the constraint C (specified as before) and a symbolic marking. A symbolic marking is a function μ from places to multisets of symbolic values. The symbolic values represent sets of numeric values for the timestamps associated with tokens in the marked places. An arc of the tree represents the symbolic firing of a transition. Starting from the root of the tree, which represents the initial symbolic marking, the tree is built by using rules similar to those of symbolic execution.

The TRT of the net in Figure 1 is shown in Figure 2. Nodes are graphically represented by two symbols: circles and squares. A circle indicates that it is possible that no transition is enabled in that symbolic state. For example, if the value of the token in $P1$ is greater than $\tau_0 + 10$ in the state $S3$ there is no transition enabled. The arcs are labeled with the name of the respective firing transition. Any set of values satisfying the constraint C associated to a node represents a feasible firing time schedule of the net up to the transition leading to such a node.

In the CABERNET environment, the procedure sketched above is implemented by a tool for specification analysis called MERLOT ([1]). MERLOT allows the user to prove bounded response and bounded invariance properties of the specified system. More specifically, the tool allows one to prove the following properties:

Occurrence of events (firings) or of special states (markings). Properties can assert either that there exists at least one execution of the system where some states or events are reached or that all possible executions reach these events or states within a given time limit. Example states in the example are those characterized by a token in the place $P1$ and a token in the place $P5$.

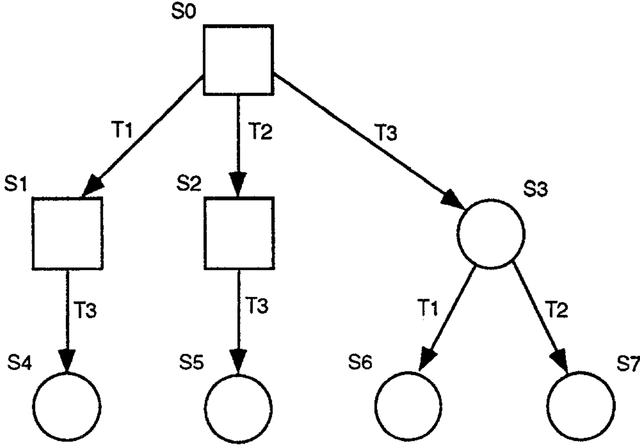
Sequence of events and states. Properties can assert some precedence constraints between different states and events. For example, that the firing of $T3$ is followed by a firing of $T1$.

Time of occurrence of events. Properties can assert some constraints on the firing time. For example that the firing time of the transition $T3$ is greater of the firing time of $T2$ plus a constant.

Any logical combination of the previous properties can also be expressed as a property to be proved.

2.4 Abstractions and hierarchies

Net-based specifications suffer from the lack of adequate structuring mechanisms. In practice, specifications may become hard to read and understand, and verification procedures may become very inefficient, as the size of the net reaches certain bounds. In particular, reachability analysis suffers from the state explosion problem.



S0: $C_0 := \tau_0 \geq 0 \ \wedge \ \tau_0 \leq 10 \ \wedge \ \tau_1 \geq \tau_0 \ \wedge \ \tau_1 \leq \tau_0 + 15$
Marking:= $\mu(P3) = \{\tau_0\}$; $\mu(P2) = \{\tau_0\}$; $\mu(P1) = \{\tau_1\}$
S1: $C_1 := C_0 \ \wedge \ \tau_2 \leq \tau_0 + 5 \ \wedge \ \tau_2 \geq \tau_0 \ \wedge \ \tau_2 \geq \tau_1$
Marking:= $\mu(P4) = \{\tau_2\}$; $\mu(P3) = \{\tau_0\}$
S2: $C_2 := C_0 \ \wedge \ \tau_3 \geq \tau_1 + 8 \ \wedge \ \tau_3 \leq \tau_0 + 10$
Marking:= $\mu(P5) = \{\tau_3\}$; $\mu(P3) = \{\tau_0\}$
S3: $C_3 := C_0 \ \wedge \ \tau_4 \leq \tau_0 + 15 \ \wedge \ \tau_4 \geq \tau_1 \ \wedge \ \tau_4 \geq \tau_0 + 3 \ \wedge \ (\tau_1 > \tau_0 + 2 \ \vee \tau_4 \leq \tau_0 + 10)$
Marking:= $\mu(P6) = \{\tau_4\}$; $\mu(P2) = \{\tau_0\}$; $\mu(P1) = \{\tau_1\}$
S4: $C_4 := C_1 \ \wedge \ \tau_5 \geq \tau_0 + 3 \ \wedge \ \tau_5 \leq \tau_0 + 15 \ \wedge \ \tau_5 \geq \tau_2$
Marking:= $\mu(P6) = \{\tau_5\}$; $\mu(P4) = \{\tau_2\}$
S5: $C_5 := C_2 \ \wedge \ \tau_6 \geq \tau_0 + 3 \ \wedge \ \tau_6 \leq \tau_0 + 15 \ \wedge \ \tau_6 \geq \tau_3$
Marking:= $\mu(P6) = \{\tau_6\}$; $\mu(P5) = \{\tau_3\}$
S6: $C_6 := C_3 \ \wedge \ \tau_7 \leq \tau_0 + 5 \ \wedge \ \tau_7 \geq \tau_4$
Marking:= $\mu(P6) = \{\tau_4\}$; $\mu(P4) = \{\tau_7\}$
S7: $C_7 := C_3 \ \wedge \ \tau_8 \geq \tau_1 + 8 \ \wedge \ \tau_8 \leq \tau_0 + 10 \ \wedge \ \tau_8 \geq \tau_4$
Marking:= $\mu(P6) = \{\tau_4\}$; $\mu(P5) = \{\tau_8\}$

Fig. 2. The TRT of the TB net of Figure 1.

We addressed these issues by allowing nets to be defined in a hierarchical, top-down manner, where one specification level implements a more abstract level. We defined what it means that a TB net I is a correct implementation of a TB net S. Intuitively, I is an implementation of S if I adds details (i.e., transitions, arcs, and places) and possibly restricts the set of behaviors that are possible at the S level. If I is a correct implementation, then if certain properties have been proved to hold for the specification, they also hold for the implementation. This result is important since it allows properties to be proved for a net of limited size, and then extended to a net of larger size, under certain assumptions on the relationship between the two nets. Furthermore, a number of constructive rules have been defined through which a specification may be refined into a

correct implementation. If these rules are followed, it is not necessary to prove the correctness of the implementation relation a-posteriori, since it is guaranteed to hold a-priori. These results are formally presented in [5].

As we mentioned, large net-based specifications are quite difficult to understand. Hierarchical definition helps, but does not solve the problem. Although nets are a graphical formalism, specifications are written according to mathematical abstractions, not using user and application-oriented concepts. Moreover, the specification language is defined once for all; it is not tailorable or adaptable to the needs of the specifier. On the other hand, it would be useful to provide a specification formalism that can be customized to match the specific needs of a specifier. It would also be useful to provide a notation that uses graphical concepts that are familiar to the user, so that requirements verification can be done more effectively. For example, in a control system for a hydraulic plant, user-oriented graphical abstractions may include valves that can be open or close, pipes, etc. This issue has considered by CABERNET, which allows a Meta-user to formally define new language layers on top of the underlying kernel formalism, using the Meta-editor mentioned in Section 2.2.

3 TRIO: a descriptive specification language

TRIO is a first-order temporal logic language for executable specification of real-time systems. The language deals with time in a quantitative way by providing a metric to indicate distance in time between events and length of time intervals.

A major goal of TRIO is *executability of specifications*. This means that TRIO formulas can be automatically checked for satisfiability or validity ([7]). When a formula specifying a given property of a system is interpreted, a *model* thereof is generated. Of course, since TRIO contains first-order theories, executability is undecidable in the general case. However, an analysis procedure on finite domains can be performed algorithmically by using the Tableaux Method ([32]), which provides an abstract interpreter of the language. The tableaux method is a widely used technique in temporal logic to constructively verify the satisfiability of a formula and to derive implementations from models of specification formulas ([22]). Although TRIO specifications are often stated by assuming an underlying infinite structure, such analysis may increase the confidence in the correctness of the specifications in much the same way as testing a program may increase the confidence in its reliability. That is, by examining the system behavior on finite domains, the user may infer the behavior on infinite domains. Such a generalization, however, cannot be proven, and can only be performed under the user's responsibility.

3.1 The TRIO language

The purpose of the following brief presentation of the TRIO language is to make the paper self-contained, *not* to provide a complete discussion of its features and its practical use. A complete description can be found in [25], [13], and [24].

TRIO is a first-order logic language, augmented with temporal operators that allow the specifier 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 in the formula, in much the same way as in temporal logic.

Syntax: the temporal operators. The alphabet of the TRIO language includes sets of names for variables, functions, and predicates, and a fixed set of basic operator symbols. Variables are divided into *time dependent* (TD) variables, whose value may change with time, and *time independent* (TI) variables, whose value is intended to be invariant with time. Every variable name x has an associated *type* or *domain*, which is the set of values the variable may assume. A distinguished domain, required to be numeric, is called the *Temporal Domain*. Every function name has an associated arity $n \geq 0$ (when $n=0$ the function is called a *constant*), and the indication of a type for every component of the domain and for the range. Similarly, every predicate name is associated with the number and type of its arguments. Like variables, predicates are divided into time dependent and time independent ones: time independent predicates always represent the same relation, while a time dependent predicates correspond to a possibly distinct relation at every time instant[4]. The predicates $<$, $\leq$, $=$, and all other usual predicates on numbers, are assumed to be time independent, so that the associated relational operations are applicable the Temporal Domain. Also, addition and subtraction are assumed to be total functions, with the usual properties, applicable to elements of the temporal domain. Symbols are divided into propositional symbols ($\wedge$ and $\neg$), the quantifier $\forall$, and a temporal operator symbol *Dist*.

The syntax of TRIO defines terms in the usual inductive way: every variable is a term, and every n-ary function applied to n terms is a term itself. A *formula* is inductively defined by the following clauses:

1. Every n-ary predicate applied to n terms of the appropriate types is a formula (atomic formula).
2. If A and B are formulas, $\neg A$ and $A \wedge B$ are formulas.
3. If A is a formula and x is a time independent variable, $\forall x A$ is a formula.
4. If A is a formula and t is a term of the temporal type, then $Dist(A,\ t)$ is a formula[5].

The formula $Dist(A, t)$ intuitively means that A holds at an instant laying t time units in the future (if $t > 0$) or in the past (if $t < 0$) with respect to the current time value, which is left implicit in the formula.

Abbreviations for the propositional operators $\vee$, $\rightarrow$, *true*, *false*, $\leftrightarrow$, and for the derived existential quantifier $\exists$ are defined as usual. A large number of

[4] In principle, functions may also be divided into time independent and time dependent, but this feature is not essential and for simplicity it is not introduced here.

[5] We are introducing minor modifications with respect to the original TRIO definition in [13].

derived temporal operators may be defined by means of quantification over TI variables in the temporal argument of *Dist*. These derived operators include all the operators of classical linear temporal logic. We mention, among others, the following ones

$$
\begin{array}{ll}
Futr(A,t) & \stackrel{\text{def}}{=} t > 0 \wedge Dist(A,t) \\
Past(A,t) & \stackrel{\text{def}}{=} t > 0 \wedge Dist(A,-t) \\
AlwF(A) & \stackrel{\text{def}}{=} \forall t(t > 0 \rightarrow Futr(A,t)) \\
AlwP(A) & \stackrel{\text{def}}{=} \forall t(t > 0 \rightarrow Past(A,t)). \\
Always(A) & \stackrel{\text{def}}{=} AlwP(A) \wedge A \wedge AlwF(A) \\
SomF(A) & \stackrel{\text{def}}{=} \neg AlwF(\neg A) \\
SomP(A) & \stackrel{\text{def}}{=} \neg AlwP(\neg A) \\
Sometimes(A) & \stackrel{\text{def}}{=} SomP(A) \vee A \vee SomF(A) \\
Lasts(A,t) & \stackrel{\text{def}}{=} \forall t'(0 < t' < t \rightarrow Futr(A,t')) \\
Lasted(A,t) & \stackrel{\text{def}}{=} \forall t'(0 < t' < t \rightarrow Past(A,t')) \\
Since(A_1,A_2) & \stackrel{\text{def}}{=} \exists t(t > 0 \wedge Past(A_2,t) \wedge Lasted(A_1,t)) \\
Since_w(A_1,A_2) & \stackrel{\text{def}}{=} AlwP(A_1) \vee Since(A_1,A_2)
\end{array}
$$

Futr(A,t) means that A will be true in the future, t units from now (*Past(A,t)* has the same meaning but respect to the past); *AlwF(A)* means that A will hold in all future time instants, while *AlwP* has the same meaning with respect to the past; *Always(A)* means that A holds in every time instant of the temporal domain; *SomF(A)* means that A will take place sometimes in the future, and *SomP* has the same meaning in the past; *Sometimes(A)* means that A takes place sometimes in the past, now or in the future; *Lasts(A, t)* means that A will be true in the next t time units; *Lasted(A, t)* means that A was true in the last t time units; $Since(A_1,A_2)$ means that A_2 took place sometimes in the past, and A_1 held since then; $Since_w(A_1,A_2)$ defines the *weak* version of Since, which does not require A_2 to actually take place.

A *TRIO specification* is a closed TRIO formula. Only closed TRIO formulas are considered, since it is well known that in formulas expressing some kind of system property all variables are quantified, although sometimes implicitly.

Example 1. A transmission line receives messages at one end and transmits them unchanged to the other end with a fixed delay. The time-dependent predicate *in(m)* means that a message m enters the line at the current time (left implicit); the predicate *out(m)* means that the same message m exits from the other end. The TRIO formula

$$Always(in(m) \rightarrow Futr(out(m), 5))$$

means that everytime a message m arrives a given time, then 5 time units later the same message m is emitted, i.e., the message does not get lost. The formula

$$Always(\forall m(out(m) \rightarrow Past(in(m), 5)))$$

means that no spurious messages are generated.

Example 2. Let *higherLevel* and *safetyLevel* be two significant temperature values for the security of a chemical plant. Let *temp* be a time dependent variable representing the present system temperature. If *lightSignal* and *soundAlarm* are two different alarms in the control system, the formulas

$$Always(lightSignal(on) \leftrightarrow temp \geq higherLevel)$$

$$Always(soundAlarm(on) \leftrightarrow temp \geq safetyLeve)$$

mean that the light alarm must be on if and only if the temperature reaches *higherLevel*, and the sound alarm must be activated if and only if the temperature reaches the *safetyLevel*. The fact that a security action must be taken whenever the pressure value exceeds a fixed threshold is expressed by the following formula

$$Always(pressure \geq valveTolerance \rightarrow Futr(Lasts(openGauge, k_1.pressure), k_2 \;/\; temp))$$

This formula specifies the gauge remains open for a duration that is proportional to the pressure, while the activation must be delayed by an interval inversely proportional to the current temperature.

3.2 TRIO's semantics

The concepts of satisfiability and validity of a TRIO formula with respect to suitable interpretations can be defined in much the same way as in classical first-order logic. A model-theoretic semantics for TRIO is based on the concept of temporal structure ([25]), from which one can derive the notion of evaluation function, that assigns to every TRIO formula a truth value for every time instant in the time domain.

A *interpretation structure* S assigns evaluation domains to variables, and values of the appropriate type to variable, function and predicate names occurring in formulas of the language. In particular, it associates a *temporal domain*, denoted by T, to temporal terms.

Traditional definitions of model-theoretic semantics introduce a meaning function S_i that assigns a value of appropriate type to terms and a truth value to formulas for every instant i of the time domain. A specification formula F is said to be *temporally satisfiable* for a given interpretation structure if there exists a time instant $i \in T$ for which $S_i(F) = true$. F is said to be *temporally valid* in the given interpretation iff $S_i(F) = true$ for every $i \in T$; it is *valid* if it is temporally valid in every syntactically adequate interpretation. An interpretation such that F is temporally satisfiable for it is called a *model* of F.

The definition of function S_i is however more complex when formulas are evaluated in a finite (time) domain. [24] introduces a model-parametric semantics that imposes restrictions on the evaluability of a formula at a time instant with reference to a given structure. A formula is considered not evaluable with respect to a given time instant if its evaluation at that time cannot be done without referencing a time point outside the time domain. In addition, the set of values

that can be assigned to the quantified variables of a formula must be adequately restricted to subsets of their types in order to prevent from exiting the time domain when evaluating the formula.

3.3 Verification of TRIO specifications

A TRIO specification can be used to model a reactive, real-time system because the physical and structural components of the modeled system have a logical counterpart in the constituents of the temporal structure. Physical components of the systems and immutable relations among them are represented by individual constants and time independent predicates; temporary relations and events are represented by time dependent predicates; values and measures of physical quantities that are subject to change are represented by time dependent variables. Functions can be used to describe some predefined and fixed operations of the specified system, and time independent variables are used as placeholders to express, through the use of quantifiers, existential and universal properties of the specified system. Thus, one model of a TRIO specification intuitively corresponds to one possible evolution (*history*) of the specified system, that satisfies the requirements expressed by the specification formula. A history can be specified as follows:

Definition 4 (Event, history). An event is a pair $< L, i >$, where

- L is a literal
- i is a time instant, belonging to the time domain.

A history is a set of events that temporally satisfies a TRIO specification.

For instance, referring to Example 1 in Section 3.1, the pair $< in, 1 >$ describes the event of receiving a message at time 1; the pair $< \neg out, 1 >$ describes that no message is sent at time 1. Figure 3 illustrates 3 histories for that example.

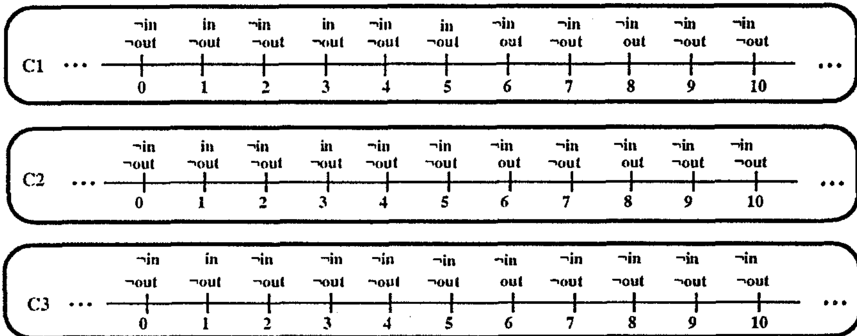


Fig. 3. Three executions for the line of example 1 in Section 3.1

Requirements validation. In order to validate requirements expressed as a TRIO specification, one may try to prove the satisfiability of the formula by constructing a model for it. In this view, some parts of the temporal structure to be constructed are assumed to be known, namely the temporal domain T, the domains for variables, and the interpretation of time independent predicates, which describe the static configuration of the specified system. Given a system specification as a TRIO formula and the static part of a structure adequate to its evaluation, the construction of the remaining parts of the structure determines the dynamic evolution of the modelled system: the events that take place and the values assumed by the relevant quantities.

If the interpretation domains for variables and the temporal domain T are all finite, the satisfiability of a TRIO formula is a decidable problem and effective algorithms to solve it can be defined. [7] presents an algorithm which, under the hypothesis of finite domains, determines the satisfiability of a TRIO specification (i.e., a closed TRIO formula) using a constructive method.

The main steps of the algorithm for verifying satisfiability are schematically shown in Figure 4. The specification formula is associated with a time value t that indicates the instant where it is assumed to hold; then a decomposition process is performed which transforms a formula into a set of simpler formulas, associated with possibly different instants, whose conjunction is equivalent to it. The decomposition uses well known (and intuitive) properties of the propositional operators, and treats universal (respectively existential) quantifications as generalized conjunctions (respectively disjunctions); it ends when each set of the subformulas, called a *tableau*, contains only literals. Every tableau that does not contain any contradiction (i.e., a literal and its negation) provides a compact representation of a model for the original formula, and thus constitutes a constructive proof of its satisfiability. Since each leaf tableau generated by the algorithm for verifying satisfiability corresponds to a history of the specified system, i.e., a temporal evolution of the system, this algorithm is called *history generator* (i.e., an interpreter that receives as input a TRIO formula and produces as result a set of histories that are compatible with such a formula).

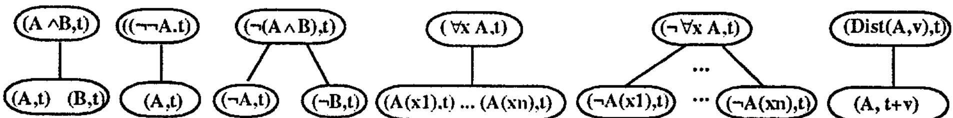


Fig. 4. Pictorial description of the decomposition of formulas by the tableaux algorithm.

Figure 5 shows part of the tableaux tree generated by a history generator for the formula Always(pressure ≥ valveTolerance → Futr (Lasts (openGauge, k_1·pressure), k_2 / temp)). It is shown for a generic time value i ranging from the minimum value of the time domain to the maximum. Since both leaves do not include contradictions, both represent models of the formula. Please notice that, for graphic reasons we omit the consonants in the predicate and variable names

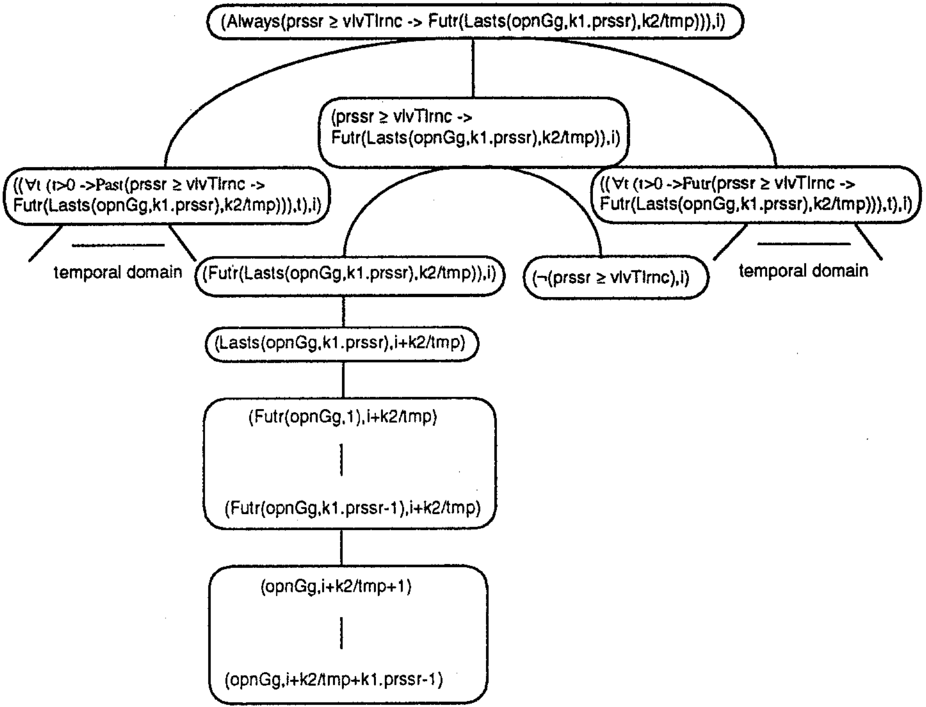


Fig. 5. The tableaux generated by the histories generator for the formula: Always(prssr $\geq$ vlvTlrnc $\rightarrow$ Futr (Lasts(opnGg,k_1.prssr), k_2 / tmp)).

The study of the complexity of the above algorithm, discussed in [7], shows that it is exponential in the number of existential quantifications, with the cardinality of the domains of the quantified TI variables appearing as the base of the exponential, while it is hyperexponential with respect to the number of universal quantifications, with the cardinality of the domain of the quantified variables appearing as the exponent.

Specification testing. Executability of TRIO formulas is also provided at lower levels of generality: the tableaux algorithm can be adapted to verify that a given temporal evolution of the system (a history) is compatible with the specification. This operation is called *history checking*, since it is analogous to what is called model checking, in the literature regarding branching-time temporal logic ([2]). Model checking refers to the operation of verifying whether a given state graph, or state automaton implementing a system, is a model of the specification formula. Hence it is equivalent to proving that every possible execution of the automaton satisfies the formula. Instead, in a linear time logic like TRIO,

history checking refers to only *one* possible evolution of the system. That is, a history checker is an interpreter that receives as input a TRIO formula and a history and states whether the history is compatible with the given formula. As we already said, history checking is implemented through a specialization of the tableaux algorithm, whose main steps are shown in Figure 6. Now each tableau includes only *one* formula associated with a time instant at which it must be evaluated. An and/or tree is built and the literals obtained in the leaf nodes are checked against the history.

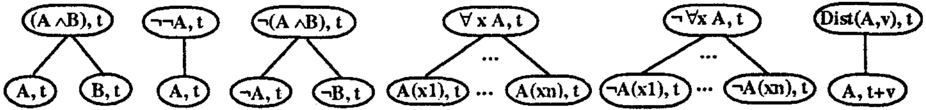


Fig. 6. Pictorial description of the decomposition of formulas by the history checker.

Complexity of the history checking algorithm has been shown to be exponential with respect to the dimension of the formula ([7]), i.e., the number of its quantifications and binary operators. It should however be noticed that the cardinality of the domains of the quantified variables appears now as the base of the exponential, not as the exponent (like in the more general algorithm for deciding satisfiability of formulas). In other words, for a given formula, the complexity of the history checking algorithm is a polynomial function of the cardinality of the evaluation domains. This result might be considered discouraging; however we point out that the dimension of the formula is usually relatively small with respect to the cardinality of the evaluation domains, so that, in the total complexity of checking the specification formula, the exponential factor has a limited influence with respect to the polynomial one. This was confirmed by experimental use of the prototype in cases of practical interest ([3]).

The history checking algorithm was the basis for the design and implementation of a prototype tool for specification testing. A set of facts representing a possible system evolution is *tested* with reference to a formula describing all the desired system behaviors. A history is represented by a set of time dependent ground literals, by the time independent predicates and functions which constitute the frame, and by the values for the time dependent variables.

The tool admits the possibility that the set of ground literals included in the history does not cover all possible instants of the temporal domain or all the argument values. In that case the user may choose to evaluate the formula under the closed world assumption or not. If the closed world assumption holds then the history is complete by definition because all the events not reported in the history are implicitly considered as false. Thus, the tool checks whether a formula is evaluable and then if it is satisfied by the history. Instead, whenever the closed world assumption does not hold, the checker is also able to recognize whether the history does not provide sufficient information to assign a truth value to the formula. In this case the formula is evaluated with respect to a

Using specifications to verify implementations. [21] proposes a method and a tool for the derivation of functional test cases for real-time systems, starting from a specification given in TRIO. [21] defines the notion of test cases adequate to performing functional testing of real-time systems specified in TRIO, and shows how the history generator and history checker can be effectively used, both to generate test cases and to support the testing activity itself.

A history (or a model) of a TRIO formula can be interpreted as a test case, since it represents an evolution trace of the modelled system and hence it can be compared with an actual execution of the system. In this view, the two mentioned interpreters (i.e. the algorithm for verifying satisfiability and the history checker) can become the core of a tool that allows to systematically generate test cases for the specified system and validate its responses to the provided stimuli. By this way, not only possible *stimuli* (i.e., system inputs) are generated from system specifications to test the system under verification, but also system *reactions* (i.e., outputs) are provided to check whether the system behavior really complies with the desired properties, thus solving the oracle problem ([18]) (i.e., given the inputs determine the expected outputs).

The TRIO language, unfortunately does not distinguish input events (data or commands introduced into the system) from output events (data or signals generated by the system). During the testing activity, however, it is essential to understand which events flow from the external environment to the system, and which flow from the system to the external environment. The solution adopted by the TRIO test case generator does not perform automatically such a classification, but solves the problem through interaction with the user. The user partitions predicate names of the specification formula into input, output, and input/output predicates.

Since each leaf tableau generated by the algorithm for verifying satisfiability corresponds to a history of the specified system, it is clear that the algorithm may generate, for some specification formulas, a very large number of histories, much more than those needed for performing an effective testing. [21] proposes some criteria to cope with such kind of complexity. When the complexity is high, the tool requires the user interaction to select the criteria it must follow or to use the history checker to check if the set of facts generated up to that moment satisfies the formula (i.e., it is a history).

3.4 Property proving

This section surveys the axiomatic definition of TRIO presented in [6]. Following the axiomatization, one can prove properties of TRIO specifications. The proof cannot be performed in a purely mechanical fashion, but requires human interaction in the general case.

Since TRIO allows almost any kind of interpretation domains to be used in specifications, any axiom system for it should include a first order theory for all three-value logic ([22]) which includes a third, "unknown" value.

of the used domains (say, real numbers for temporal domain, integers for some variables, booleans for others, etc.). Thus the final axiomatization depends on the selected domain. Following the same approach as adopted in a variety of temporal logics ([28, 20, 17]), all the valid formula of the chosen domains are implicitly assumed to hold as additional axioms in the TRIO's axiomatization.

TRIO's axioms are given below. For convenience they are partitioned into *general axioms*, which are shared with any first-order theory with equality, and *temporal axioms*, which are peculiar of the language. A universal axiom schema is added at the end of both class.

Let $\alpha, \beta, \omega, \ldots$ denote any TRIO formula; let s, v, u ...denote any term of a generic domain, whereas t, t1, t2, ...denote any term of temporal type; let x, y, ...denote any variable and c any constant.

General axioms These, in turn, are split into first-order predicate axioms and into equality axioms as shown below.

First order axioms

1. All instances of propositional calculus tautologies
2. $\forall x\ \alpha \rightarrow \alpha^x_s$ where s is a term substitutable for x in α [6] ([22])
3. $\forall x\ (\alpha \rightarrow \beta) \rightarrow (\forall x\ \alpha \rightarrow \forall x\ \beta)$
4. $\alpha \rightarrow \forall x\ \alpha$ if x is not free in α

Equality axioms

5. $s = s$, for any term s
6. $u = s \rightarrow (\alpha \rightarrow \alpha')$ where u and s are terms, α' is obtained from α by substituting zero or more occurrences of u in α by s.

Temporal axioms

7. $Dist(\alpha, 0) \leftrightarrow \alpha$
8. $Dist(\alpha, t1 + t2) \leftrightarrow Dist(Dist(\alpha, t1), t2)$
9. $Dist(\alpha \rightarrow \beta, t) \leftrightarrow (Dist(\alpha, t) \rightarrow Dist(\beta, t))$
10. $Dist(\neg\alpha, t) \leftrightarrow \neg Dist(\alpha, t)$
11. $\alpha \rightarrow Alw(\alpha)$ if α is time independent

Generalization For each formula ω in the above list 1 through 11, all formulas of the kind $Alw(\omega)$, and all their applications of universal quantifications (generalizations) are TRIO axioms.

There is a single rule of inference, namely Modus Ponens (MP). By using classical Hoare's notation, it is denoted as

$$\frac{\Gamma \vdash \alpha, \Gamma \vdash \alpha \rightarrow \beta}{\Gamma \vdash \beta}$$

Axioms 7 through 10 describe the essential properties of the basic temporal operator Dist: when in the formula $Dist(\alpha, t)$ the temporal argument t is 0 then the other argument, α, is asserted at the current instant; furthermore, nested applications of the operator compose additively, according to the properties of the temporal domain, and the operator is transparent with respect to propositional operators of its non-temporal argument. Axiom 11 simply states the time

[6] As usual, α^x_s denotes the result of substituting any free occurrence of x in α by s.

invariance of time independent formulas (those which do not contain any time dependent predicate or temporal operator): if the formula is true *now* then it is true at any time instant of the temporal domain (the converse is trivially true, and in fact the corresponding formula, $Alw(\alpha) \rightarrow \alpha$, is a theorem whose proof is immediate).

TRIO's axiomatization provides also the counterpart of well-known metatheorems[7] which hold for the most widely accepted axiomatizations of first-order logic, namely the Generalization theorem (GEN), the Deduction theorem (DED) and the Existential Instantiation theorem (EI). Similarly, since TRIO's axiomatization includes a standard first-order part, all the derived inference rules usually employed for first-order logic ([22]) are also valid in TRIO. Moreover, the axiomatization provides some useful temporal metatheorems:

Temporal Translation Theorem (TT)

This metatheorem asserts that if a formula α can be proved under a given set of assumptions that hold at the present time instant and all these assumptions hold at a different time instant, then it can be proved that α holds at that time instant too. The metatheorem is formalized as follows.

$$\text{if } \Gamma \vdash \alpha \text{ then } \{Dist(\gamma, t) | \gamma \in \Gamma\} \vdash Dist(\alpha, t)$$

Temporal Generalization Theorem (TG)

This metatheorem is an extension to the preceding result. It states that if the set of assumptions on which the proof of a property is based is true in every instant of the temporal domain, then the proven formula is also always true. Formally,

if $\Gamma \vdash \alpha$ and every formula of Γ is of the type $Alw(\gamma)$ or is time independent, then $\Gamma \vdash Alw(\alpha)$.

An important corollary of TG is obtained by taking $\Gamma = \emptyset$. In this case TG reduces to: if $\vdash \alpha$ then $\vdash Alw(\alpha)$. This corresponds to the intuitive fact that if property α is derived without making any assumption about the current time instant, then α holds at every time instant. Another consequence of TG is that any theorem τ of first-order logic is not only inherited as such in TRIO, but its temporal generalization, $Alw(\tau)$ is also a theorem. For instance, $Alw(\alpha(t) \rightarrow \exists z\ \alpha(z))$ holds by the fact that $\alpha(t) \rightarrow \exists z\ \alpha(z)$ is a theorem in any first-order logic.

3.5 Towards more usable real-time logic languages

TRIO is a quite terse language. It is an excellent notation for mathematically reasoning about specifications, but it is difficult to use in practice. Specifiers and readers have no ways of mastering the complexity of large specifications; no application-specific abstractions are provided to support end-users in the verification of a specification. Following the spirit of what was done for net-based

[7] Metatheorems are properties of the axiomatization. Instead, theorems are derived from deductions using the axiomatization.

specifications, TRIO is viewed as the semantic kernel notation of a specification environment, not as the notation used in practice. Other language layers have therefore been defined on top of TRIO.

TRIO+ ([26]) is an object-oriented extension of TRIO. TRIO+ allows the partition of the universe of objects into classes, the inheritance of relations among classes, and provides mechanisms such as inheritance and genericity to support reuse of specification modules and their top-down, incremental development. Moreover, an expressive graphic representation of classes in terms of boxes, arrows, and connections is defined. Such a representation allows depiction of class instances and their components, information exchanges, and logical equivalences among (parts of) objects.

ASTRAL ([8, 9]) is another linguistic layer defined on top of TRIO. ASTRAL views a real-time system under specification as a collection of abstract machines. Abstract machines may communicate with one another via exported variables. They may also interact with the external environment, which may cause state transitions to occur. ASTRAL is formally defined by means of a translation scheme into TRIO. The TRIO "code" generated by the translation may then be manipulated by the available TRIO tools. In particular, it is possible to test a specification by history checking.

4 The Dual Language Approach

An increasing interest is recently arising on the so-called dual-language approach, in order to support a complete formalization of specifications and the corresponding analysis [28]. In a dual language approach two entities are distinguished: a set of properties one wishes to verify and the system (or system part) about which these properties are to be verified. The dual language approach requires both entities to be formally described: the properties are described using an assertional (descriptive) language and the system is described using a model-based (operational) language. In this way properties are formally stated and can be formally proved. [6] proposed a dual language method where properties are expressed in terms of TRIO and the systems are modeled by means of TB nets. One may view TRIO as the language in which abstract properties and requirements are formally stated, while TB nets are used to model a more concrete operational description of an abstract implementation.

The basic idea of the method proposed in [6] consists of an axiomatization of the behavior of timed Petri nets in terms of TRIO axioms and proof rules, in the same style as Hoare's rules are provided for Pascal-like programs. Then, net properties, such as marking and firing conditions, are expressed as TRIO formulas and their validity is proved using the axiomatization presented in Section 3.4. The method supports the verification of properties of any kind of nets, whereas existing mechanical methods only apply to restricted subclasses and do not scale

classical benchmarks for the analysis of concurrent and real-time systems, such as an elevator system and a real-time version of the dining philosophers problem. For example, in the case of the elevator system, given some assumptions, it is possible to prove that if a person calls the elevator in certain circumstances, it will arrive within the next Δ time units; or, if a person pushes a button to close the elevator's doors and a person tries to enter the elevator while its doors are closing, the doors reopen.

5 Conclusions

In this paper, we surveyed the work done by our group in the area of specification and verification of reactive, real-time systems. Research is still active in the areas we reviewed here. For example, complete formal treatment of the graphical language extensions in CABERNET is still under scrutiny. In the case of TRIO, work is presently addressing further issues of higher-level language layers defined on top of the kernel (such as the cited TRIO+ and ASTRAL) and interlevel translation schemes, the problem of handling different time granularities in a specification, and others. The dual language approach is under further investigation, and will probably lead to an integration of the present environments supporting the two specification styles, nets and real-time logic in a single comprehensive environment.

Acknowledgements

The work we survey in this paper is the result of the work of many individuals over the past five years. The results we present could have never been achieved without their insights and enthusiasm. In particular, we wish to thank Dino Mandrioli, for his contributions to the whole research field, Mauro Pezzé for his contributions to CABERNET, Angelo Morzenti for his contributions to TRIO, Dick Kemmerer and Alberto Coen for their contributions to ASTRAL, Sandro Morasca for his contributions to CABERNET and test case generation from TRIO specifications, Pierluigi San Pietro for his contributions to TRIO+. Several generations of students contributed to the design of the current prototype environments.

References

1. Bellettini, C., Felder, M., Pezzè, M.: MERLOT: A tool for analysis of real-time specifications. Proceedings of the 7th International Workshop on Software Specifications and Design, Los Angeles, California, 1993. (to appear)
2. Clarke, C., Emerson, E., Sistla, S.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM-Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986.

3. Coen, A., Morzenti, A., Sciuto, D.: Specification and verification of hardware systems using the temporal logic language TRIO. In *Computer hardware description languages and their application,* Borrione, D. and Waxman, R., IFIP, North-Holland, Marseille, France, April 1991, pp.43–62.
4. Dillon, L.K., Avrunin, G.S., Wileden, J.C.: Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. ACM-Transactions on Programming Languages and Systems, Vol. 10, No. 3, pp. 374–402, July 1988.
5. Felder, M., Ghezzi, C., Pezzè M.: Analyzing refinements of state based specifications: the case of TB nets. Proceedings of International Symposium on Software Testing and Analysis 1993, Cambridge, Massachusetts. (to appear)
6. Felder, M., Mandrioli, D., Morzenti, A.: Proving properties of real-time systems through logical Specifications and Petri Nets Models. IEEE-Transactions on Software Engineering (to appear). *Also in Tech-Report 91-072, Dip. di Elettronica-Politecnico di Milano, December 1991.*
7. Felder, M., Morzenti, A.: Specification testing for real-time systems by history checking in TRIO. Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, May 1992.
8. Ghezzi, C., Kemmerer, R.A.: ASTRAL: An assertion language for specifying real-time systems. Proceedings of the 3rd European Software Engineering Conference, Milano, Italy, October 1991.
9. Ghezzi, C., Kemmerer, R.A.: Executing formal specifications: the ASTRAL to TRIO translation approach. TAV'91, Symposium on Testing, Analysis and Verification, Victoria, Canada, October 1991.
10. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A general way to put time in Petri nets. Proceedings of the 4th International Workshop on Software Specifications and Design, Monterey, California, April 3-4, 1987.
11. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A unified high-level Petri net formalism for time-critical systems. IEEE Transactions on Software Engineering, Vol. 17, No. 2, February 1991.
12. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: Symbolic execution of concurrent programs using Petri nets. Computer Languages, April 1989.
13. Ghezzi, C., Mandrioli, D., Morzenti, A.: TRIO: A logic language for executable specifications of real-time systems. Journal of Systems and Software, June 1990.
14. Ghezzi, C., Morasca, S., Pezzè, M.: Timing analysis of time basic nets". *submitted for publication*
15. Gomaa, H.: Software development of real-time systems. Communications of the ACM, Vol. 29, No. 7, July 1986.
16. Hatley, D.J., Pirbai, I.A.: *Strategies for Real-Time System Specification.* Dorset House, 1988.
17. Henzinger, T., Manna, Z., Pnueli, A.: Temporal proof methodologies for real time systems. Proceedings of the 18th ACM Symposium on Principles of Programming Languages, pp. 353–366, 1991.
18. Howden, W.E.: *Functional Program Testing & Analysis.* Mc Graw Hill, 1987.
19. Kemmerer, R.A.: Testing software specifications to detect design errors. IEEE Transactions on Software Engineering, Vol. 11, No. 1, January 1985.
20. Koymans, R.: Specifying Message Passing and Time-Critical Systems with Temporal Logic. PhD Thesis, Eindhoven University of Technology, 1989.

21. Mandrioli, D., Morzenti, A. and Morasca, S.: Functional test case generation for real-time sytems. Proceedings of 3rd International Working Conference on Dependable Computing for Critical Applications, IFIP, 1992 pp.13–26.
22. Mendelson, E.: *Introduction to mathematical logic.* Van Nostrand Reinold Company, New York, 1963.
23. Morasca, S. and Pezzè, M.: Validation of concurrent Ada programs using symbolic execution. Proceedings of the 2nd European Software Engineering Conference, LNCS 387, pages 469–486. Springer-Verlag, 1989.
24. Morzenti, A., Mandrioli, D., Ghezzi, C.: A model parametric real-time logic. ACM Transactions on Programming Languages and Systems, Vol. 14, No. 4, pp. 521–573, October, 1982.
25. Morzenti, A.: The Specification of Real-Time Systems: Proposal of a Logic Formalism. PhD Thesis, Dipartimento di Elettronica, Politecnico di Milano, 1989.
26. Morzenti, A., San Pietro, P.: An object oriented logic language for modular system specification. Proceedings of the European Conference on Object Oriented Programming '91, LNCS 512, Springer Verlag, July 1991.
27. Nagl, M.: A tutorial and bibliography survey on graph grammars. LNCS 166, Springer Verlag, 1985.
28. Ostrof, J.: *Temporal Logic For Real-Time Systems.* Research Studies Press LTD., Advanced Software Development Series, Taunton, Somerset, England, 1989.
29. Pezzè, M. and Ghezzi, C.: Cabernet: a customizable environment for the specification and analysis of realtime systems. *submitted for publication*, 1993.
30. Quirk, W.J.: *Verification and Validation of Real-Time Software.* Springer Verlag, Berlin, 1985.
31. Reisig, W.: *Petri Nets: an Introduction.* Springer Verlag, 1985.
32. Smullian, R.M.: *First Order Logic.* Springer Verlag, Berlin, 1968.
33. Taylor, R.: A general-purpose algorithm for analyzing concurrent programs. Communications of the ACM, Vol. 26, No.5, pp. 362–376, May 1983.