The Symmetry of the Past and of the Future: Bi-infinite Time in the Verification of Temporal Properties

Matteo Pradella
IEIIT
Consiglio Nazionale delle Ricerche
v. Ponzio 34/5
Milano, Italy
pradella@elet.polimi.it

Angelo Morzenti
Dipartimento di Elettronica e Informazione
Politecnico di Milano
v. Ponzio 34/5
Milano, Italy
morzenti@elet.polimi.it

Pierluigi San Pietro*
Dipartimento di Elettronica e Informazione
Politecnico di Milano
v. Ponzio 34/5
Milano, Italy
sanpietr@elet.polimi.it

ABSTRACT

Model checking techniques have traditionally dealt with temporal logic languages and automata interpreted over ω-words, i.e., infinite in the future but finite in the past. However, time with also an infinite past is a useful abstraction in specification. It allows one to ignore the complexity of system initialization in much the same way as system termination may be abstracted away by allowing an infinite future. One can then write specifications that are simpler and more easily understandable, because they do not include the description of the operations (such as configuration or installation) typically performed at system deployment time. The present paper is centered on the problem of *satisfiability* checking of linear temporal logic (LTL) formulae with past operators. We show that bounded model checking techniques can be adapted to deal with bi-infinite time in temporal logic, without incurring in any performance loss. Our claims are supported by a tool, whose application to a case study shows that satisfiability checking may be feasible also on nontrivial examples of temporal logic specifications.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*

General Terms

Theory, Verification

Keywords

Temporal Logic, Bounded model checking, Bi-infinite time, Satisfiability checking

*The research of Pierluigi San Pietro has been supported in part by MUR grants FIRB RBAU01MCAC, PRIN 2005015419-002.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.

1. INTRODUCTION

Linear-time Temporal Logic (LTL), also in its version augmented with past operators (PLTL: past LTL), has been extensively applied to the specification, validation and verification of critical, real-time systems [34, 26, 12]. Although its expressive power has some limitations [40], its success resides also in the availability of powerful model checking engines, which allowed the task of verification or debugging of real-life systems (see, e.g., [11, 9, 22]). Given a system model M, defined using a finite state-transition system, model checking is the verification whether a formula ϕ, describing an (un)desired property of M, holds. Following the automata-theoretic approach [39, 21, 38, 17, 18], this is achieved by translating LTL formula $\neg\phi$ into a Büchi automaton, and then checking whether $L(M) \cap L(\neg\phi) = \emptyset$ holds, i.e., the intersection of the language of M and the language of $\neg\phi$ is empty. Hence, model checkers such as SPIN are basically emptiness checkers for Büchi automata. In the Symbolic Model Checking [27] approach (typically, but not exclusively, applied to Computation Tree Logic [10] rather than LTL), data structures such as Binary Decision Diagrams (BDD) [6] allow for a compact symbolic representation of M, often achieving better performance. More recently, in Bounded Model Checking (BMC) [4], given a bound $k > 0$, both formula ϕ and system M are translated into a formula $\Xi_{k,M,\phi}$ of boolean logic, which is satisfiable iff M has a counterexample of length k to property ϕ. Hence, the verification problem is reduced to checking the satisfiability of $\Xi_{k,M,\phi}$. Since SAT solvers, such as MiniSat [14] and Chaff [32], while solving an NP-hard problem, are often very efficient in practice, bounded model checking may be faster than automata-theoretic model checking or even BDD-based model checking in many practical cases. The main obstacle to the application of BMC is obviously the bound k itself, which makes BMC an incomplete procedure. In fact, if the resulting boolean formula is satisfiable, then we are assured that property ϕ (e.g., a safety or a liveness property) is falsified. If, however, the formula is reported to be unsatisfiable there is no assurance that the original property ϕ is verified on M, since there might only exist counterexamples to ϕ of length greater than k. This problem has been tackled for instance in [37, 5]: given M, ϕ and the bound k, another boolean formula $\Psi_{k,M,\phi}$ may be derived to check whether the bound k is enough to exhaust all possible system behaviors. If both $\Psi_{k,M,\phi}$ and $\Xi_{k,M,\phi}$ are unsatisfiable with bound k, then ϕ is verified on M. In the worst case, this value of k is exponential in the length of M and ϕ, but in practice this k may not be very large, especially if the system has a limited level of nondeterminism.

Throughout our past research, we have dealt heavily with temporal logic specifications and their application to industrial, criti-

cal real-time systems [31, 28, 8, 7]. In particular, our experience has been based on the TRIO language [19]. TRIO is a first order, linear-time temporal logic with both future and past operators and a quantitative metric on time, providing also structuring and modularization constructs. Our approach has focused on using TRIO for requirements specifications, without relying on machine models such as automata. 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), similarly to what occurs in any other purely descriptive specification notation. Hence, the problem of property proving in TRIO takes a form that is rather different from the model-checking approach, since both the system to be specified and the property to be proven are formalized as TRIO formulae. Property proving is therefore formulated in terms of the validity of a logic formula of the kind *specification* $\rightarrow$ *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.

Our verification tools have mainly been based on test case generation [36, 15, 25] and theorem proving [16], due to the general undecidability of TRIO language. More recently, we defined, for a decidable fragment of TRIO (semantically equivalent to PLTL), a tool for satisfiability checking via automata-theoretic model checking [30, 35, 3, 2]. Our results confirm the obvious expectation that satisfiability checking is harder than model checking of a system of the same size, but that it may still be feasible in interesting cases.

One of the features of TRIO that differentiates it from other temporal logic languages is its ability to deal with different time domains: dense or discrete, finite or infinite [29]. In particular, most TRIO specifications adopt a bi-infinite time domain, i.e. a time domain that is infinite both in the future and in the past.

Although philosophers such as Prior have always considered the case that time may be bi-infinite, most temporal logics and automata models used in specification and verification consider time to be finite in the past, even though Automata Theory has also considered bi-infinite computations [20, 33]. This has historical reasons, since both automata and temporal logic were applied to model programs, where there is actually an initialization step, and where logics with only future operators seemed adequate. This corresponds to the case where a system, such as a program, does not necessarily have a final state (i.e., it may not terminate). It has been widely argued that allowing time to be infinite in the future is very convenient when describing reactive systems and studying their properties (such as liveness and fairness), even though obviously all real systems have to terminate, sooner or later. Hence, nontermination is only an abstraction, useful to write specifications without explicitly considering the final disposal of the analyzed system. For instance, the controller of a railroad crossing may be considered as nonterminating, since one might simply not want to model explicitly the case when the controller is stopped for failures, maintenance or replacement. This allows one to write simpler formulae.

It is widely recognized that past operators allow one to write specifications that are easier, shorter, more intuitive and, in some significant cases even exponentially more succinct than LTL specifications [24]. But also the semantics of temporal logic languages with both past and future modalities (such as PLTL) is usually defined in terms of words which are infinite only in the future.

Here we argue that, analogously to the mono-infinite case where termination may be ignored, interpreting a PLTL specification on bi-infinite time is convenient to deal with system models where initialization may also be ignored. This may add another layer of abstraction, since one can write specifications that are simpler and more easily understandable, because they do not include the description of the operations (such as configuration, installation, ...) typically performed at system deployment time. For instance, for reactive systems embedded into devices that continuously monitor or control some process, initialization may often be ignored and one may focus only on routine behavior.

Using bi-infinite time with PLTL also helps in solving a technical problem, called the "border effect" [29, 13]. The problem arises in conjunction with bounded metric operators, such as the "previous time" $\bullet$ operator. Consider the following example.

EXAMPLE 1. A transmission line *Consider a simple transmission line, that receives messages at one end and delivers them at the opposite end with a fixed delay (e.g., 1"). The arrival of a message is represented by the propositional letter in, while its delivery is represented by out. The following PLTL formula expresses that every received message is delivered, and no spurious message is emitted (i.e., every out is always preceded by an in).*

$$\Box(out \rightarrow \bullet in) \wedge \Box(\neg out \rightarrow \bullet \neg in)$$

The meaning of the formula should be clear by recalling that $\Box$ is the globally operator ($\Box\phi$ requires ϕ to be true in every instant from now), $\circ$ is the next-time operator and $\bullet$ is the last-time (or Yesterday) operator.

Since time is finite "on the left", the evaluation of past operators may be problematic. The traditional solution is to return a "default" false value when the evaluation of a subformula is outside the time domain (the typical PLTL semantics of $\bullet\phi$ is $w, i \models \bullet\phi \Leftrightarrow i-1 \geq 0 \wedge w, i-1 \models \phi$, which is false if $i = 0$). This definition may easily lead to subtle specification errors. For instance, the above specification is unsatisfiable: at instant 0, both $\bullet\neg in$ and $\bullet in$ are false. Therefore, at instant 0, if out holds then the formula $out \rightarrow \bullet in$ is false; otherwise, if out does not hold at 0, then $\neg out \rightarrow \bullet\neg in$ is false. But if we rewrite $\bullet\neg in$ as $\neg\bullet in$, the original formula becomes satisfiable. This is because, in general $\neg \bullet \phi$ may have a different value from $\bullet\neg\phi$. This behavior may be somehow "fixed" by allowing two different forms of the $\bullet$ operators, the second one being defined to the default true value when its argument cannot be evaluated. This is clearly cumbersome and counterintuitive.

The simplest and most effective solution to the above described subtle semantic problems is to adopt bi-infinite time, where the past operators behave like the future operators: they are always defined. Notice that the usage of bi-infinite time does not rule out the explicit modeling of the initial state of a system, and hence it incurs in no loss of expressive power (e.g., just use a propositional symbol $start$, with the additional constraint that $start$ must occur at some time).

The goal of this paper is to show that the techniques developed for Bounded Model Checking may be adapted to deal with temporal logic specifications on bi-infinite time. While the encoding of automata and LTL into boolean logic is simple enough, being based on fixed point characterizations of the various temporal operators, the encoding of PLTL has only recently been defined in a satisfying way [1, 23]. In fact, the asymmetrical definition of past and future in PLTL actually complicates the translation of PLTL into a boolean formula. Our encoding for PLTL is actually simpler, because we consider past and future as completely symmetrical.

The paper is structured as follows: Section 2 briefly recalls the definition of LTL, introduces a syntactic variant of LTL (Metric LTL) with metric operators on time and finally summarizes an encoding of LTL into boolean logic. Section 3 introduces our new

encoding of PLTL extended with past operators on bi-infinite time. Section 4 reports on a tool, called ZOT, for translating Metric LTL with Past into boolean logic formulae, to be fed to a SAT solver. Section 5 introduces a nontrivial case study of specifications written in Metric LTL with past and bi-infinite time, and reports on verification results that support our goal. Section 6 reports future developments and draws a few conclusions.

2. PRELIMINARIES

Given a finite alphabet Σ, Σ^* denotes the set of finite words over Σ. An ω-word over Σ is an infinite sequence $w = a_0 a_1 a_2 \ldots$, with $a_j \in \Sigma$ for every $j \geq 0$. The set of all ω-words over Σ is denoted as Σ^ω. We denote an element a_j of $w = a_0 a_1 a_2 \ldots$ as $w(j)$, and the finite prefix $a_0 a_1 \ldots a_i$ of w as w_i.

We briefly recall here traditional Linear Temporal Logic (LTL) [34]. LTL could be defined as having also past operators (we call it PLTL in this case), but in this section only its (more common) future fragment is defined.

Syntax of LTL The alphabet of LTL includes: a finite set Ap of propositional letters; two propositional connectives $\neg, \vee$ (from which other traditional connectives such as $\top$ (true), $\bot$ (false), $\neg, \vee, \wedge, \rightarrow, \ldots$ may be defined); two temporal operators, the "until" operator $\mathcal{U}$, and the "next-time" operator $\circ$ (from which other temporal operators can be derived). Formulae are defined in the usual inductive way: a propositional letter $p \in Ap$ is a formula; $\neg\phi, \phi \vee \psi, \phi \mathcal{U} \psi, \circ\phi$, where ϕ, ψ are formulae, are formulae; nothing else is a formula. The traditional eventually and globally operators may be, respectively, defined as: $\Diamond\phi$ is $\top \mathcal{U} \phi$, $\Box\phi$ is $\neg\Diamond\neg\phi$.

Another useful operator is the dual of Until, i.e., the Release operator: $\phi \mathcal{R} \psi$ is $\neg(\neg\phi \mathcal{U} \neg\psi)$. Every LTL formula ϕ on the alphabet $\{\neg, \vee, \mathcal{U}, \circ\} \cup Ap$ may be transformed into an equivalent formula ϕ' on the alphabet $\{\wedge, \vee, \mathcal{U}, \mathcal{R}, \circ\} \cup Ap \cup \overline{Ap}$, where $\overline{Ap}$ is the set of atomic formulae of the form $\neg p$ for $p \in Ap$. Hence, negation in ϕ' may only occur on atoms. This is very convenient when defining encodings of LTL into propositional logic.

Semantics of LTL The semantics of LTL may be defined on ω-words. For all LTL formulae ϕ, for all $w \in (2^{Ap})^\omega$, for all natural numbers i, the satisfaction relation $w, i \models \phi$ is defined as follows.

$$
\begin{array}{l}
w, i \models p, \Longleftrightarrow p \in w(i), \text{for } p \in Ap \\
w, i \models \neg\phi \Longleftrightarrow w, i \not\models \phi \\
w, i \models \phi \vee \psi \Longleftrightarrow w, i \models \phi \text{ or } w, i \models \psi \\
w, i \models \circ\phi \Longleftrightarrow w, i+1 \models \phi \\
w, i \models \phi \mathcal{U} \psi \iff \\
\exists k \geq 0 : w, i+k \models \psi, \text{ and } \forall 0 \leq j < k : w, i+j \models \phi.
\end{array}
$$

Metric operators LTL can be extended by adding metric operators, on discrete time. Metric operators are very convenient for modeling hard real time systems, whose requirements include quantitative time constraints. We call the resulting logic Metric LTL.

Syntax of Metric LTL The alphabet is obtained by extending the alphabet of LTL with a *bounded until* operator $\mathcal{U}_{\sim c}$, where $\sim$ represents any relational operator (i.e., $\sim \in \{\leq, =, \geq\}$), and c is a natural number. Also, we allow n-ary predicate letters (with $n \geq 1$) and the $\forall, \exists$ quantifiers as long as their domains are finite. Hence, one can write, e.g., formulae of the form: $\exists p\ \mathrm{gr}(p)$, with p ranging over $\{1, 2, 3\}$ as a shorthand for $\bigvee_{p \in \{1,2,3\}} \mathrm{gr}_p$.

The bounded globally, bounded eventually and bounded release operators are defined as follows: $\Diamond_{\sim c}\phi$ is $\top \mathcal{U}_{\sim c} \phi$, $\Box_{\sim c}\phi$ is $\neg\Diamond_{\sim c}\neg\phi$, $\phi \mathcal{R}_{\sim c} \psi$ is $\neg\phi \mathcal{U}_{\sim c} \neg\psi$.

The "next-time" operator $\circ$ may then be considered as not primitive since $\circ\phi$ is $\top \mathcal{U}_{=1} \phi$.

In the following, as a useful shorthand, we will use also the versions of the bounded operators with a strict bound. For instance, $\phi \mathcal{U}_{>0} \psi$ stands for $\circ(\phi \mathcal{U}_{\geq 0} \psi)$, and similarly for the other ones.

Semantics of Metric LTL The semantics of Metric LTL may be defined on ω-words, by a straightforward translation of its operators into LTL:

$$
\begin{array}{l}
\tau(\phi_1 \mathcal{U}_{\leq 0} \phi_2) := \phi_2 \\
\tau(\phi_1 \mathcal{U}_{\leq t} \phi_2) := \phi_2 \vee (\phi_1 \wedge \circ\tau(\phi_1 \mathcal{U}_{\leq t-1} \phi_2)), \text{with } t > 0 \\
\hline
\tau(\phi_1 \mathcal{U}_{\geq 0} \phi_2) := \phi_1 \mathcal{U} \phi_2 \\
\tau(\phi_1 \mathcal{U}_{\geq t} \phi_2) := \phi_1 \wedge \circ\tau(\phi_1 \mathcal{U}_{\geq t-1} \phi_2), \text{with } t > 0 \\
\hline
\tau(\phi_1 \mathcal{U}_{=0} \phi_2) := \phi_2 \\
\tau(\phi_1 \mathcal{U}_{=t} \phi_2) := \phi_1 \wedge \circ\tau(\phi_1 \mathcal{U}_{=t-1} \phi_2), \text{with } t > 0 \\
\hline
\tau(\Diamond_{\sim t} \phi_1) := \tau(\top \mathcal{U}_{\sim t} \phi_1) \\
\hline
\tau(\phi_1 \mathcal{R}_{\sim t} \phi_2) := \neg\tau(\neg\phi_1 \mathcal{U}_{\sim t} \neg\phi_2) \\
\hline
\tau(\Box_{\sim t} \phi_1) := \tau(\bot \mathcal{R}_{\sim t} \phi_1)
\end{array}
$$

Hence, in what follows we will consider Metric LTL as a syntactic sugared version of LTL.

2.1 Mono-infinite future-tense Encoding for Bounded Model Checking

We describe next the encoding of LTL formulae, whose result includes additional information on the finite structure over which an LTL formula is interpreted, so that the resulting boolean formula is satisfied in the finite structure if and only if the original LTL formula is satisfied in a (finite or possibly) mono-infinite structure. This encoding is reported, to make the paper self-contained, from [5], Section 3.2: BMC for LTL with Eventualities.

For brevity in the following we call state S_i the set of assignments of truth values to propositional variables at time i. The idea on which the encoding is based is graphically depicted in Figure 1 (a). A ultimately periodic mono-infinite structure has a finite representation that includes the initial non periodic portion, and the periodic portion with a cycle that is encoded by having two equal states in the sequence: the interpreter of the formula (in our case, the SAT solver), when it needs to evaluate the subformula at a state beyond the last state S_k, will follow the "backward link" and consider the states S_l, S_{l+1}, ... as the states following S_k.

Let Φ be a LTL formula. Its semantics is given as a set of boolean constraints over the so called *formula variables*, i.e., fresh unconstrained propositional variables. There is a variable $|[\phi]|_i$ for each subformula ϕ of Φ and for each instant $0 \leq i \leq k+1$ (instant $k+1$, which is not explicitly shown in Figure 1, has a particular role in the encoding, as we will show next).

First, to allow for the representation of a mono-infinite structure into a finite one composed of $k+1$ states $S_0, S_1, \ldots S_k$, other $k+1$ fresh propositional variables $l_0, l_1, \ldots l_k$ must be introduced, called *loop selector variables*, which describe the loop that may exist in the finite structure. At most one of these loop selector variables may be true. If l_i is true then state $S_{i-1} = S_k$, i.e., the bit vectors representing the state S_{i-1} are identical to those for state S_k. Further propositional variables, InLoop_i ($0 \leq i \leq k$) and LoopExists, respectively mean that position i is inside a loop and that a loop actually exists in the structure.

The variables defining the loops in the finite structure are constrained by the following set of formulae, called *loop constraints*.

Loop constraints:

Base	$\neg l_0 \wedge \neg\mathrm{InLoop}_0$
$1 \leq i \leq k$	$l_i \Rightarrow (S_{i-1} = S_k)$
	$\mathrm{InLoop}_i \iff \mathrm{InLoop}_{i-1} \vee l_i$
	$\mathrm{InLoop}_{i-1} \Rightarrow \neg l_i$
	$\mathrm{LoopExists} \iff \mathrm{InLoop}_k$

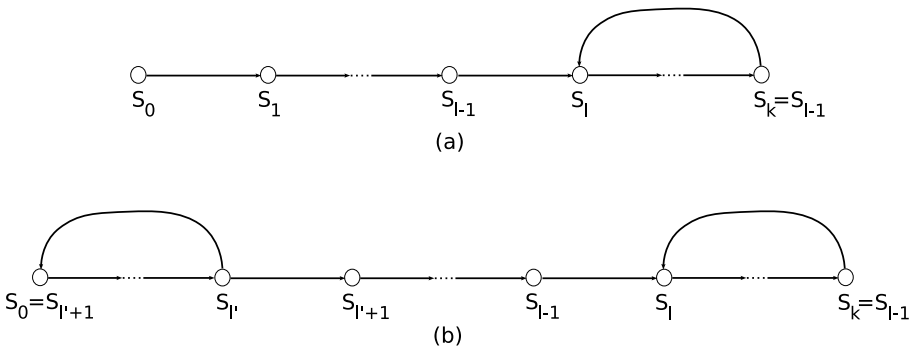

Figure 1: (a) Mono- and (b) bi-infinite bounded paths.

The above loop constraints state that the structure may have at most one loop. In the case of a cyclic structure, they allow the SAT solver to nondeterministically select exactly one of the (possibly) many loops.

Other formulae constrain in a natural way the propositional operators in Φ. For instance, if $\phi_i \wedge \phi_2$ is a subformula of Φ, then each variable $|[\phi_i \wedge \phi_2]|_i$ must be equivalent to the conjunction of variables $|[\phi_1]|_i$ and $|[\phi_2]|_i$.

Propositional constraints, with p denoting a propositional symbol:

ϕ	$0 \leq i \leq k$
p	$\|[p]\|_i \iff p \in S_i$
$\neg p$	$\|[\neg p]\|_i \iff p \notin S_i$
$\phi_i \wedge \phi_2$	$\|[\phi_i \wedge \phi_2]\|_i \iff \|[\phi_1]\|_i \wedge \|[\phi_2]\|_i$
$\phi_i \vee \phi_2$	$\|[\phi_i \vee \phi_2]\|_i \iff \|[\phi_1]\|_i \vee \|[\phi_2]\|_i$

The following formulae define the basic temporal behavior of LTL operators, by using their traditional fixpoint characterizations.

Temporal subformulae constraints (future):

ϕ	$0 \leq i \leq k$
$\circ\phi_1$	$\|[\circ\phi_1]\|_i \iff \|[\phi_1]\|_{i+1}$
$\phi_1 \mathcal{U} \phi_2$	$\|[\phi_1 \mathcal{U} \phi_2]\|_i \iff \|[\phi_2]\|_i \vee (\|[\phi_1]\|_i \wedge \|[\phi_1 \mathcal{U} \phi_2]\|_{i+1})$
$\phi_1 \mathcal{R} \phi_2$	$\|[\phi_1 \mathcal{R} \phi_2]\|_i \iff \|[\phi_2]\|_i \wedge (\|[\phi_1]\|_i \vee \|[\phi_1 \mathcal{R} \phi_2]\|_{i+1})$

Notice that such constraints do not consider the implicit eventualities that the definitions of $\mathcal{U}$ and $\mathcal{R}$ impose (they treat them as the "weak" until and release operators), nor consider that there may be loops (and hence "backward" constraints). To properly define eventualities, we need to introduce new propositional letters $\langle\langle \Diamond \phi_2 \rangle\rangle_i$, for each $\phi_1 \mathcal{U} \phi_2$ subformula of Φ, and for every $0 \leq i \leq k+1$. Analogously, we need to consider subformulae containing the operator $\mathcal{R}$, such as $\phi_1 \mathcal{R} \phi_2$, by adding the new propositional letters $\langle\langle \Box \phi_2 \rangle\rangle_i$. Then, constraints on these eventuality propositions are quite naturally stated as follows.

Eventuality constraints:

ϕ	Base
$\phi_1 \mathcal{U} \phi_2$	$\neg\langle\langle \Diamond \phi_2 \rangle\rangle_0 \wedge \left(\text{LoopExists} \Rightarrow \begin{pmatrix} \|[\phi_1 \mathcal{U} \phi_2]\|_k \Rightarrow \\ \langle\langle \Diamond \phi_2 \rangle\rangle_k \end{pmatrix}\right)$
$\phi_1 \mathcal{R} \phi_2$	$\langle\langle \Box \phi_2 \rangle\rangle_0 \wedge \left(\text{LoopExists} \Rightarrow \begin{pmatrix} \|[\phi_1 \mathcal{R} \phi_2]\|_k \Leftarrow \\ \langle\langle \Box \phi_2 \rangle\rangle_k \end{pmatrix}\right)$

ϕ	$1 \leq i \leq k$
$\phi_1 \mathcal{U} \phi_2$	$\langle\langle \Diamond \phi_2 \rangle\rangle_i \iff \langle\langle \Diamond \phi_2 \rangle\rangle_{i-1} \vee (\text{InLoop}_i \wedge \|[\phi_2]\|_i)$
$\phi_1 \mathcal{R} \phi_2$	$\langle\langle \Box \phi_2 \rangle\rangle_i \iff \langle\langle \Box \phi_2 \rangle\rangle_{i-1} \wedge (\neg\text{InLoop}_i \vee \|[\phi_2]\|_i)$

Last, the formulae in the following table provide the constraints that must be included in the encoding to account for the absence of a loop in the structure (the first line of the table states that if there is no loop nothing is true beyond the k-th state) or its presence (the second line states that if there is a loop at position i then state S_{k+1} and S_i are equivalent).

Last state constraints:

Base	$\neg$LoopExists $\Rightarrow \neg\|[\phi]\|_{k+1}$
$1 \leq i \leq k$	$l_i \Rightarrow (\|[\phi]\|_{k+1} \iff \|[\phi]\|_i)$

The complete encoding of Φ consists of the logical conjunction of the above components regarding loops, propositional connectives, temporal operators, and eventualities, together with $|[\Phi]|_0$ (i.e. Φ is evaluated only at instant 0).

3. PAST OPERATORS AND BI-INFINITE TIME

The definition of LTL in Section 2 may easily be extended to consider also past operators, by adding the (unbounded) since operator $\mathcal{S}$ and a past-time (or Yesterday) operator, $\bullet$, thus leading to the temporal logic language PLTL.

Clearly, one can also add $\mathcal{S}_{\sim c}$ to Metric LTL, to define a metric temporal logic with past, Metric PLTL. The cited TRIO language without first-order variables and quantifiers, is actually equivalent to Metric PLTL.

The past version of the eventually and globally operators, also in the bounded versions, may be defined symmetrically to their future counterparts: $\blacklozenge\phi$ is $\top \mathcal{S} \phi$, $\blacksquare\phi$ is $\neg\blacklozenge\neg\phi$; $\blacklozenge_{\sim c}\phi$ is $\top \mathcal{S}_{\sim c} \phi$, $\blacksquare_{\sim c}\phi$ is $\neg\blacklozenge_{\sim c}\neg\phi$. A useful operator for MPLTL is the Always operator $\mathcal{A}lw$, which can be defined by $\mathcal{A}lw\phi_1 := \Box\phi_1 \wedge \blacksquare\phi_1$. The intended meaning of $\mathcal{A}lw\phi_1$ is that ϕ_1 must hold in every instant in the future and in the past. Its dual is the Sometimes operator $\mathcal{S}om\phi$ defined as $\neg\mathcal{A}lw\neg\phi$.

3.1 Bi-infinite Encoding of PLTL

Traditionally, (e.g., in [5]) the encoding of PLTL is given on a mono-infinite temporal domain, such as the one showed in Figure 1 (a). Such BMC semantics can be tricky to define, because of the asymmetric role of future and past. Moreover, it needs the introduction of a conventional value for $\bullet\phi$ at instant 0, usually $\bot$, because it refers to an instant (-1) outside of the temporal domain. For this reason, to push negation to propositional letters it is necessary to have another operator, usually called *Zeta* (or $\bullet'$), such that $\bullet\phi := \neg\bullet'(\neg\phi)$. Furthermore, because of the asymmetric

structure of time, one must take into account also the possibility that the evaluation of any (sub)formula involves some time point outside the temporal domain, therefore the encoding defined in [5] includes a complex mechanism that permits to set apart the cases where some border effect in the formula evaluation may take place. This is based on comparing the depth of the nesting of temporal operators with the distance from the origin of the time axis of the "current interpretation time", which in turn requires keeping track of the number of times that the loop in the temporal structure has been unfolded to reach the current time.

On the other hand, defining semantics for PLTL is more natural on a bi-infinite time domain, because in this case past and future have the same role and importance.

A bounded bi-infinite temporal structure is shown in Figure 1 (b). As the reader may notice, it is a natural extension of the mono-infinite structure: the loops are now at most two, one towards the future (as before), and a new one towards the past. There are new loop selector variables l'_i to define the loop which goes towards the past, and the corresponding propositional letters $\mathrm{InLoop'}_i$, and $\mathrm{LoopEsists'}$.

To define the bi-infinite encoding of PLTL, we have to use the following constraints, similar to those presented in Section 2.1, for the future part. In addition to them, we introduce the following, for the back-loops.

Back-loop constraints:

Base	$\neg l'_k \wedge \neg \mathrm{InLoop'}_k$
$1 \leq i \leq k$	$l'_i \Rightarrow (S_{i+1} = S_0)$ $\mathrm{InLoop'}_i \iff \mathrm{InLoop'}_{i+1} \vee l'_i$ $\mathrm{InLoop'}_{i+1} \Rightarrow \neg l'_i$ $\mathrm{LoopExists'} \iff \mathrm{InLoop'}_0$

symmetrical to the temporal constraints for the future subformulae, we introduce their past counterparts, and eventualities.

Temporal subformulae constraints (past):

ϕ	$0 \leq i \leq k$
$\bullet\phi_1$	$\|[\bullet\phi_1]\|_i \iff \|[\phi_1]\|_{i-1}$
$\phi_1 \mathcal{S} \phi_2$	$\|[\phi_1 \mathcal{S} \phi_2]\|_i \iff \|[\phi_2]\|_i \vee (\|[\phi_1]\|_i \wedge \|[\phi_1 \mathcal{S} \phi_2]\|_{i-1})$
$\phi_1 \mathcal{T} \phi_2$	$\|[\phi_1 \mathcal{T} \phi_2]\|_i \iff \|[\phi_2]\|_i \wedge (\|[\phi_1]\|_i \vee \|[\phi_1 \mathcal{T} \phi_2]\|_{i-1})$

Eventuality constraints (past):

ϕ	Base
$\phi_1 \mathcal{S} \phi_2$	$\neg\langle\langle \blacklozenge \phi_2 \rangle\rangle_k \wedge \left(\mathrm{LoopExists'} \Rightarrow \begin{pmatrix} \|[\phi_1 \mathcal{S} \phi_2]\|_0 \\ \Rightarrow \\ \langle\langle \blacklozenge \phi_2 \rangle\rangle_0 \end{pmatrix} \right)$
$\phi_1 \mathcal{T} \phi_2$	$\langle\langle \blacksquare \phi_2 \rangle\rangle_k \wedge \left(\mathrm{LoopExists'} \Rightarrow \begin{pmatrix} \|[\phi_1 \mathcal{T} \phi_2]\|_0 \\ \Leftarrow \\ \langle\langle \blacksquare \phi_2 \rangle\rangle_0 \end{pmatrix} \right)$

ϕ	$1 \leq i \leq k$
$\phi_1 \mathcal{S} \phi_2$	$\langle\langle \blacklozenge \phi_2 \rangle\rangle_i \iff \langle\langle \blacklozenge \phi_2 \rangle\rangle_{i+1} \vee \begin{pmatrix} \mathrm{InLoop'}_i \\ \wedge \\ \|[\phi_2]\|_i \end{pmatrix}$
$\phi_1 \mathcal{T} \phi_2$	$\langle\langle \blacksquare \phi_2 \rangle\rangle_i \iff \langle\langle \blacksquare \phi_2 \rangle\rangle_{i+1} \wedge \begin{pmatrix} \neg\mathrm{InLoop'}_i \\ \vee \\ \|[\phi_2]\|_i \end{pmatrix}$

Then, symmetrically to the last state, we must define first state (i.e. 0 time) constraints (notice that in the bi-infinite encoding instant -1 has a symmetric role of instant $k+1$).

First state constraints:

Base	$\neg\mathrm{LoopExists'} \Rightarrow \neg\|[\phi]\|_{-1}$
$1 \leq i \leq k$	$l'_i \Rightarrow (\|[\phi]\|_{-1} \iff \|[\phi]\|_i)$

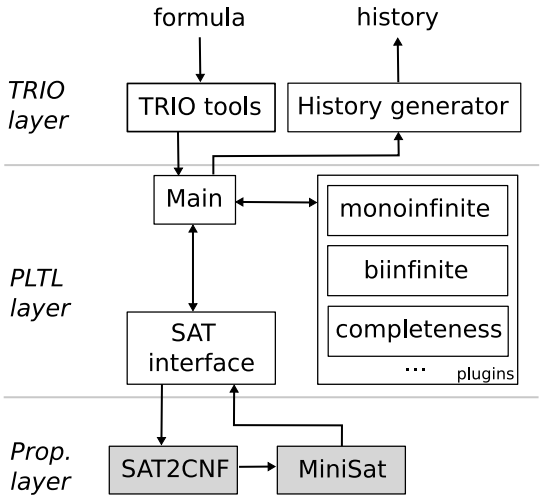

Figure 2: ZOT's architecture (external tools in grey).

Last, the following constraints must be added to allow mixing past and future formulae. The past part is necessary also in a mono-infinite time structure (and can therefore be found also in [5]). It defines the behavior of the past operators on the future loop, if this is present.

Stabilizing constraints (past):

ϕ	$1 \leq i \leq k$
$\bullet\phi_1$	$\|[\bullet\phi_1]\|_i \iff$ $(l_i \wedge \|[\phi_1]\|_k \vee \neg l_i \wedge \|[\phi_1]\|_{i-1})$
$\phi_1 \mathcal{S} \phi_2$	$\|[\phi_1 \mathcal{S} \phi_2]\|_i \iff$ $\|[\phi_2]\|_i \vee (\|[\phi_1]\|_i \wedge (l_i \wedge \|[\phi]\|_k \vee \neg l_i \wedge \|[\phi]\|_{i-1}))$
$\phi_1 \mathcal{T} \phi_2$	$\|[\phi_1 \mathcal{T} \phi_2]\|_i \iff$ $\|[\phi_2]\|_i \wedge (\|[\phi_1]\|_i \vee (l_i \wedge \|[\phi]\|_k \vee \neg l_i \wedge \|[\phi]\|_{i-1}))$

Similarly and symmetrically, we must take into account the behavior of future operators on the back-loop, if present.

Stabilizing constraints (future):

ϕ	$1 \leq i \leq k$
$\circ\phi_1$	$\|[\circ\phi_1]\|_i \iff$ $(l'_i \wedge \|[\phi_1]\|_0 \vee \neg l'_i \wedge \|[\phi_1]\|_{i+1})$
$\phi_1 \mathcal{U} \phi_2$	$\|[\phi_1 \mathcal{U} \phi_2]\|_i \iff$ $\|[\phi_2]\|_i \vee (\|[\phi_1]\|_i \wedge (l'_i \wedge \|[\phi]\|_0 \vee \neg l'_i \wedge \|[\phi]\|_{i+1}))$
$\phi_1 \mathcal{R} \phi_2$	$\|[\phi_1 \mathcal{R} \phi_2]\|_i \iff$ $\|[\phi_2]\|_i \wedge (\|[\phi_1]\|_i \vee (l'_i \wedge \|[\phi]\|_0 \vee \neg l'_i \wedge \|[\phi]\|_{i+1}))$

4. A TOOL

The encoding of Section 3.1 was incorporated into a tool called ZOT, an agile and easily extendible bounded satisfiability checker, designed and implemented by the first author. ZOT can be downloaded at http://www.elet.polimi.it/upload/pradella, together with the case study described in Section 5. The tool supports different logic languages through a multi-layered approach: its core uses PLTL, and on top of it a decidable predicative fragment of TRIO is defined. An interesting feature of ZOT is its ability to support different encodings of temporal logic as SAT problems. Indeed, the user can choose a particular encoding to carry out verification, while the tool automatically loads the corresponding plugin. This

approach encourages experimentation, as plugins are usually quite simple, compact (usually around 500 lines of code), easily modifiable, and extendible. At the moment, a few variants of some of the encodings presented in [5] are supported, together with the bi-infinite encoding presented here. Figure 2 shows ZOT's internal architecture. At present ZOT and communicates with the SAT solver MiniSat through files. Being the input file format for SAT solvers standard (DIMACS CNF), it is easy to adapt ZOT to use other SAT solvers.

At present, ZOT essentially adapts bounded model checking techniques to purely descriptive TRIO specifications. This means that both the model and the property are expressed as TRIO formulae, like in [30], and unlike typical model checking.

ZOT offers two basic usage modalities:

1. *Bounded satisfiability checking (BSC)*: given as input a specification formula, the tool returns a (possibly empty) history (i.e., an execution trace of the specified system) which satisfies the specification. An empty history means that it is impossible to satisfy the specification with the given bound.

2. *History checking and completion (HCC)*: The input specification file can also contain a partial (or complete) history H. In this case, if H complies with the specification, then a completed version of H is returned as output, otherwise the output is empty.

The provided output histories have always temporal length $\leq k$, the bound given by the user, but may represent infinite behaviors thanks to the loop selector variables, marking the start of the periodic sections of the history. The BSC modality can be used to check if a property *prop* of the given specification *spec* holds over every periodic behavior with period $\leq k$. In this case, the input file contains $\mathrm{spec} \wedge \neg \mathrm{prop}$, and, if prop indeed holds, then the output history is empty. If this is not the case, the output history is a useful counterexample which explains why prop does not hold.

ZOT also supports completeness checks, through a logic-based variant of some of the algorithms described in [37]. Essentially, completeness checking is based on an encoding very similar to the one presented in the previous part of this paper. The main difference resides in the loops: the completeness encoding considers only the *finite* component of the temporal structure. To assure completeness, constraints are added asserting that it is impossible to have two states S_i and S_j, such that $S_i = S_j$, for any $i \neq j$. Therefore, we are considering all the possible finite behaviors of the system under analysis. This check can trivially be iterated to automatically find the bound which ensures completeness. For systems described using pure logic, this check can be very expensive, because they usually contain a high degree of nondeterminism, and the completeness check is by its very nature exhaustive. We will show some consequences of this behavior in the next section.

5. CASE STUDY AND EXPERIMENTS

To assess the actual feasibility of our approach, we applied it to a case study consisting of a real-time allocator which serves a set of client processes, competing for a shared resource. Each process p requires the resource by issuing the message $\mathrm{rq}(p)$, by which it identifies itself to the allocator. Requests have a time out: they must be served within T_{req} time units, or else be ignored by the allocator. If the allocator is able to satisfy p's request within the time-out, then it grants the resource to p by a $\mathrm{gr}(p)$ signal. Once a process is assigned the resource by the allocator, it releases the resource, by issuing a rel signal, within a maximum of T_{rel} time units. The allocator grants the request to processes according to a FIFO policy, considering only requests that are not timed out yet and in a timely manner, i.e., no process will have to wait for the resource while it is not assigned to any other process.

5.1 Formal specification of the case study

The following table lists the alphabet of the specification, including the names of predicates, predicate letters, time constants with a sketchy description of their meaning.

Spec. item	Description
$\mathrm{rq}(p)$	process p requests the resource
$\mathrm{gr}(p)$	the resource is granted by the allocator to process p
rel	the resource is released (by the process currently holding it)
T_{rel}	Timeout after which any process must release the resource
T_{req}	Timeout of a request: after it elapses the request expires (it is not active any more) and the allocator must ignore it
$\mathrm{APR}(p)$	(Active Pending Request) in the recent past process p has issued a request that is still active (timeout T_{req} not elapsed yet) and is pending (it has not been satisfied so far)
$\mathrm{LRAPR}(p)$	(Least Recent Active Pending Request) there is an active pending request (see predicate $\mathrm{APR}(p)$) by process p, and it is the least recent one (all other active pending requests are more recent)
$\mathrm{available}$	the resource is available (not assigned to any process)

The specification is composed of the following axioms, where variables p and q denoting processes are ranging over the integer set $[1..n]$, n being the assumed number of processes.

1. The resource is assigned to process p iff it is available and p has the least recent active pending request

$$\mathrm{gr}(p) \leftrightarrow \mathrm{available} \wedge \mathrm{LRAPR}(p)$$

2. The resource is available iff it has not been granted to any process, either from the release by the last process that was assigned it, or forever in the past

$$\mathrm{available} \leftrightarrow \neg\exists p\ (\mathrm{gr}(p)\mathcal{S}_{>0}\mathrm{rel}) \vee \blacksquare\neg\exists p\ \mathrm{gr}(p)$$

3. A request by process p is active and pending if it was issued by a process p less than T_{req} time units ago, and the resource was not granted to p since then

$$\mathrm{APR}(p) \leftrightarrow \left(\begin{array}{c} \neg\mathrm{gr}(p)\mathcal{S}_{>0}\mathrm{rq}(p) \\ \wedge \\ (\bullet\neg\mathrm{gr}(p))\mathcal{S}_{<T_{req}}\mathrm{rq}(p) \end{array} \right)$$

4. A request by process p is the least recent active pending one if it is an active pending request and if there is no other less recent request by another process q. Here variables t_p and t_q indicate time distances, and range on the set $[1..T_{req}]$.

$$\begin{array}{c} \mathrm{LRAPR}(p) \\ \leftrightarrow \\ \exists t_p \left(\begin{array}{c} \mathrm{APR}(p) \\ \wedge \\ \neg\mathrm{rq}(p)\mathcal{S}_{=t_p} rq(p) \\ \wedge \\ \neg\exists q\ \exists t_q \left(\begin{array}{c} q \neq p \wedge \\ t_q > t_p \wedge \\ \mathrm{APR}(q) \wedge \\ \neg\mathrm{rq}(q)\mathcal{S}_{=t_p} rq(q) \end{array} \right) \end{array} \right) \end{array}$$

5. Once granted the resource, any process p keeps it for at least one time unit and releases it within T_{rel} time units

$$\mathrm{gr}(p) \rightarrow \neg\mathrm{rel} \wedge \Diamond_{<=T_{rel}} \mathrm{rel}$$

6. There are no spurious release signals: a release signal is issued only if there has been no previous release since the last grant of the resource to any process p

$$\mathrm{rel} \rightarrow \neg\mathrm{rel}\mathcal{S}_{>0}\exists p\ \mathrm{gr}(p)$$

7. There can be no simultaneous requests by two distinct processes (this is an immaterial, simplifying assumption, which may correspond to some feature of the device that samples the resource requests by the processes)

$$\neg\exists p\ \exists q\ (\mathrm{rq}(p) \wedge \mathrm{rq}(q) \wedge p \neq q)$$

8. There are no spurious resource requests by any process, i.e., a process will not issue a resource request if there is an active pending request by the same process, or if the process is holding the resource (the resource has been granted to the process and since then it has not released it)

$$(\mathrm{APR}(p) \vee \neg\mathrm{rel}\mathcal{S}_{\geq 0}\mathrm{gr}(p)) \rightarrow \neg\mathrm{rq}(p)$$

The overall specification of the real-time allocator system is obtained by prefixing all axioms by universal quantifications on any free variable, by conjoining the axiom and prefixing universal temporal quantification operator $\mathcal{A}lw$.

5.2 Property verification

We employed the tool, as reported in the next subsection, to analyze the following (conjectured) properties.

If a process that does not obtain the resource always requests it again immediately after the request is expired, then if it requests the resource it will eventually obtain it. This property is called *SimpleFairness*.

$$\begin{array}{c} \mathcal{A}lw\left((\mathrm{rq}(p) \wedge \Box_{\leq T_{req}} \neg\mathrm{gr}(p)) \rightarrow \Diamond_{=T_{req}} \mathrm{rq}(p)\right) \\ \rightarrow \\ \mathcal{A}lw\left(\mathrm{rq}(p) \rightarrow \Diamond\mathrm{gr}(p)\right) \end{array}$$

The results we obtained by analyzing the property with the tool depend on the number of processes that are assumed to be included in the system.

With two processes, the tool procedure to find a bound k on length number of states in the interpretation structure that ensures completeness succeeded with $k = 65$, and the fact that the negation of the property was declared unsatisfiable enables us to conclude that the property in fact holds.

On the other hand, for a configuration of three processes, the procedure for finding the bound k ensuring completeness did not converge in any reasonable time. However, the the tool provided as a counterexample of the analyzed property a trace of the system where two of the processes are alternating in requiring and getting the resource, at the expenses of the third process, who continuously requires and never receives it because one of the other two has required the resource right before it. Hence the negation of the property is satisfiable, and the property is therefore disproved, even though we were not able produce the bound ensuring completeness.

A second, more complex property may be intuitively described as a sort of "conditional fairness". Let us define the notion of "unconstrained rotation" among processes: a process will require the resource only after all other ones have requested and obtained it. Notice that this requirement does not impose any precise ordering among the requests made by the processes (though, once requests take place in a given order, the order remains unchanged from one rotation among processes to the next one). This property is described by the following formula:

$$\mathcal{A}lw\left(\begin{array}{c} \mathrm{rq}(p) \rightarrow \\ \forall q\left(q \neq p \rightarrow \neg\mathrm{rq}(p)\mathcal{S}\left(\begin{array}{c} \mathrm{rq}(q) \wedge \\ \Diamond_{\leq T_{req}}\mathrm{gr}(q) \end{array}\right)\right) \end{array}\right)$$

Under this assumption of "unconstrained rotation" the allocator system is fair for all processes: if a process, when it requests the resource and does not obtain it, always requests it again after the request is expired, then, when it requests the resource, it will eventually obtain it. If for brevity we symbolically indicate the property of "unconstrained rotation" as *UNROT*, then the "*ConditionalFairness*" property may be stated as:

$$\mathrm{UNROT} \rightarrow \left(\begin{array}{c} \mathcal{A}lw(\mathrm{rq}(p) \wedge \Box_{\leq T_{req}} \neg\mathrm{gr}(p) \rightarrow \Diamond_{>0}\mathrm{rq}(p)) \\ \rightarrow \\ \mathcal{A}lw(\mathrm{rq}(p) \rightarrow \Diamond_{>0}\mathrm{gr}(p)) \end{array}\right)$$

Again, the property was proven by the tool, for a k ensuring completeness, in the cases of two processes. For the case with three processes, the tool answered "unsatisfiable" for the negation of the property with reference to any structure with a number of states up to 200, hence we are led to conjecture that the property in fact holds, but as long as no bound k ensuring completeness is found in the case of three processes this must remain a (likely) conjecture.

The property of unconstrained rotation, in the simple form of the above UNROT formula (this was the simplest formalization we could devise) implies a sequence of request events (and corresponding grant and release) that goes back indefinitely towards the past, therefore it can be satisfied only by the kind of bi-infinite structure that can be built by our tool. While it is a trivial fact that no events sequence that goes infinitely towards the past can actually take place, we remark that since the structure considered by the tool by any simulation is periodic, the objects generated by the tool during the analysis of the property are finite; besides, the formal characterization of the UNROT property is quite compact right because of this assumption of periodicity in the past: the formula

k	2 processes	3 processes
50	unsat (183s,50s,117Mb)	sat (405s,101s,170Mb)
100	valid	not valid
200	valid	not valid

Table 1: Verification of SimpleFairness (translation time, SAT-solver time, memory)

k	2 processes	3 processes
50	unsat (198s,54s,122Mb)	unsat (459s,130s,180Mb)
100	valid	unsat (1764s,528s,347Mb)
200	valid	unsat (6456s,2814s,399Mb)

Table 2: Verification of ConditionalFairness (translation time, SAT-solver time, memory)

describing that property would be much more complex if the designer had to refer to a finite trace that includes a start event before which no other event took place. In summary, in can be stated that the assumption of a sequence of events that extends itself indefinitely in the past is a useful abstraction with respect to the start of the allocator system: a designer might prefer to ignore the behavior of the allocator right after its start and consider its properties only on routine behavior.

5.3 Experimental results

The experiments were run on a PC equipped with AMD Athlon 64 X2 4600+, 2 Gb RAM, Linux OS. The SAT-solver was MiniSat, v. 2.0 beta, along with SAT2CNF, part of the Alloy Analyzer (http://alloy.mit.edu). The experimental results on properties SimpleFairness and ConditionalFairness and on completeness for the allocator case study are shown in Tables 1,2 and 3. The experiments considered three different values of k (the bound for the bounded model checking enconding): 50, 100, 200. We also studied the differences in performance of allowing two or three processes in the case study.

As it can be seen, both SimpleFairness and ConditionalFairness could easily be checked by the tool, in the cases of two and three processes, for every chosen value of k. As reported, ConditionalFairness holds in every case, while SimpleFairness holds for the case of two processes (the tool returned "unsat"), but it is violated (the tool returns "sat") with three processes already with $k = 50$.

The actual bound k used in the verifications, while influencing the performance, was not really critical. Completeness was critical instead. The tool is able to find that, with two processes, the bound $k = 50$ is enough for completeness. Hence, since both SimpleFairness and ConditionalFairness were unsatisfiable with $k = 50$, there was no need to bother with greater values of k: completeness ensures that these results hold on any bi-infinite domain. For the case of three processes, the result for SimpleFairness ("sat") is enough to say that the property is violated. However, since no bound could be found for completeness in the case of three processes, the verification of property ConditionalFairness up to $k = 200$ is not enough to ensure that the property actually holds, since in theory there might exist a counterexample of length greater than 200.

Translating from PLTL into a boolean formula was the most time-consuming activity. This is not always the case, since when a property is very hard to be verified, the time for the SAT solver may be dominant. Translation time is heavily dependent on the normalization of the boolean formulae encoding PLTL into the conjunctive normal form expected by SAT solvers. This could be improved by better tools.

k	2 processes	3 processes
50	unsat (32s,54s,71Mb)	sat (74s,16s,71Mb)
100	useless	? (>24h)
200	useless	? (>24h)

Table 3: Completeness output (translation time, SAT-solver time, memory (Mb))

6. CONCLUSIONS AND FUTURE WORK

In this paper we have argued that bi-infinite time in specification is a useful abstraction, allowing one to ignore the complexity of system initialization. Although bi-infinite time has certainly been used before in specification (also by us), we are not aware of any other work to extend automated formal verification in general, and bounded model checking in particular, to deal with bi-infinite time.

Our approach in the present paper was centered on the problem of *satisfiability* checking of temporal logic formulae, since this was our motivating concern with the TRIO language. Our main focus was to show that bounded model checking techniques may be adapted to deal with bi-infinite time in temporal logic, without essentially any loss in performance with respect to the traditional mono-infinite case. The experimental results of Section 5 show that satisfiability checking may be feasible also on nontrivial examples of temporal logic specifications. Even the size of the bound was not particularly critical.

The only difficulty came from the procedure for computing the bound k that ensures completeness of the proofs, which suffered from the combinatorial explosion of allowing three, rather than two, processes. In fact, the computation of this bound with three processes was the only case where our tool was unable to terminate in reasonable time. We plan to refine the implemented procedure for completeness, which was derived with limited adjustments from similar procedures appeared in the literature on bounded model checking. We are aware, however, that the problem is intrinsically combinatorial, and strongly influenced by the degree of nondeterminism of a model, hence we do not expect major breakthrough in this direction.

7. REFERENCES

[1] M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *Proceedings of the Intern. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2003, Warsaw, Poland, April 7-11, 2003.*, volume 2619 of *Lecture Notes in Computer Science*, pages 18–33. Springer, 2003.

[2] D. Bianculli, A. Morzenti, M. Pradella, P. San Pietro, and P. Spoletini. Trio2Promela: a model checker for temporal metric specifications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 61–62, Los Alamitos, CA, USA, May 2007. IEEE Computer Society. Research Demo.

[3] D. Bianculli, P. Spoletini, A. Morzenti, M. Pradella, and P. San Pietro. Model checking temporal metric specification with Trio2Promela. In *Proc. of IPM International Symposium on Fundamentals of Software Engineering (FSEN'07)*, Lecture Notes in Computer Science. Springer, 2007. To appear.

[4] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[5] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2(5):1–64, 2006.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[7] R. Capobianchi, A. Coen-Porisini, D. Mandrioli, and A. Morzenti. A framework architecture for supervision and control systems. *ACM Comput. Surv.*, 32(1es):26, 2000.

[8] E. Ciapessoni, P. Mirandola, A. Coen-Porisini, D. Mandrioli, and A. Morzenti. From formal models to formally based methods: An industrial experience. *ACM Trans. Softw. Eng. Methodol.*, 8(1):79–113, 1999.

[9] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV '02: Proceedings of the 14th Intern. Conf. on Computer Aided Verification*, pages 359–364, London, UK, 2002. Springer-Verlag.

[10] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

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

[13] A. Coen-Porisini, M. Pradella, and P. San Pietro. A finite-domain semantics for testing temporal logic specifications. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 5th Intern. Symposium, FTRTFT'98, Lyngby, Denmark, September 14-18, 1998, Proceedings*, pages 41–54, 1998.

[14] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[15] M. Felder and A. Morzenti. Validating real-time systems by history-checking trio specifications. *ACM Trans. Softw. Eng. Methodol.*, 3(4):308–339, 1994.

[16] A. Gargantini and A. Morzenti. Automated deductive requirements analysis of critical systems. *ACM Trans. Softw. Eng. Methodol.*, 10(3):255–307, 2001.

[17] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conf. on Computer Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 53–65. Springer Verlag, 2001.

[18] P. Gastin and D. Oddoux. LTL with past and two-way very-weak alternating automata. In *Proceedings of the 28th Intern. Symposium on Mathematical Foundations of Computer Science (MFCS'03)*, number 2747 in Lecture Notes in Computer Science, pages 439–448. Springer Verlag, 2003.

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

[20] F. Gire and M. Nivat. Langages algébriques de mots biinfinis. *Theoret. Comput. Sci.*, 86(2):277–323, 1991.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special Issue: Formal Methods in Software Practice.

[22] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison Wesley, 2004. HOL g 03:1 1.Ex.

[23] T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is better: Efficient bounded model checking for past LTL. In R. Cousot, editor, *Proceedings of the 6th Intern. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 380–395, Paris, France, January 2005. Springer-Verlag.

[24] O. Lichtenstein, A. Pnueli, and L. D. Zuck. The glory of the past. In *Proceedings of the Conf. on Logic of Programs*, pages 196–218, London, UK, 1985. Springer-Verlag.

[25] D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real-time systems from logic specifications. *ACM Trans. Comput. Syst.*, 13(4):365–398, 1995.

[26] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[27] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[28] S. Morasca, A. Morzenti, and P. San Pietro. A case study on applying a tool for automated system analysis based on modular specifications written in TRIO. *Autom. Softw. Eng.*, 7(2):125–155, 2000.

[29] A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14(4):521–573, 1992.

[30] A. Morzenti, M. Pradella, P. San Pietro, and P. Spoletini. Model-checking TRIO specifications in SPIN. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 542–561. Springer, 2003.

[31] A. Morzenti and P. San Pietro. Object-oriented logical specification of time-critical systems. *ACM Trans. Softw. Eng. Methodol.*, 3(1):56–98, 1994.

[32] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. In *DAC '01: Proceedings of the 38th Conf. on Design automation*, pages 530–535, New York, NY, USA, 2001. ACM Press.

[33] D. Perrin and J. Éric Pin. *Infinite Words*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004. ISBN 0-12-532111-2.

[34] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, Providence, Rhode Island, 31– 2 1977. IEEE Computer Society Press.

[35] M. Pradella, P. San Pietro, P. Spoletini, and A. Morzenti. Practical model checking of LTL with Past. In *ATVA03, Taipei, Taiwan, 2003*, 2003.

[36] P. San Pietro, A. Morzenti, and S. Morasca. Generation of execution sequences for modular time critical systems. *IEEE Trans. Software Eng.*, 26(2):128–149, 2000.

[37] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD00, volume 1954 of LNCS*, pages 108–125. Springer-Verlag, 2000.

[38] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2000.

[39] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 332–344, Washington, DC, 1986. IEEE Computer Society Press.

[40] P. Wolper. Temporal logic can be more expressive. In *FOCS*, pages 340–348. IEEE, 1981.