Model-checking TRIO specifications in SPIN[*]

Angelo Morzenti[1], Matteo Pradella[2], Pierluigi San Pietro[1], Paola Spoletini[1]

[1]Dipartimento di Elettronica e Informazione, Politecnico di Milano,
P.za Leonardo da Vinci 32,
20133 Milano, Italia
[2]CNR Istituto di Elettronica e di Ingegneria dell'Informazione e delle Telecomunicazioni, sez. Milano
Via Ponzio 34/5,
20133 Milano, Italia
email: {morzenti, pradella, sanpietr, spoleti}@elet.polimi.it

Abstract. We present a novel application on model checking through SPIN as a means for verifying purely descriptive specifications written in TRIO, a first order, linear-time temporal logic with both future and past operators and a quantitative metric on time. The approach is based on the translation of TRIO formulae into Promela programs guided by an equivalence between TRIO and 2-way alternating Büchi automata. An optimization technique based on the modularized TRIO specifications is also shown. The results of our experimentation are quite encouraging, as we are able to verify properties of the Railway Crossing Problem, a well-known benchmark used in the Formal Methods community, for values of the temporal constants that make the verification totally infeasible with traditional tools and approaches.

Keywords: temporal logic, model checking, modular specifications, Spin.

1. Introduction

TRIO is a first order, linear-time temporal logic with both future and past operators and a quantitative metric on time, that has been extensively applied to the specification, validation and verification of critical, real-time systems [13]. The logic TRIO has also been enriched with constructs, inspired by Object-Oriented Analysis and Design, for structuring specifications into a set of modules with clearly defined interfaces, thus providing a very useful support to the structuring and management of specification of highly complex systems, and at the same time building a bridge from requirements specification to high-level design. Over the years a variety of methods and tools have been defined to support typical V&V activities in TRIO. Validation of TRIO specification is obtained through generation of execution traces or checking of such simulations for consistency against the TRIO specification [14]. The execution traces derived from TRIO specifications, suitably classified and annotated, can be employed as functional test cases to support post-design verification [15]. A more systematic and general means of validation and verification can be pursued through proof of

[*] Work partially supported by the MIUR projects: "QUACK: Piattaforma per la qualità di sistemi embedded integrati di nuova generazione" and "FIRB: Applicazioni della Teoria degli Automi all'Analisi, alla Compilazione e alla Verifica di Sistemi Critici e in Tempo Reale."

properties derivable from the TRIO axioms composing the specification. As TRIO is a first order logic that includes arithmetic on the temporal domain, it is undecidable in the general case, hence two basic approaches were devised to address the goal of providing mechanical support to the verification of TRIO specifications. One consists of adopting a deductive approach, based on the definition of a suitable axiomatization of the logic and on its encoding in the notation of a general purpose theorem prover, such as PVS [16]; this allows the construction of a tool supporting the semiautomatic (i.e., manual with assistance from the tool) derivation of system properties in the form of theorems. In this approach one maintains the generality and expressive power of the full language, at the price of sacrificing the construction of a completely automatic (so-called push-button) tool. Another, complementary approach to verification aims at the construction of tools that are fully automatic, or at least provides a quite strong support to the designer: it consists of defining a decidable approximation of a specification, upon which applying methods and algorithms for deciding satisfiability or, less generally but more efficiently, for checking satisfaction with respect to a given interpretation structure. In the past, the latter approach has been based on finitizing the domains of the variables that appear in the specifications [7], leading to the construction of tools built either on tableaux-based verification procedures [7, 14, 15] or on the encoding of TRIO into propositional languages and the use of sophisticated SAT-solvers [17]. This approach has the advantage of allowing the development of "pushbutton" tools, such as the cited [17], but the approximations introduced to make verification decidable (and feasible) may not assure the conservation of properties of the original specification. For instance, time must be finite rather than infinite for a SAT-solver to be used, making the verification of various fairness and liveness properties hard or even impossible.

In the present paper, we pursue a different approach for the mechanical verification of TRIO specifications, namely the definition of a decidable fragment of the logic that includes a suitable subset of its original operators, and the use of a well-known model checker such as SPIN [18] to perform proof of properties and simulation (in the form of generation of execution traces).

A TRIO specification consists of a set of temporal logic formulae that describe the desired properties of the system being designed; this kind of specification does not include any operational component (such as a state-transition system) that can *generate* values for the elements of the alphabet of the specification (predicates and variables representing the state of the various parts composing the system under design); on the contrary, and similarly to what occurs in any other purely descriptive specification notation, TRIO formulae define constraints on the values that the items appearing in the specification can assume in the "legal" (i.e., consistent with the specification) evolutions. Therefore, the problem of property proving in TRIO, when addressed in a model-checking approach, takes a form that is rather different from the one typically encountered in the literature on the subject. It is formulated in terms of the validity of a logic formula of the kind *specification* → *property*, where the premise *specification* is still a set of TRIO formulae describing properties that are *assumed to hold* for the analyzed system, and *property* is another TRIO formula describing the conjecture that we want to prove to be implied by the properties stated in the premise. As it happens in some other approaches based on model-checking, what we actually check might in fact be the negation of the above implication, i.e., ¬(*specification* → *property*) and the counterexamples generated by the model-checker in this case can be used as simulations or functional test cases for the desired property. Therefore, in our approach what we call *specification* has a role similar to the one played by the so-called *model* in the usual model checking scenery (e.g., a Promela program in SPIN) while what we call *property* in the above implication is usually called *specification* and takes the form of a formula in temporal logic (e.g., an LTL formula in SPIN).

Our approach to model checking TRIO specifications is based on the translation of the TRIO formulae into a set of Promela processes, derived from a well known correlation between temporal logic and alternating automata [10]. As opposed to previous approaches, however, the Promela code generated from TRIO formulae performs an actual simulation of an alternating automaton, rather than simulating a Büchi automaton equivalent to the alternating one, resulting in a Promela code whose size is essentially proportional to the length of the TRIO specification (although of course the state space may not be affected in either way). This is by itself a remarkable result since the TRIO logic, which contains metric and past operators, is quite concise compared with propositional, future-time temporal logics like LTL. Our approach can be naturally compared with recent works appeared in the literature (such as those on LTL2BA [2] and Wring [11]) that aim at the translation of LTL properties into Büchi automata and then Promela programs--such comparison will be provided in Section 4). We point out, however, that the result of those tools is usually the construction, as in the traditional model-checking scenario, of a so-called *never claim*, i.e., an automaton specifying the negation of a temporal logic property over an already available state-transition system. In our approach, instead, the Promela processes obtained from the translation of the TRIO specification act globally as an acceptor of a language defined over the alphabet of the specification, and therefore they must be coupled with some additional Promela program fragments generating the values, over time, for the logical variables that constitute the specification alphabet. This ”generative” component of the Promela program can trivially be obtained by encoding a systematic, exhaustive enumeration of all possible variable values over time, but this can potentially lead to a combinatorial explosion of the search state space, thus making the proposed approach infeasible in practice. To address this issue we adopt two basic techniques, which can be roughly described as follows. First, we exploit the modular structure of TRIO specifications to obtain a maximum of encapsulation in the verification process, so that the execution of any Promela process verifying a TRIO subformula will be affected only by the values for the elements of the alphabet occurring in that subformula, and not by other ones. Second, we restrict the purely combinatorial generation of values to those variables that are truly independent from every other one; from a methodological standpoint, these variables can be easily recognized as the elements of the specification alphabet corresponding to components of the specified system that are a “pure input”, thus ruling out any computed state or output value. These translation techniques, combined with other minor optimizations, related for example with the management of TRIO past-time operators, allowed us to perform efficiently the verification in SPIN of a system which is universally adopted as a benchmark in the verification of time-critical systems, namely the Railroad Crossing Example, fully described by a TRIO specification.

The remaining sections are organized as follows. To make the paper self-contained, Section 2 provides a brief introduction to the TRIO logic and outlines the decidable subset of the language on which we perform verification via model checking. Section 3 discusses the relation between TRIO and 2-way alternating automata (also introducing an extension of the classical model that explicitly deals with finite counters to account for TRIO’s quantitative notion of time) and the translation schema from TRIO to Promela. Section 4 deals with specific issues related with verification via model-checking: it motivates the introduction of a network of processes, gives a rationale for the optimization performed, and discusses the results obtained on the case study, occasionally providing comparisons with related approaches. Finally, Section 5 draws conclusions and outlines directions of future research.

2. A brief introduction to TRIO

TRIO formulae are built much in the same way as in traditional mathematical logic, starting from variables, functions, predicates, predicate symbols, and quantifiers (a detailed and formal definition of TRIO can be found in [7]).

In TRIO, first-order variables and quantifiers are allowed over finite or infinite, dense or discrete, domains, including the time domain. Besides the usual propositional operators and the quantifiers, one may compose TRIO formulae by using a single basic modal operator, called *Dist*, that relates the *current time*, which is left implicit in the formula, with another time instant: the formula *Dist(F, t)*, where *F* is a formula and *t* a term indicating a time distance, specifies that *F* holds at a time instant at *t* time units from the current instant. Many *derived temporal operators* can be defined from the basic *Dist* operator through propositional composition and first order quantification on variables representing a time distance. The traditional operators of linear temporal logics can be easily obtained as TRIO derived operators. For instance, SomF (Sometimes in the Future) corresponds to the "Eventually" operator of temporal logic. Moreover, it can be easily shown that the operators of several versions of temporal logic (e.g., interval logic) can be defined as TRIO derived operators. This argues in favor of TRIO's generality since many different logic formalisms can be described as particular cases of TRIO.

For instance, the following TRIO formula specifies that every message *m* entering a channel is always delivered within 10 time instants. The meaning of the various symbols is obvious once the (bounded) derived temporal operator WithinF(A,t) is interpreted as "A will hold within t instants in the future":

$$\mathrm{AlwF}(\forall m\ (\mathrm{in}(m) \rightarrow \mathrm{WithinF}(\mathrm{out}(m),10))).$$

A decidable subset of TRIO

In general, TRIO formulae, adopting the full-fledged power of first-order logic are undecidable. In this paper, however, we consider a decidable subset of TRIO, where the time domain is the set of natural numbers, no time variable is allowed and every other domain is finite. This version of TRIO is basically a syntactically sugared, more concise version of PLTLB, Propositional Linear Time Temporal Logic with Both past and future operators (here we follow the terminology introduced by [1], while using the standard name "LTL" to denote the future fragment of PLTLB).

The syntax of TRIO formulae is described by the following grammar, where ϕ is the axiom, p stands for any element in a finite set *Ap* of atomic propositions, c stands for any element of a finite set of natural numbers, and {*Since, Until, Futr, Past, Lasts, Lasted, (,)*} is the set of terminal symbols:

$$\phi ::= p \mid \phi \wedge \phi \mid \neg \phi \mid \mathit{Until}(\phi,\phi) \mid \mathit{Since}(\phi,\phi) \mid \mathit{Futr}(\phi,c) \mid \mathit{Past}(\phi,c) \mid \mathit{Lasts}(\phi,c) \mid \mathit{Lasted}(\phi,c)$$

Usual shorthands for logical symbols such as *true*, *false*, $\vee$, $\rightarrow$, $\equiv$ are standard. Notice that, to rule out negative numbers, a pair of operators, namely *Futr* and *Past*, replace the basic TRIO Dist operator. Table 1 introduces a few derived temporal operators, with a short definition and explanation.

Operator	**Definition**	**Intuitive Meaning**
SomF(F)	Until(true, F)	Sometimes F holds
SomP(F)	Since(true, F)	Sometimes F held
AlwP(F)	$\neg$SomP($\neg$F)	F always held in the past
AlwF(F)	$\neg$SomF($\neg$F)	F will always hold
$\mathrm{Since}_{ii}(F_1, F_2)$	$F_1 \wedge \mathrm{Since}(F_1, F_1 \wedge F_2)$	Since, both temporal extremes included
$\mathrm{UntilW}_{ie}(F_1, F_2)$	$F_1 \wedge (\mathrm{Until}(F_1,F_2) \vee \mathrm{AlwF}(F_1))$	Weak until
WithinF(F,c)	$\neg$Lasts($\neg$F,c)	F will hold within c instants in the future
WithinP(F,c)	$\neg$Lasted($\neg$F,c)	F held within c instants in the past
$\mathrm{Lasts}_{ie}(F,c)$	F $\wedge$ Lasts(F,c)	Lasts(F,c) with the current instant included
$\mathrm{Lasted}_{ie}(F,c)$	F $\wedge$ Lasted(F,c)	Lasted(F,c) with the current instant included
$\mathrm{WithinF}_{ii}(F,c)$	F $\wedge$ WithinF(F,c) $\wedge$ Futr(F,c)	WithinF, both temporal extremes included
$\mathrm{WithinP}_{ii}(F,c)$	F $\wedge$ WithinP(F,c) $\wedge$ Past(F,c)	WithinP, both temporal extremes included
UpToNow(F)	Past(F,1)	F held (for at least one instant) until now

Table 1. A sample of derived temporal operators in TRIO.

For instance, the formula *AlwF(push → Lasts(on, 6))* may specify the property that from now on the event of pushing a button causes a lamp to be on for the next 6 instants.

Semantics

The standard TRIO semantics is called *model parametric semantics* [7], and it is based on Kripke structures that can accommodate different time domains. However, since the version of TRIO used in this paper is equivalent to PLTLB, we simply define the semantics of basic TRIO formulae as a translation into the standard PLTLB.

A PLTLB formula has the following syntax:

$\phi ::= p \mid \phi \wedge \phi \mid \neg \phi \mid \phi \mathrm{U} \phi \mid \phi \mathrm{S} \phi \mid \mathrm{X}\phi \mid \mathrm{P}\phi$

where U and S are the Until and Since operators, respectively, and X and P are the Next and Previous operators. We also define, for every integer constant $t \geq 0$, $X^t\ \phi$ as X X…X ϕ (X repeated t times) if $t>0$, ϕ if $t=0$. $P^t\ \phi$ is its past conterpart and is defined analogously. Other standards operators, such as the eventually operator F (also denoted as <>) and the globally operator G (also denoted as []) can be defined as usual, e.g.: $F\ \phi = \mathrm{true}\ U\ \phi$, $G\ \phi = \neg F \neg \phi$.

The translation δ from TRIO formulae to PLTLB formulae is defined inductively as follows:

$\delta(\phi) = \phi$ if $\phi \in$ Ap

$\delta(\phi_1 \wedge \phi_2) = \delta(\phi_1) \wedge \delta(\phi_2)$

$\delta(\neg\phi) = \neg\delta(\phi)$

$\delta(\mathrm{Until}(\phi_1, \phi_2)) = \delta(\phi_1)\ \mathrm{U}\ \delta(\phi_2)$

$\delta(\mathrm{Since}(\phi_1, \phi_2)) = \delta(\phi_1)\ \mathrm{S}\ \delta(\phi_2)$

$\delta(\text{Futr}(\phi,t)) = X^t\,\delta(\phi)$

$\delta(\text{Past}(\phi,t)) = P^t\,\delta(\phi)$

$\delta(\text{Lasts}(\phi,t)) = \text{true if } t = 0,\ X\,\delta(\phi) \wedge X^2\,\delta(\phi) \wedge \ldots \wedge X^{t-1}\,\delta(\phi) \text{ if } t>0.$

$\delta(\text{Lasted}(\phi,t)) = \text{true if } t = 0,\ P\,\delta(\phi) \wedge P^2\,\delta(\phi) \wedge \ldots \wedge P^{t-1}\,\delta(\phi) \text{ if } t>0.$

Notice that the usage of both past and future operators is widely recognized [5] as making specifications simpler and more concise than using either only future or only past operators. TRIO adds another level of succinctness because of the metric operators Lasts and Lasted (and their duals WithinF and WithinP).

For instance, a simple TRIO formula such as:

$$\text{WithinF}(\text{Lasts}(B,h), k)$$

for some h, k > 0, may be expressed only with a LTL formula, whose length is proportional to $h \cdot k$.

Modular TRIO specifications

The TRIO logic is augmented with object-oriented constructs for supporting modular specifications [6]. A modular TRIO (often called TRIO$^+$) specification is built by defining suitable *classes*. Classes can be *simple* or *structured*. Simple classes contain a set of logic axioms along with the declaration of the elements of the alphabet (i.e., the predicate and function signatures) and a definition of those predicates, variables, and functions that belong to the class interface. Predicates, variables, and functions are collectively called *items*. Simple classes are graphically represented as boxes with arrows representing *input* and *output* items in the interface. Truly modular specifications are obtained by defining *structured classes*, i.e., classes whose instances have components –called *modules*– that are instances of other classes.

Items in the interface of the whole structured class and of the composing modules may be *connected* to denote an identity or equivalence that, for instance, may abstractly describe an information flow. A structured class may also include axioms, called global axioms, involving its own items and the interface items of its modules. Since the items of the component modules satisfy the axioms of their original class, the overall semantics of a structured class is given by the conjunction of the axioms of the class with those of the class of the component modules (if several instances of a class are included as modules of a structured class, then the axioms of the composing class are "duplicated" i.e., applied to duplicates of the class items (see [6]). From a strictly semantic viewpoint a structured class is thus equivalent to a TRIO formula obtained by flattening the modular structure and conjoining the axioms of the various modules. Figure 2 shows a structured class *KRC (Kernel Railroad Crossing),* describing a TRIO version of the standard railroad crossing problem [3], with three modules specifying the *trainModel,* the *controller* and the *gate*. More details on this example will be provided in the presentation of the case study illustrated in Section 4.

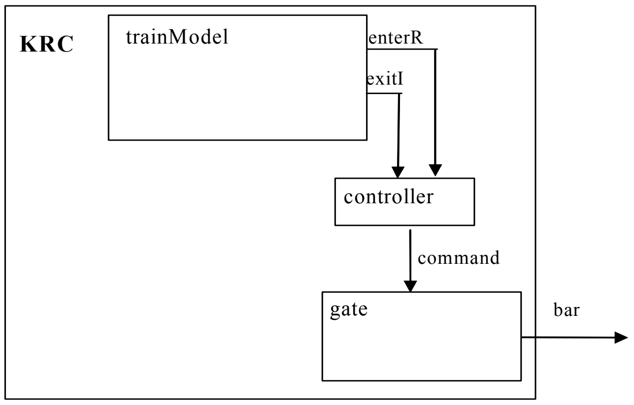

Fig. 1. An example of a TRIO structured class.

3. Translation

A TRIO specification can be translated into Promela, by introducing a network of communicating processes (*process network*), that may exchange truth values. When the process representing the whole specification returns false, the execution of the program is blocked. The translation is conceptually based on Alternating Automata and is presented in this section.

2-way Alternating Modulo Counting Automata (2AMCA)

We introduce an intermediate notation useful to define our TRIO-to-Promela translation, namely 2-way Alternating Modulo Counting Automata (or *2AMCA* for short). Conceptually, a 2AMCA is a version of Büchi alternating automata (see for instance [12,4]). A brief and intuitive description of alternating automata is the following. In a *deterministic* automaton, the transition function maps a *⟨state, input symbol⟩* pair to a *single* state, called the next state. The automaton accepts its input if either *state* is final and the input is finished, or from the next state the remaining suffix of the input word is accepted. On the other hand, in a nondeterministic automaton a ⟨state, input symbol⟩ pair is mapped to a *set* of states. Here we have two possible different interpretations of the transition function: either as an existential branching mode, or as a universal branching mode. In the existential mode, which is the standard interpretation of nondeterminism, the automaton accepts if at least *one* of the states of the set accepts the remaining input suffix; in the universal mode, it accepts if *all* the states of the set accept the remaining input suffix. An alternating automaton provides both existential and universal branching modes. Its transition function maps a ⟨state, input symbol⟩ pair into a (positive) boolean combination of states. Quite naturally, $\wedge$ is used to denote universality, while $\vee$ denotes existentiality. Alternating automata are a very convenient tool since they may be exponentially more concise than nondeterministic automata and are very well suited for dealing with logic formulae.

To define properly TRIO's metric temporal operators, we use internal finite counters, associated with states. Moreover, we use *bidirectionality* for defining the past tense operators, following an approach presented in [5,9].

Here are some preliminary definitions, following standard terminology (e.g., [19]). Let $\mathbb{N}$ be the set of natural numbers, and let $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$. A *tree* is a set $T \subseteq \mathbb{N}^*$ such that $x.c \in T \Rightarrow x \in T$ (c is called a *children* of x). The empty word ε is called the *root* of T. Elements of T are called *nodes*. A node is a *leaf* if it has no children. A *path* P of a tree T is a set $P \subseteq T$ which contains ε and such that for every $x \in P$, either x is a leaf or there exists a unique c such that $x.c \in P$.

An infinite word over Σ is a sequence $w = a_0\, a_1\, a_2\, ...$, with $a_j \in \Sigma$. We will indicate an element a_j of w as w(j). Moreover, we will denote the set of all infinite word over Σ as Σ^ω.

A *2-way Alternating Modulo Counting Automata* (2AMCA) is a six-tuple $A = (\Sigma,Q,\mu,q_0,\tau,F)$, where Σ is the (finite) *alphabet*, Q is the set of *states*, μ is a positive integer, $q_0 \in Q$ is the *initial state*, τ is the *transition function*, $F \subseteq Q$ is the set of *final states*. Call Cnt = $[0..\mu]$ the *counter set*. The transition function is τ: $Q \times Cnt \times \Sigma \rightarrow B^+(\{-1,+1\} \times Q \times Cnt)$, where, for every M, $B^+(M)$ indicates a positive boolean combination of elements in M, i.e. a boolean combination using $\wedge$ and $\vee$ but not using $\neg$. The set $\{+1,-1\}$ denotes the possible relative movements of the reading head. To improve readability, we will use the symbol '/' to separate Q from Cnt, and we will use +q or -q to denote (+1,q), (-1,q), respectively.

Consider a word $w \in \Sigma^\omega$. A *run* of A on w is a $Q \times Cnt \times \mathbb{N}$-labeled tree (T, ρ), where ρ is the labeling function, such that $\rho(\varepsilon) = (q_0/0,0)$ and for all $x \in T$, with $\rho(x) = (q/k,n)$, the set $\{(q'/h,d) \mid c \in \mathbb{N}, x.c \in T, h \in Cnt, d \in \{-1,+1\}, \rho(x.c) = (q'/h,n+d)\}$ satisfies the formula $\tau(q/k,w(n))$.

For a path P, $Inf(\rho, P) := \{s \mid$ there are infinitely many $x \in P$ with $\rho(x) \in \{s\}\times\mathbb{N}\}$. A run (T, ρ) of a 2AMCA is accepting if all infinite path P in T have $Inf(\rho,P) \cap F \neq \emptyset$.

From TRIO to 2AMCA

The translation of TRIO formulae into their equivalent 2AMCA follows the approach presented in [10].

Let *Ap* be a finite set of atomic propositions, and let φ be a TRIO formula on Ap and *Sf(φ)* be the set of subformulae of φ.

The 2AMCA automaton for φ is $A_\varphi = (\Sigma,Q,\mu,q_0,\tau,F)$ where:

$\Sigma = \wp(Ap)$, $Q = \{\phi \mid \phi \in Sf(\varphi)$ or $\neg\phi \in Sf(\varphi)\}$, $q_0 = \varphi$,

μ is the greatest bounded temporal distance occurring in φ, and

$F = \{\phi \mid \phi \in Q$ and $\phi \mid \phi$ has the form $\neg$Until(A,B) $\}$

The *dual operation* dual(ϕ) is defined for every formula ϕ as the formula ϕ' obtained from ϕ, by switching true and false, $\vee$, $\wedge$, and by complementing all subformulae of ϕ.

The transition function is defined as follows:

$\tau(C/0, a) = +true/0$ for $C\in Ap$ and $C\in a$

$\tau(C/0, a) = +false/0$ for $C\in Ap$ and $C\notin a$

$\tau(A\wedge B/0, a) = \tau(A/0, a) \wedge \tau(B/0, a)$

$\tau(\neg A/0, a) = dual(\tau(A/0, a))$

$\tau(Futr(A,n)/n, a) = \tau(A/0, a)$

$\tau(Futr(A,n)/k, a) = +Futr(A,n)/k+1$, where $0 \leq k < n$

$\tau(Past(A,n)/n, a) = \tau(A/0, a)$

$\tau(Past(A,n)/k, a) = -Past(A,n)/k+1$, where $0 \leq k < n$

$\tau(Lasts(A,n)/n-1, a) = true/0$

$\tau(\text{Lasts}(A,n)/k, a) = A/0 \wedge +\text{Lasts}(A,n)/k{+}1$, where $0 \leq k < n-1$

$\tau(\text{Lasted}(A,n)/n{-}1, a) = \text{true}/0$

$\tau(\text{Lasted}(A,n)/k, a) = A/0 \wedge -\text{Lasted}(A,n)/k{+}1$, where $0 \leq k < n-1$

$\tau(\text{Until}(A,B)/0, a) = \tau(B/0, a) \vee (\tau(A/0, a) \wedge +\text{Until}(A,B)/0)$

$\tau(\text{Since}(A,B)/0, a) = \tau(B/0, a) \vee (\tau(A/0, a) \wedge -\text{Since}(A,B)/0)$

The transition function is undefined for every case not listed above.

From 2AMCA to Promela

The outcome of the previous step is a 2AMCA equivalent to the original TRIO specification. The expressive richness of the Promela language makes the 2AMCA-to-Promela translation a simple task, if we do not take into account optimizations. Indeed, we use the Promela code to directly simulate the alternating automaton equivalent to the original TRIO specification.

Conceptually, every state of the automaton will correspond to a single type of process (*proctype*). As in classical nondeterministic automata, an or-combination of states ($s_1 \vee s_2$) in the transition function will correspond to a nondeterministic choice (*if ::s_1; ::s_2; fi*). Analogously, an and-combination $s_1 \wedge s_2$ will correspond to the starting of two new processes, having type s_1 and s_2, respectively.

As far as process synchronization is concerned, we have to proceed bottom-up: quite naturally, processes corresponding to simpler subformulae must be evaluated before more complex ones. The system does not require asynchronous communication among processes. In fact, it is possible to determine an arbitrary total evaluation order, starting from the original partial order defined by the relation "subformula of". Therefore, we used a single rendezvous channel.

Bounded temporal operators (Futr, Lasts and WithinF) use simple counting loops and variables to determine where to start and stop evaluating, and to store partial evaluations.

As an example, consider the formula: *AlwF(push $\rightarrow$ Lasts(on, 7))*.

The non-optimized Promela code contains two process types, one for *AlwF* and one for *Lasts*. In general, it is not necessary to define multiple process types for boolean operators applied to atomic propositions. For instance, in our example the implication '*push $\rightarrow$*' can be handled directly within the AlwF process.

```
#define MAXP 6 /* maximum number of launched Lasts processes */

proctype AlwF(chan environment; chan sync) {
bool push,on;
byte n;
bool ex[MAXP], dying, result;
chan to_lasts = [0] of {bool, byte};
chan from_lasts = [0] of {bool, bool, byte};
do
:: environment?push,on;
          n = 0;
          do
          :: n < MAXP ->
            if
            :: ex[n] -> to_lasts!on,n;
              from_lasts?dying,result,eval(n);
              if
              :: dying -> ex[n] = 0;
              :: else;
              fi;
              if
```

```
                  :: !result -> sync!0; goto stop; /* error */
                  :: else;
                  fi;
                :: else;
                fi;
                n++;
              :: n == MAXP -> break;
              od;
              if
              :: !push -> sync!1;
              :: push -> n = 0;
                do
                :: n < MAXP ->
                  if
                  :: !ex[n] -> break;
                  :: else -> n++;
                  fi
                :: n == MAXP -> sync!0; goto stop; /* overflow */
                od;
                ex[n] = 1;
                run Lasts(to_lasts,from_lasts,MAXP,n);
                sync!1;
              fi;
od;
stop: skip;
}

proctype Lasts(chan from_alw; chan to_alw; byte k; byte id) {
bool on;
do
:: from_alw?on,eval(id);
              if
              :: on && k == 1 -> to_alw!1,1,id; break;
              :: on && k > 1 -> to_alw!0,1,id; k--;
              :: !on -> to_alw!1,0,id; break;
              fi;
od;
}
```

In this case, the AlwF process may launch at most six different instances of the Lasts process, since the boolean argument of Lasts must be checked in six different instants. This bound is dealt with by the constant definition of MAXP in the very first line of the Promela code

In the previous piece of code, we use two channels to manage the communication between the AlwF process and its children. First, AlwF sends to every alive instance of its children the value of *on* coming from the environment, then it reads the results of their evaluation. A Lasts process may send two boolean signals to AlwF: the first is about its immediate termination, while the second is the result of its evaluation. Both the AlwF process and the Lasts processes use an identifier (*n* and *id*, respectively) for synchronization purposes.

Past operators are treated a bit differently. The actual Promela code does not directly implement a back movement of the reading head of the automaton. On the contrary, it follows some of the ideas presented in [9,8] for obtaining a 1-way automaton from a 2-way one, specifically tailored and optimized for the TRIO language. For instance, bounded past operators, instead of "going back in time" and evaluating subformulae, use arrays to store a bounded amount of previous subformula evaluations, so that they may directly access to them. Unbounded operators are implemented by means of processes which check, and properly store, past subformula evaluations.

A first comparison with LTL2BA

LTL2BA (*LTL to Büchi Automata*) [2] is a tool that translates LTL formulae into Promela *never claims*. The formula of the previous example, *AlwF(push* $\rightarrow$ *Lasts(on, 7))*, can be written in LTL as follows: G(push $\rightarrow$ X(on $\wedge$ X(on $\wedge$ X(on $\wedge$ X(on $\wedge$ X(on $\wedge$ X(on))))))). LTL2BA uses alternating automata as an intermediate notation - the resulting Promela code is a direct representation of the equivalent (and simplified) Büchi automaton. In our example, we obtain the following code:

```
never {
accept_init:
          if
          :: (!push) -> goto accept_init
          :: (1) -> goto accept_S2
          fi;
accept_S2:
          if
          :: (!push && on) -> goto accept_S9
          :: (on) -> goto accept_S2
          fi;
accept_S9:
          if
          :: (!push && on) -> goto accept_S17
          :: (on) -> goto accept_S2
          fi;
accept_S17:
          if
          :: (!push && on) -> goto accept_S29
          :: (on) -> goto accept_S2
          fi;
accept_S29:
          if
          :: (!push && on) -> goto accept_S31
          :: (on) -> goto accept_S2
          fi;
accept_S31:
          if
          :: (!push && on) -> goto accept_S33
          :: (on) -> goto accept_S2
          fi;
accept_S33:
          if
          :: (!push && on) -> goto accept_init
          :: (on) -> goto accept_S2
          fi;
}
```

The major difference is that LTL does not support metric operators; therefore both the formula and the resulting code size depend on the constants (only 7 in this case). With our technique, the Promela code for *AlwF(push $\rightarrow$ Lasts(on, k))*, for any given k, is not dependant on *k*, apart from the definition of MAXP. Hence, in general the size of the code does not depend on the values of the temporal constants, making the translation very concise. In this case, the size depends linearly on the size of the formula, because all the occurrences of bounded temporal operators (one in this case, namely Lasts) are not nested. Other techniques, such as LTL2BA, even though based on sophisticated optimizations to reduce the size of the resulting code, always enumerate explicitly all states of the Büchi automaton equivalent to the intermediate alternating automaton, which may have up to $n \cdot 2^n$ states, where n is the number of states of the alternating automaton (see [2] for details). Of course, SPIN is an exhaustive model checker, enumerating all reachable states, and the constants like MAXP do increase the state space. However, applying a translation process like LTL2BA to TRIO specifications may not be able to generate Promela code short enough to attempt verification, even in those cases, as shown below, where verification is actually feasible.

4. Verification

In traditional model checking, a property (e.g., a LTL formula) is verified against a model of the system (an automaton such as a Promela program). When translating a whole TRIO specification in order to check its satisfiability, however, no automaton model is already present. As a result, a special automaton, called a *generator*, is introduced and added to the process network. The generator exhaustively produces random input values, then it sends them to the Promela program; hence, the generator is able to generate any system behavior. These behaviors are verified by the process network. An event generator allows model checking of resulting Promela code, corresponding to satisfiability verification of the original TRIO specification. The event generator may however increase the number of reachable state of the resulting system. As shown below, modularity in the specification may be used to introduce *modular generators*, significantly reducing this increase.

The Promela representation of a 2AMCA may however be further optimized in order to obtain compact and easily verifiable code. Some optimizations are:

- Each occurrence of a bounded operator Lasts is translated using only one process that is updated whenever the temporal subformula has to be checked.
- Bounded operators nested in a Futr or Past operators are translated by shifting the starting point of the same constant amount used in the Futr (or Past) operator.
- A coordinator process is in charge of managing communication, to reduce the synchronization effort required by the potentially high number of processes produced by the translation of all subformulae.
- An additional component of each process manages error propagation to terminate immediately the whole process network when the specification is violated. More precisely, when a flag denoting an unacceptable behavior is activated, an error is propagated to kill each currently alive process. This actually reduces the number of reachable states, since in this way there is only one "error" state.

A case study

As a case study, the approach was applied to the KRC specification shown in Figure 1. The first version of the case study is not modularized, even though the original specification was, and later will be clear how the axioms are distributed among the modules. A "flat list" of the axioms is the following:

(K1) $train = EnterR \rightarrow Lasts(\neg train = EnterR, \mu)$

(K2) $train = EnterI \rightarrow Lasts(\neg train = EnterI, \mu)$

(K3) $train = ExitI \rightarrow Lasts(\neg train = ExitI, \mu)$

(K4) $train = EnterR \rightarrow Futr(WithinF_{ii}(train = EnterI,\ d_M - d_m),\ d_m)$

(K5) $train = EnterI \rightarrow Futr(WithinF_{ii}(train = ExitI,\ h_M - h_m),\ h_m)$

(K6) $train = EnterI \rightarrow Past(WithinP_{ii}(train = EnterR,\ d_M - d_m),\ d_m)$

(K7) $train = ExitI \rightarrow Past(WithinP_{ii}(train = EnterI,\ h_M - h_m),\ h_m)$

(K8) $d_M \geq d_m > 0 \wedge h_M \geq h_m > 0 \wedge \mu > d_M + h_M \wedge d_m > \gamma$

(S1) $InR \leftrightarrow WithinP_{ii}(train = EnterR, d_M) \wedge Since_{ii}(\neg train = EnterI,\ train = EnterR)$

(S2) $\mathrm{InI} \leftrightarrow \mathrm{WithinP}_{ii}(\mathrm{train} = \mathrm{EnterI}, h_M) \wedge \mathrm{Since}_{ii}(\neg \mathrm{train} = \mathrm{ExitI}, \mathrm{train} = \mathrm{EnterI})$

(M1) $\mathrm{UpToNow}(\mathrm{bar} = \mathrm{closed}) \wedge \mathrm{command} = \mathrm{goUp} \rightarrow$

$\mathrm{Lasts}_{ie}(\mathrm{bar} = \mathrm{mvUp}, \gamma) \wedge \mathrm{Futr}(\mathrm{UntilW}_{ie}(\mathrm{bar} = \mathrm{open}, \mathrm{command} = \mathrm{goDown}), \gamma)$

(M2) $\mathrm{UpToNow}(\mathrm{bar} = \mathrm{open}) \wedge \mathrm{command} = \mathrm{goDown} \rightarrow$

$\mathrm{Lasts}_{ie}(\mathrm{bar} = \mathrm{mvDown}, \gamma) \wedge \mathrm{Futr}(\mathrm{UntilW}_{ie}(\mathrm{bar} = \mathrm{closed}, \mathrm{command} = \mathrm{goUp}), \gamma)$

(M3) $\mathrm{AlwP}_i(\neg \mathrm{command} = \mathrm{go(down)}) \rightarrow \mathrm{bar} = \mathrm{open}$

(C1) $\mathrm{command} = \mathrm{goDown} \leftrightarrow \mathrm{Past}(\mathrm{train} = \mathrm{EnterR}, d_m - \gamma)$

(C2) $\mathrm{command} = \mathrm{goUp} \leftrightarrow \mathrm{train} = \mathrm{ExitI}$

The goal of the original KRC specification in TRIO [20] was twofold: a formal definition of the KRC system, and the proof of the safety property that, whenever the train is inside the railway crossing, the bar is always down. Notice that KRC is a toy example per se, but in this case we are completely defining it with a temporal logic specification, thus obtaining a logic formula much bigger and more complex than those used in traditional model checking, where the KRC is defined with an automaton and short temporal logic formulae are used only for safety or utility properties.

We encoded the possible values of the *train* variable as: 1 (EnterR), 2 (EnterI), 3 (ExitI), and 0 for all the other situations. Likewise, *bar* may assume values: 0 (open), 1 (closed), 2 (mvUp), 3 (mvDown). *command* may be: 0 (no indications), 1 (goUp), 2 (goDown). Moreover variables train, command, bar may take the additional value 4 to signal an erroneous configuration.

For this unmodularized version of the specification, a unique generator for all events is defined, by means of the following Promela code:

```
proctype EventGenerator(chan in; chan out){
bool sync;
do
          ::in?sync,eval(1);
            if
            ::s==0 ->
               out!4,0,0,4,4,2;       /* train,inR,inI,go,st to  */
               break;                 /* Process 2 (Coordinator) */
            ::else;
            fi;
            if
            ::train=0;
            ::train=1;
            ::train=2;
            ::train=3;
            fi;
            if
            ::inR=0; inI=0;
            ::inR=1; inI=0;    /* inR, inI are mutually exclusive */
            ::inR=0; inI=1;
            fi;
            if
            ::command=0;
            ::command=1;
            ::command=2;
            fi;
            if
            ::bar=0;
            ::bar=1;
            ::bar=2;
            ::bar=3;
            fi;
          out!train,inR,inI,go,st,2;
od
}
```

As already explained, events are nondeterministically generated, then sent to the coordinator through the channel *out*. First, the generator waits for the synchronization signal *sync* from the coordinator. When a process signals an error to the coordinator, *sync* takes the value 0, halting the network (*train*, *command*, *bar* take the error value 4).

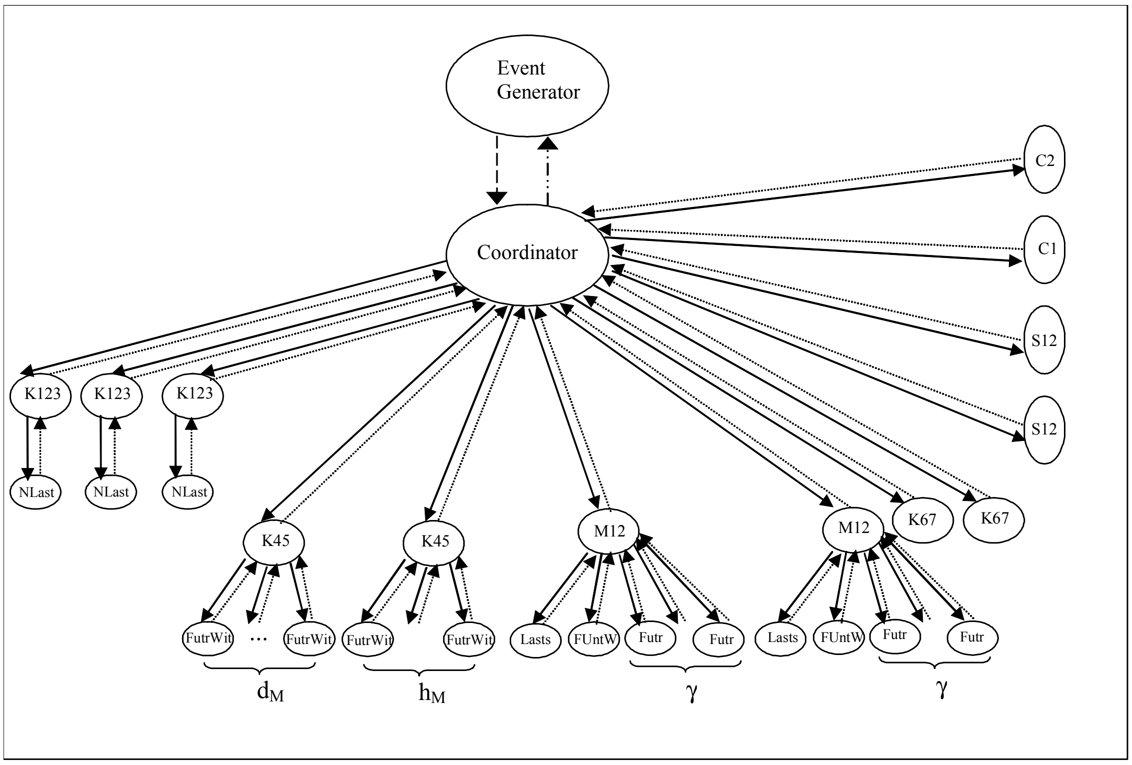

Fig. 2. The process network (nonmodular case)

The resulting process network for the KRC, depicted in Figure 2, is composed by the following processes:

- the *coordinator*, which manages synchronization and checks validity of axiom M3,
- one event *generator*,
- 2 process instances each for K1, K2 and K3 (those based upon the Lasts operator), having process type K123 and NLast (a process type that defines Lasts(¬A,k), for some A and k),
- d_M-d_m and h_M-h_m process instances for K4 and K5 respectively (bounded eventualities), having process type K45 and FutrWit (composition of Futr and WithinF),
- a single process type (K67, S12) and two process instances each for the comparatively simpler K6, K7 and S1, S2 axioms, respectively;
- four process types each for M1 and M2 (M12, Lasts, FUntW, Futr), one instance for each of the first three types, and γ instances of the last.

Notice that in the picture we used different styles for different channels. For instance, signals are sent by coordinator to axiom processes through a single channel, denoted by a solid arrow.

Unfortunately, state explosion may be caused by both the exhaustive case enumeration carried on by the event generator, and by the high number of processes. For instance, consider Table 2, which contains a summary of our verification results (we used a PC equipped with a Pentium 4 processor @ 2GHz, 256 MB of RAM, and every computation took less than 2 minutes). When $\mu = 30$, values of h_M higher than 14 are not tractable (i.e., they cause a memory overflow).

We also ran LTL2BA and Wring on the same specification with $\mu = 7$, $h_M = d_M = 3$, $h_m = d_m = 2$, and $\gamma = 1$. LTL2BA crashed after a memory overflow, while Wring was still running after three days and was therefore aborted.

μ	d_M	d_m	h_M	h_m	γ	Depth	Memory (KByte)	States	Transitions
10	5	4	4	3	2	4813	40644	584216	585504
15	7	4	7	3	2	6739	69419	972878	974738
20	7	5	12	5	3	8009	84369	1086440	1088150
20	10	8	9	5	3	9909	104747	1359000	1361140
25	15	12	9	7	6	15815	203267	2335110	2338110
25	15	12	9	7	9	> 210 MB			
30	15	12	14	12	3	16525	204367	2375400	2378400

Table 2. Verification - nonmodular case

MODULAR APPROACH

The results presented above clearly show that our approach is not well suited to large specifications. Hence, we decided to exploit the modular structure of a TRIO specification, computing, rather than randomly generating, some of the events.

First, we have to partition the set of atomic propositions into three subsets: *input*, *output*, and *state* propositions. After that, event generators are associated only with modules that directly deal with input predicates. On the other hand, output variables tend to be, by their very nature, deterministic, and therefore the TRIO modules using them do not need a generator.

To show how this approach works, consider the KRC case study, which consists of three modules: the first module (*trainModel*) describes the train position with respect to the critical regions R and I (axioms K1, K2, K3, K4, K5, K6 and K7); the second module (*globalAxioms*) contains the definition of *inR* and *inI* (axioms S1 and S2); the third module (*controller*) contains the bar control logic (C1 and C2); the last module (*gate*) is used to define the position of the bar (axioms M1 and M2).

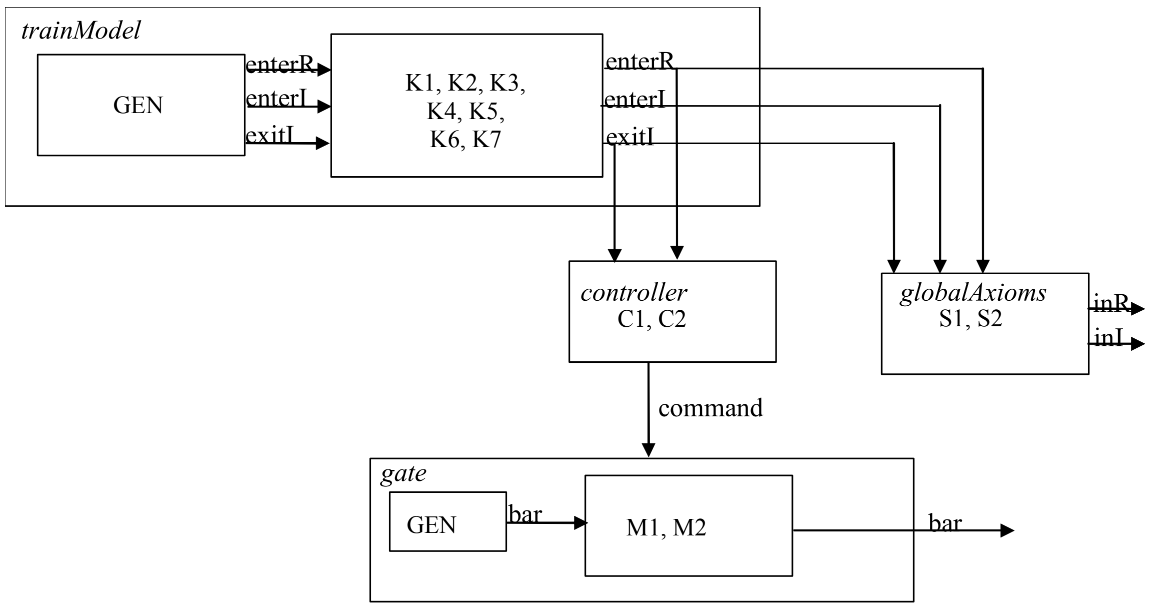

Fig. 3. Modular structure of KRC

In this case it is quite easy to identify the inputs: *enterR*, *enterI*, *exitI* (i.e., the variable *train*), and *bar*; while *inR*, *inI*, *command* and *bar* may be computed. In fact, we can identify *inR* and *inI* as state variables, defined by axioms S1 and S2 respectively. On the other hand, *command* is an output variable, defined by axioms C1 and C2.

As an example, let us consider the code that corresponds to C1, C2:

```
proctype CommandGenerator(chan ev; chan sync){ /* axioms C1 and C2 */
byte t,n;
byte memo[Dm]; /* store the last significant values of train */
command=0;
do
           ::ev?t,eval(3); /* receive 'train' from Process 3 */
           n=Dm-G;
           do
             ::n>0->memo[n]=memo[n-1];n--;
             ::n==0 -> memo[n]=t; break;
           od;
           if
             ::t==4 -> ev!4,5; goto stop;
             ::else ->
               if
               ::memo[Dm-G]==1 -> command=2;
               ::t==3 -> command=1;
               ::else -> command=0;
               fi;
               ev!command,5; /* send command to Process 5 */
           fi;
od;
stop:      skip;
}
```

The complete modular structure is presented in Figure 3. Compared to the previous case, the number of processes is smaller, because axioms that are only used to compute variable values can always be translated into a single process. Moreover, we need one generator and one coordinator for each one of the two modules *trainModel* and *gate*.

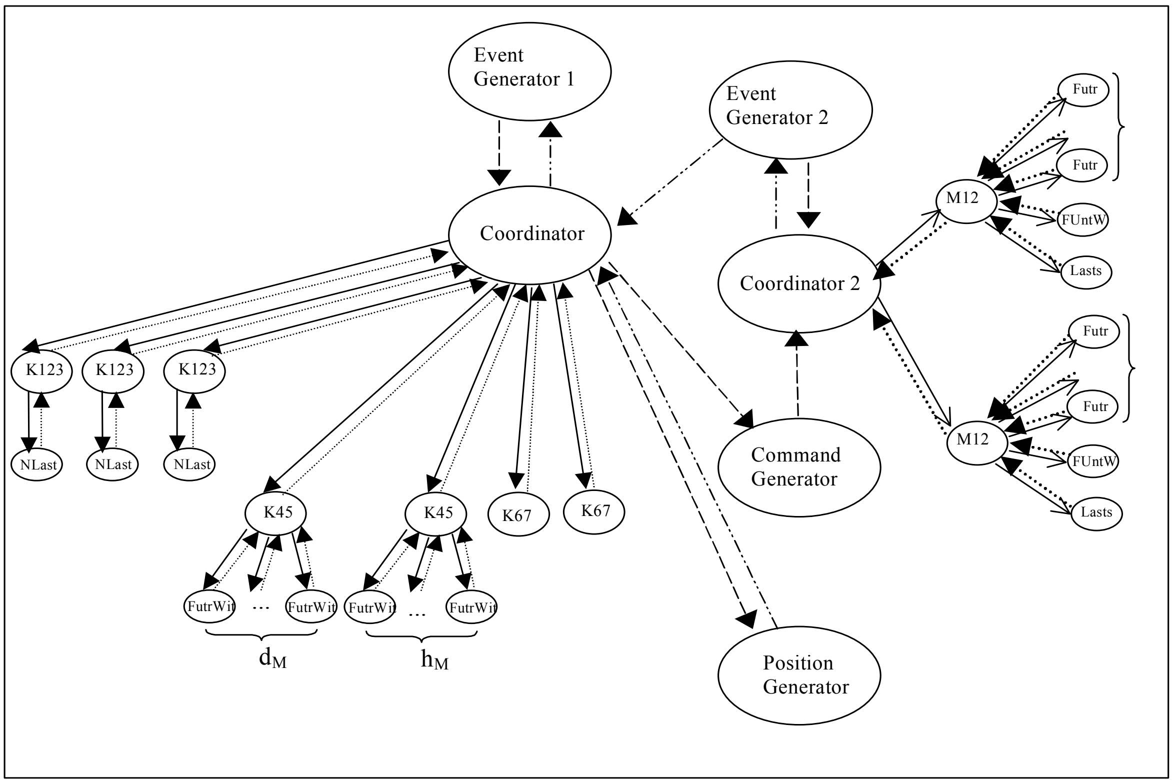

Fig. 4. The process network (modular case)

Verification results, with the same system configuration for the results of Table 2, are shown in Table 3. Performance increases noticeably for the same constant values: used memory is more than halved, and state number considerably decreased (running times are not reported, but always below two minutes).

μ	d_M	d_m	h_M	h_m	γ	Depth	Memory (KB)	States	Transitions
10	5	4	4	3	2	6791	12680	162727	164944
15	7	4	7	3	2	10606	39928	526592	531193
20	7	5	12	5	3	15503	59998	742398	747850
20	10	8	9	5	3	15371	48120	580929	585481
25	15	12	9	7	6	21482	84369	923474	928826
25	15	12	9	7	9	22532	105259	1097340	1102690
30	15	12	14	12	3	26517	94712	1043610	1050300
30	15	12	14	12	9	28917	150520	1500430	1507120
30	20	17	9	7	12	30127	153297	1433200	1438990
30	20	13	9	7	3	26343	130961	1462620	1471300
30	9	7	20	17	3	27001	92459	1025200	1031770
40	20	17	19	17	6	42397	191167	1826740	1835220
40	20	17	19	17	9	> 210 MB			
40	30	27	9	7	3	41117	115597	1138780	1145450
40	30	27	9	7	9	44117	175057	1569050	1575720

Table 3. Verification - modular case

5. Conclusions and directions of future research

We presented a novel application of model checking through SPIN as a means for verifying purely descriptive specifications written in TRIO. The approach is based on the translation of TRIO formulae into Promela programs guided by an equivalence between TRIO and 2-way alternating modulo counting automata. The set of TRIO axioms is partitioned into the *specification* part, i.e., axioms describing assumptions on the systems being checked, and the *property* part, that must be proven to hold under such assumptions; then the SPIN model checker is employed to prove the validity of the implication *specification→property*. Since a TRIO specification does not include any operational model that accounts for the behavior to be checked against the property, an additional Promela component must be combined with the result of the translation of TRIO specification, to generate the values for variables occurring in the TRIO specification. This generative component of the Promela programs is the major source of complexity in the verification process, which is addressed at a linguistic level by exploiting the modular structure of TRIO specifications, and at an applicative/methodological level by limiting the combinatorial generation to the logical variables that are completely independent of any other value.

The results of our experimentation are quite encouraging, as we were able to verify properties of the Railway Crossing Problem for values of the temporal constants in the TRIO axioms that make the verification totally unfeasible with tools such as LTL2BA or Wring.

The translation of TRIO formulae into Promela programs was performed manually for the case study, but we are confident that it can be easily automated. This is apparent for what concerns the translation of the various TRIO constructs relying on their correspondence with 2-way alternating modulo counting automata; the optimizations based on the modular structure can also be automated if the TRIO specification is provided in modular form, while those based on restricting generation to input variables can be applied mechanically by taking into account the direction of the arrows representing items in the interface of specification modules (in any case the information on which are the input to the system under study is rather clear starting from the early phases of requirements analysis).

We have defined a verification procedure in a scenario where both the system under design and the desired property to be proven are expressed as TRIO axioms. *A fortiori*, our approach can be applied in the case when the system under design is described by means of an operational model such as a Promela program or any state-transition system that can be translated into Promela. In fact, in this case the verification procedure would be more efficient, as the principal source of complexity, namely the combinatorial generation of values for logical variables, would be avoided. From this viewpoint, our approach is more effective than other methods that construct a never claim from an LTL specification, because the Promela code produced from the TRIO formulae grows in size linearly with the size of the TRIO formula, while other approaches suffer from an "exponential blow up" of the state space: for instance, LTL2BA produces Promela code that coincides, in fact, with the Büchi automaton. Under this respect, we can therefore claim that our approach better exploits the potential of SPIN and Promela for verification.

Another application of the ideas presented here, that conceptually is a mere by-product of the verification method but has a relevant potential for construction of verification tools, is the generation of execution traces and of functional test cases starting from a purely descriptive specification given in

TRIO. We expect the computational complexity of generating a simulation to be orders of magnitude smaller than the one of property proving, thus permitting the implementation of industrial-strength tools supporting validation of specifications and specification-based functional testing.

6. References

1. E. Allen Emerson. Temporal and Modal Logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics 1990*, J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, Pages 995-1072.
2. P. Gastin, D. Oddoux, *Fast LTL to Büchi Automata Translation*, Proceedings of CAV'01, Lecture Notes in Computer Science 2102, p. 53-65, 2001.
3. C. Heitmeyer and D. Mandrioli, editors. *Formal Methods for Real-Time Computing*, volume 5 of Trends in Software. Wiley, 1996.
4. O. Kupferman, M. Vardi, *Weak Alternating Automata Are Not That Weak*, Proceedings of the Fifth Israel Symposium on Theory of Computing and Systems, ISTCS'97, 1997
5. O. Kupferman, N. Piterman, M. Vardi, *Extended Temporal Logic Revisited*, CONCUR'01, 2001.
6. A. Morzenti, P. San Pietro, *Object-Oriented Logic Specifications of Time Critical Systems*, ACM Trans. on Softw. Engin. and Meth., vol.3, n.1, Jan.1994, pp. 56-98.
7. A. Morzenti, D. Mandrioli, C. Ghezzi, *A Model Parametric Real-Time Logic*, ACM Trans. on Programming Languages and Systems 14, 4 (October 1992), 521-573.
8. N. Piterman, M. Vardi, *From Bidirectionality to Alternation*, MFCS'01, 2001.
9. M. Vardi, *A Temporal Fixpoint Calculus*, POPL'88, 1988.
10. M. Vardi, *An automata-theoretic approach to linear temporal logic*, Banff'94, 1994.
11. F. Somenzi and R. Bloem, *Efficient Büchi automata from LTL Formulae*, CAV'00, pp.248-263, 2000
12. A. Chandra, D. Kozen, and L. Stockmeyer, Alternation. *Journal of the Association for Computing Machinery 28*, 1 (January 1981), 114-133.
13. C.Ghezzi, D.Mandrioli, A.Morzenti, *TRIO, a logic language for executable specifications of real time systems*, The Journal of Systems and Software, Elsevier Science Publishing, vol.12, no.2, pp. 107-123, May 1990.
14. M.Felder, A.Morzenti, *Validating real-time systems by history-checking TRIO specifications*, ACM TOSEM-Transactions On Software Engineering and Methodologies, vol.3, n.4, October 1994.
15. D.Mandrioli, S.Morasca, A.Morzenti, *Generating Test Cases for Real-Time Systems from Logic Specifications*, ACM TOCS-Transactions On Computer Systems, Vol. 13, No. 4, November 1995. pp.365-398.
16. A. Gargantini, A.Morzenti, *Automated Deductive Requirements Analysis of Critical Systems*, ACM TOSEM - Transactions On Software Engineering and Methodologies, Vol. 10, no. 3, July 2001, pp. 225-307.
17. FAST-ESPRIT Project No. 25581 – *Synthesis of the evaluation of the FAST toolset Experimentation*, FAST Report D7.5.1, The FAST Consortium, November 2000.
18. G. Holzmann, *The Model Checker SPIN*, IEEE Transactions on Software Engineering, Vol. 23, 5, May 1997.
19. Wolfgang Thomas: *Automata Theory on Trees and Partial Orders*. TAPSOFT, 1997.
20. Mandrioli D., Morzenti A., Pezzè M., San Pietro P. Silva S., *A Petri Net and Logic Approach to the Specification and Verification of Real Time Systems*, in [3].