Benchmarking Model- and Satisfiability-Checking on bi-infinite time

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

[1] IEIIT, Consiglio Nazionale delle Ricerche, Milano, Italy
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{pradella, morzenti, sanpietr}@elet.polimi.it

Abstract. Model checking techniques traditionally deal with temporal logic languages and automata interpreted over ω-words, i.e., where time is infinite in the future but finite in the past. This is motivated by the study of reactive systems, which are typically nonterminating: system termination may be abstracted away by allowing an infinite future. In the same way, if time is infinite also in the past one is allowed to ignore the complexity of system initialization. Specifications may then be simpler and more easily understandable, because they do not necessarily include the description of operations (such as configuration or installation) typically performed at system deployment time. In this paper, we investigate the feasibility of bounded model checking and bounded satisfiability checking when dealing with bi-infinite automata and logics. We present a tool and we discuss its application to a set of case studies, arguing that bi-infinite time does not entail significant penalties in verification time and space.

Keywords: Bounded model checking, bi-infinite words and automata, metric temporal logic.

1 Introduction

Temporal logics and automata models used in specification and verification usually consider time to be finite in the past, i.e., with a “first” time instant. The reason is both pragmatical and historical: finite automata and temporal logic were applied to model programs or hardware, where often there is an initialization step. Hence, automata and temporal logics (the latter also extended with past operators) on ω-words seemed adequate for modeling and verification. The only concession to infinity was in the future: a reactive system 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. 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. Nontermination is only an abstraction, useful to write simpler models that avoid explicit consideration of the final disposal of the analyzed system, and to verify and analyze properties that essentially refer to infinite behaviors, such as fairness.

However, philosophers and logicians such as Prior have always considered that time may be bi-infinite, i.e., infinite both in the future and in the past. Also Automata Theory has considered bi-infinite computations [10, 22]. Actually, bi-infinity may be a useful abstraction, too. Analogously to the mono-infinite case where termination may be ignored, bi-infinite time is convenient for modeling systems where initialization may be ignored. 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, one may often focus only on regime behavior, ignoring initialization. As an example, consider a simple mutual exclusion problem, where, say, three processes may need to gain exclusive access to a shared resource R. A resource allocator might have a policy, in case of conflicting resource requests, to allocate R first to the process, among those that are currently requesting R, that accessed the resource least recently. This fairness property may be formalized more easily by assuming that every request by a process is preceded by a previous request by some other process, i.e., that the sequence of requests extends indefinitely in the past (a property similar to this one will be formalized in Section 5).

Recent developments in the research on Bounded Model Checking (**BMC**), a verification technique originally defined only for Linear Time Temporal Logic (LTL) [2], have extended its applicability also to PLTL (LTL with Past time operators) [1, 14]. In these works, however, the time domain is infinite in the future only, so that the asymmetrical definition of past and future in PLTL actually complicates the translation of PLTL into a boolean formula. In our work [23] we presented a novel, bi-infinite encoding of PLTL for bounded model checking, which is significantly simpler than the previous ones, because we consider past and future as completely symmetrical. Furthermore, we introduced a variant of bounded model checking where both the system under analysis and the property to be checked are expressed in a single uniform notation as formulae of temporal logic, without any reference to operational components. In this novel setting, which we called bounded *satisfiability* checking (**BSC**), the system under analysis is modeled through the set of all its fundamental properties as a formula ϕ (that in all non-trivial cases would be of significant size) and the additional property to be checked (e.g. a further desired requirement) is expressed as another (usually much smaller) formula ψ. Aim of the verification activity is then to prove that any implementation of the system under analysis possessing the assumed fundamental properties ϕ would also ensure the additional property ψ; in other terms, the verification tool would prove that the formula $\phi \rightarrow \psi$ is valid, or equivalently that its negation is not satisfiable (hence the term satisfiability checking).

The present paper provides two more contributions. First, the above outlined method of satisfiability checking is generalized by permitting, also for the case of a bi-infinite time, the more customary operation of bounded *model* checking, where the system under analysis is modeled by means of a finite state automaton: to this end we provide the definition of automata on bi-infinite strings and the encoding of the automaton on bi-infinite time structures. Second, we validate our approach and the tool implemented to support it by comparing the performance figures obtained on mono- and bi-infinite time structures, by performing either model or satisfiability checking on a set of selected case studies. The results allow us to state that the bi-infinite approach is feasible and can be applied with no significant penalty, also when considering a mono-infinite specification. Since bi-infinite time is also more natural and more expressive, we argue that one may use a bi-infinite approach to specification and verification, even when the system to be modeled is mono-infinite.

The paper is structured as follows: Section 2 provides definitions of bi-infinite words, automata and logic, while Section 3 motivates the usefulness of a bi-infinite semantics. Section 4 briefly describes a toolkit, called Zot, extended with the new encoding, translating models and formulae into boolean logic. Section 5 presents experimental results, using Zot and MiniSat solver, comparing mono-infinite and bi-infinite verifications of a set of case studies, showing the feasibility of bi-infinite bounded model checking. Finally, Section 6 draws some conclusions.

2 Automata and logics on bi-infinite words

Given a finite alphabet Σ, Σ^* denotes the set of finite words over Σ. A bi-infinite word w over Σ (also called a $\mathbb{Z}$-word) is a function $w : \mathbb{Z} \longrightarrow \Sigma$. Hence, $w(j) \in \Sigma$ for every j. Word w is also denoted as $\dots w(-1)w(0)w(1)\dots$ and each $w(j)$ also as w_j. The set of all bi-infinite words over Σ is denoted by $\Sigma^{\mathbb{Z}}$. An ω-word over Σ is a function from $\mathbb{N} \rightarrow \Sigma$, i.e., it has the form $w(0)w(1)\dots$. The shift function $\sigma : \Sigma^{\mathbb{Z}} \rightarrow \Sigma^{\mathbb{Z}}$ is defined for every $w \in \Sigma^{\mathbb{Z}}$ and for every $n \in \mathbb{Z}$, by $\sigma(w)(n) = w(n-1)$. Given a language $L \subseteq A^{\mathbb{Z}}$, $\sigma(L) = \{\sigma(w) \mid w \in L\}$. L is said to be shift invariant if $L = \sigma(L)$. Shift invariance basically means that the instant 0 (the “origin” for ω-words) has no special role. Finite automata and linear temporal logic may only define shift invariant languages.

2.1 Automata on bi-infinite words

An automaton A is a five-tuple (Q, Σ, T, I, F), where Q is a finite set of *states*, $T \subseteq Q \times \Sigma \times Q$ is the set of *transitions*, $I \subseteq Q$ is the set of *initial* states and $F \subseteq Q$ is the set of *final* states. Two transitions (q, a, q'), (p, b, p') are consecutive if $q' = p$. A *bi-infinite path* of A is a bi-infinite sequence of consecutive transitions. A path is *successful* if $q_n \in I$ for infinitely many $n \leq 0$, and if $q_n \in F$ for infinitely many $n \geq 0$. For a path $\dots(q_{-1}, a_{-1}, q_0)(q_0, a_0, q_1)(q_1, a_1, q_2)\dots$, define its label as the bi-infinite word $\dots a_{-1}a_0a_1\dots$. The language $L(A)$ of A is the set of labels of successful bi-infinite paths of A. It is easy to see that $L(A)$ is shift-invariant.

A bi-infinite run is a bi-infinite word $\dots q_{-1}q_0q_1q_2\dots$ on $Q^{\mathbb{Z}}$ such that there exists a bi-infinite path $\dots(q_{-1}, a_{-1}, q_0)(q_0, a_0, q_1)(q_1, a_1, q_2)\dots$ of A. The run is successful if it corresponds to a successful path. In this paper, we are rarely interested in the language of A, but rather we remove the input alphabet and add both a finite set Ap of boolean propositions and an evaluation function $S : Q \rightarrow 2^{Ap}$. T is then a subset of $Q \times Q$. S indicates the set of propositions that are true in a state. For simplicity, we still call this structure (Q, Ap, S, T, I, F) a bi-infinite automaton (although it is a Kripke structure with bi-infinite fairness constraints). This form is especially convenient for model checking. Given a run $\dots q_{-1}q_0q_1q_2\dots$, the corresponding sequence of assignments $\dots S(q_{-1})S(q_0)S(q_1)S(q_2)\dots$ is denoted with $\dots S_{-1}S_0S_1S_2\dots$. For simplicity, each S_i may also be called a *state* of the automaton. No confusion can arise since one can always assume, by extending Ap, that $S(q) = S(q')$ if, and only if, $q = q'$.

2.2 A temporal logic on bi-infinite time

We define here Linear Temporal Logic with past operators (PLTL), in the version first introduced by Kamp [12]. However, rather than using more traditional ω-words, semantics will be defined on $\mathbb{Z}$-words.

Syntax of PLTL The alphabet of PLTL includes: a finite set Ap of propositional letters; two propositional connectives $\neg, \vee$ (from which other traditional connectives such as $\top, \bot, \neg, \vee, \wedge, \rightarrow, \dots$ may be defined); four temporal operators (from which other temporal operators can be derived): the "until" operator $\mathcal{U}$, the "next-time" operator $\circ$, the "since" operator $\mathcal{S}$ and the "past-time" (or Yesterday) operator, $\bullet$. Formulae are defined in the usual inductive way: a propositional letter $p \in Ap$ is a formula; $\neg\phi$, $\phi \vee \psi$, $\phi\mathcal{U}\psi$, $\circ\phi$, $\phi\mathcal{S}\psi$, $\bullet\phi$, where ϕ, ψ are formulae, are formulae; nothing else is a formula.

The traditional eventually and globally operators may be defined as: $\Diamond\phi$ is $\top\mathcal{U}\phi$, $\Box\phi$ is $\neg\Diamond\neg\phi$. Their past counterparts are: $\blacklozenge\phi$ is $\top\mathcal{S}\phi$, $\blacksquare\phi$ is $\neg\blacklozenge\neg\phi$. Another useful operator for PLTL is the Always operator $\mathcal{A}lw$, which can be defined by $\mathcal{A}lw\ \phi := \Box\phi \wedge \blacksquare\phi$. The intended meaning of $\mathcal{A}lw\ \phi$ is that ϕ must hold in every instant in the future and in the past. Its dual is the Sometimes operator $\mathcal{S}om\ \phi$ defined as $\neg\mathcal{A}lw\neg\phi$.

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

$w, i \models p, \iff p \in w(i)$, for $p \in Ap$
$w, i \models \neg\phi \iff w, i \not\models \phi$
$w, i \models \phi \vee \psi \iff w, i \models \phi$ or $w, i \models \psi$
$w, i \models \circ\phi \iff w, i+1 \models \phi$
$w, i \models \phi\mathcal{U}\psi \iff \exists k \geq 0 \mid w, i+k \models \psi,$ and $w, i+j \models \phi\ \forall 0 \leq j < k$
$w, i \models \bullet\phi \iff w, i-1 \models \phi$
$w, i \models \phi\mathcal{S}\psi \iff \exists k \geq 0 \mid w, i-k \models \psi,$ and $w, i-j \models \phi\ \forall 0 \leq j < k$

2.3 A bi-infinite encoding

In [23] we defined how PLTL formulae may be encoded into boolean formulae. The encoding includes additional information on the finite structure over which a PLTL formula is interpreted, so that the resulting boolean formula is satisfied in the finite structure if and only if the original PLTL formula is satisfied in a finite or possibly bi-infinite structure. Our encoding is essentially a bi-infinite generalization of a classical mono-infinite BMC encoding (see e.g. [3]). The interested reader can find its complete description in [23].

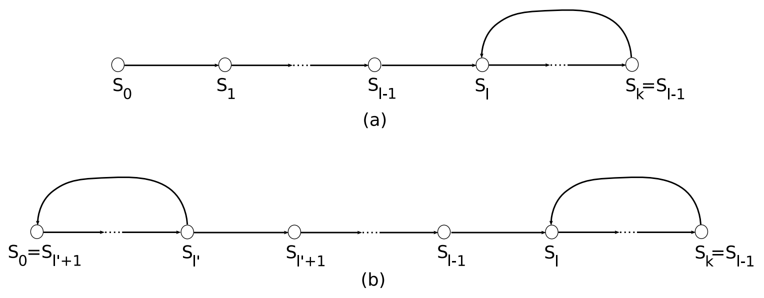

Fig. 1. a) Mono-infinite and b) bi-infinite bounded paths.

The idea on which the encoding is based is graphically depicted in Figure 1(b). A ultimately-periodic bi-infinite structure has a finite representation that includes a non-periodic portion, and two periodic portions corresponding to two cycles that are encoded by having two pairs of equal states in the sequence: when the interpreter of the formula (in our case, a SAT solver), needs the truth value of a subformula at a state beyond the last state S_k, it follows a “backward link” (resp., “forward link”) and considers the states S_l, S_{l+1}, ... as the states following S_k. Analogously, when it is necessary to evaluate a subformula before the first state S_0, then the interpreter follows a “forward link” and considers the states $S_{l'}$, $S_{l'-1}$, ... as the states preceding S_0.

Coming to the automaton encoding, to perform bounded *model* checking, we represent symbolically the transition relation of the system M as a propositional formula, where the states are represented as bit vectors. The k-times unrolling of the transition relation represents all the finite paths of length k:

$$|[M]|_k \iff \bigwedge_{0\le i<k} T(S_i, S_{i+1})$$

where T is a total transition relation predicate. Notice that, being M a bi-infinite automaton, there is no initial state predicate.

2.4 Metric temporal operators

PLTL can also 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 PLTL, although it does not actually extend the expressive power of PLTL.

Metric PLTL extends the alphabet of PLTL with a *bounded until* operator $\mathcal{U}_{\sim c}$ and a *bounded since* operator $\mathcal{S}_{\sim c}$, where $\sim$ represents any relational operator (i.e., $\sim\in \{\le, =, \ge\}$), and c is a natural number. Also, we allow n-ary predicate letters (with $n \ge 1$) 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 and bounded eventually operators are defined as follows: $\Diamond_{\sim c}\phi$ is $\top\mathcal{U}_{\sim c}\phi$, $\Box_{\sim c}\phi$ is $\neg\Diamond_{\sim c}\neg\phi$. The past versions of the bounded eventually and globally operators may be defined symmetrically to their future counterparts: $\blacklozenge_{\sim c}\phi$ is $\top\mathcal{S}_{\sim c}\phi$, $\blacksquare_{\sim c}\phi$ is $\neg\blacklozenge_{\sim c}\neg\phi$.

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.

The semantics of Metric PLTL may be defined by a straightforward translation τ of its operators into PLTL:

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

and symmetrically for the operators in the past.

Hence, in what follows we will consider Metric PLTL as a syntactically-sugared, but considerably more succinct, version of PLTL.

3 Bi-infinite time: a short motivation

It is widely recognized that allowing past operators in temporal logic, as in PLTL, makes it possible to write specifications that are easier, shorter, and, in some significant cases even exponentially more succinct than LTL specifications [16]. However, the ω-word semantics of PLTL is asymmetric: past is treated differently from future. Asymmetry in itself may seem a minor glitch, but it entails a problem: only a conventional value is returned when the evaluation of past operators requires time instants before the origin. For instance, consider the $\bullet$ operator: in a mono-infinite time domain, when $\bullet\phi$ is evaluated at instant $i > 0$, it returns the value of ϕ at instant $i - 1$; if $i = 0$, $\bullet\phi$ is conventionally evaluated to false. This ω-word semantics may easily lead to subtle specification errors, since natural, "expected" properties of the temporal operators are violated. These problems are usually "fixed" by allowing two dual forms of the $\bullet$ operator, the second one being defined to the default value true when its argument cannot be evaluated. This issue is even worsened when considering metric time operators, to be used for specifying real-time systems, since they may very easily refer to non-existent time instants in the past. For example, one may describe a system having a fixed cycle of operation of m time units by the formula $\Box(shutdown \leftrightarrow \blacklozenge_{=m}startup)$ (i.e., a shutdown occurs if and only if a startup took place m time units before), which could be rewritten (e.g., as a consequence of some automatic transformation performed by some tool that analyzes it) in the following, supposedly equivalent form:

$$\Box\begin{pmatrix}(\neg shutdown \vee \blacklozenge_{=m}startup) \\ \wedge \\ (shutdown \vee \blacklozenge_{=m}\neg startup)\end{pmatrix}.$$

Unfortunately, this latter, simple specification is unsatisfiable on a mono-infinite time domain, because in the first $m-1$ instants of the domain, both $shutdown$ and $\neg shutdown$ must be true, since both $\blacklozenge_{=m} startup$ and $\blacklozenge_{=m} \neg startup$ are (conventionally) false. This effect is dependent on the syntax used: for instance, if the lower subformula $(shutdown \vee \blacklozenge_{=m} \neg startup)$ is written in the apparently equivalent form: $shutdown \vee \neg \blacklozenge_{=m} startup$, then the behavior becomes the intended one, because in this case the conventional false value for $\blacklozenge_{=m}$ makes $shutdown \vee \neg \blacklozenge_{=m} startup$ true in the first $m-1$ instants. Clearly, these subtle semantics issues may easily escape notice in a more complex specification.

By adopting bi-infinite time, where event sequences may extend indefinitely in the past, past operators have a simple semantics that is symmetrical to that of the corresponding future-time operators: they are always defined and there is no need to use conventional values. 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 exactly once). Hence, one may use a convenient bi-infinite semantics even when specifying a mono-infinite system.

Throughout our past research, we have heavily dealt with temporal logic specifications and their application to industrial, critical real-time systems [5, 20, 17, 4]. Our approach has focused on using TRIO (a first order, linear-time temporal logic with a quantitative metric on time) for requirements specifications, without relying on machine models such as automata. One of the main features of TRIO is its ability to deal with different time domains: dense or discrete, finite or infinite [18]. In particular, most TRIO specifications adopt a bi-infinite time domain, using both future and past time operators. The application of the BMC techniques to a decidable fragment of TRIO was one of our original motivations for dealing with bi-infinity.

4 The Zot toolkit

Zot is an agile and easily extendible bounded model and satisfiability checker, which can be downloaded at http://home.dei.polimi.it/pradella/, together with the case studies and results 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 [9] is defined (essentially, equivalent to Metric PLTL). An interesting feature of Zot is its ability to support different encodings of temporal logic as SAT problems by means of plugins. This approach encourages experimentation, as plugins are expected to be 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 [3] are supported, a dense-time variant of MTL [8], and the bi-infinite encoding presented in [23].

Zot offers three 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.

2. *Bounded model checking (BMC)*: given as input an operational model of the system, the tool returns a (possibly empty) history (i.e., an execution trace of the specified system) which satisfies it.
3. *History checking and completion (HCC)*: The input 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 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/BMC modalities 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 counterexample, explaining why prop does not hold.

The tool and its plugins were validated on mono-infinite examples, such as the Mutex examples included in the distribution of NuSMV. The results were exactly the same as those obtained by using NuSMV [6] with the same encoding. On one hand, Zot is in general slower than NuSMV, but being quite small and written in *Common Lisp* is quite flexible, and promotes experimentation with different encodings and logic languages. On the other hand, in practice its performances are usually acceptable, because for non-trivial verifications the bottleneck typically resides in the SAT solver rather than in the translator.

Zot supports the model checkers MiniSat [7], zChaff, [21], and the recent multi-threaded MiraXT solver [15].

5 Case studies and experiments

To assess the actual feasibility of our approach, we applied it to some significant case studies, illustrated in the following sections. For all examples we apply the tool with reference to both mono- and bi-infinite structure, and then compare the results by computing the ratio of the various (time and memory) figures obtained in the two cases. We point out that some of the following case studies are framed as bounded *satisfiability* checking problems (because the analyzed system is described by means of a set of PLTL formulas without any constraint on their structure, and in particular on the nesting of their temporal operators): this is the case of the In/out channel, the Kernel Railway Crossing, and the Fischer's protocol. Three case studies are instead expressed as bounded *model* checking problems (the analyzed system is modeled through a set of PLTL formulae that are at all similar to a finite state automaton, because they relate the system current state with its next state): the In/Out channel (the only one to be considered in both ways), the simple mutual exclusion protocol, and the Real-time allocator.

5.1 A simple In/Out channel

The simplest example on which we tested the tool is that of a transmission line where any message entering at one end (represented by the predicate letter in) at any time is emitted at the other end (predicate out) after k time units. No message is lost nor is generated spuriously, so the transmission line is described by the formula:

$$\mathcal{A}lw(in \leftrightarrow \Diamond_{=k} out).$$

We considered two possible values for the delay k, namely, 5 and 15 time units. Moreover, to allow for bounded *model* checking, besides the above "descriptive" formalization, we used also a different, "operational" characterization of the transmission line system, composed of constraints that refer only to "current" and the "next" time instants. This requires the introduction of a counter to, which starts at value k when in holds, and is then decremented at each successive time instant, until out holds. In the tables reporting the experimentation results four versions of this examples are considered, corresponding to the descriptive or state-based style (suffix "d" or "s") and to the time bounds (5 or 15). For this simplest example the tool was only used to generate a possible trace of execution, as opposed to the other examples, for which we also carried out the proof of a few selected properties.

5.2 Kernel Railway Crossing

The Railway Crossing problem is a standard benchmark in real time systems verification [11]. It considers a railway crossing composed of a sensor, a gate and a controller. When a train is sensed to approach the crossing, a signal is sent to the controller. The controller then sends a command to the gate, closing the railway crossing to cars. The system operates in real time, ensuring safety (when the train is inside the railway crossing then the bar gate is closed) while maximizing utility (the bar should be open as long as possible). To this goal, there are various assumptions on the minimum and maximum speed of trains (e.g., the minimum time it takes for a train to enter the crossing after being sensed) and on the bar speed (the time it takes for the bar to be moved up or down). The Kernel Railroad crossing problem is a simplified version, where there is only one track and hence only one train at a time may enter the crossing. The goal of the KRC specification is twofold: a formal definition of the KRC system, and the proof of the safety and utility properties.

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 to model safety or utility properties.

In our example we studied the KRC problem with two different sets of constants, calling the two cases KRC1 and KRC2. Satisfiability of the specification, a safety property and a utility property were considered for the experiments. A complete specification, composed of a dozen axioms, of KRC1 and KRC2 and their properties can be found in [19].

5.3 Fischer's protocol

As a third case study, we consider Fischer's algorithm [13]. Fischer's is a timed mutual exclusion algorithm that allows a number of timed processes to access a shared resource. These processes are usually described as timed automata, and are often used as a benchmark for timed automata verification tools.

We considered a pure-logic description of the system in two variants. The first one, called Fischer1, considers 2 processes with a delay after the request of 3 time units. The second one, called Fischer2, considers 5 processes with a delay after the request of 6 time units.

We used the tool to check the safety property of the system (*safety-m* and *safety-b* in the tables of the following section), i.e. it is never possible that two different processes enter their critical sections at the same time instant.

As a last test for this system, we added a constraint to generate a behavior where there is always at least an alive process (*alive-m* and *alive-b* in the tables).

5.4 Simple mutual exclusion protocol

The fourth case study is a simple Mutual exclusion protocol for two processes, originally found in the distribution of NuSMV, which is called Mutex1. This was also extended to consider mutual exclusion with three processes, a model called Mutex2. Both examples have been defined with an automaton model.

For Mutex1 we considered the following property (where, quite naturally, $\mathrm{turn} = i$ means that it is the turn of process i):

$$\mathcal{A}lw \begin{pmatrix} (\mathrm{turn} = 1 \to \Diamond(\mathrm{turn} = 2))\wedge \\ (\mathrm{turn} = 2 \to \Diamond(\mathrm{turn} = 1)) \end{pmatrix}.$$

This is the variant of the previous property that we considered for Mutex2:

$$\mathcal{A}lw \begin{pmatrix} (\mathrm{turn} = 1 \to \Diamond(\mathrm{turn} = 2 \vee \mathrm{turn} = 3)) \wedge \\ (\mathrm{turn} = 2 \to \Diamond(\mathrm{turn} = 1 \vee \mathrm{turn} = 3)) \wedge \\ (\mathrm{turn} = 3 \to \Diamond(\mathrm{turn} = 2 \vee \mathrm{turn} = 3)) \end{pmatrix}.$$

5.5 Real-time allocator

The last case study consists of a real-time allocator which serves a set of client processes, competing for a shared resource. The system is a purely operational version of the one presented in [23].

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.

Two cases were considered in the following experiments: Alloc1 is the allocator model with two processes and $T_{rel} = 2$ and $T_{req} = 3$. Alloc2 is the allocator model with two processes and $T_{rel} = 4$ and $T_{req} = 5$.

Two hard real time properties $p1$ and $p2$ of Alloc1 and Alloc2 are considered in the experiments.

The first is a simple fairness property $p1$. 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:

$$(p1): \begin{array}{c} \mathcal{A}lw\left(\mathrm{rq}(p) \wedge \Box_{\leq T_{req}} \neg \mathrm{gr}(p) \rightarrow \Diamond_{=T_{req}} \mathrm{rq}(p)\right) \\ \rightarrow \\ \mathcal{A}lw\left(\mathrm{rq}(p) \rightarrow \Diamond \mathrm{gr}(p)\right) \end{array}$$

A second, more complex property may be intuitively described as a sort of "conditional fairness". Let us first 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 round 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*, this *conditional fairness* property $p2$ may be stated as:

$$(p2): \mathrm{UNROT} \rightarrow \left(\begin{array}{c} \mathcal{A}lw(\mathrm{rq}(p) \wedge \Box_{\leq T_{req}} \neg \mathrm{gr}(p) \rightarrow \Diamond_{>0} \mathrm{rq}(p)) \\ \rightarrow \\ \mathcal{A}lw(\mathrm{rq}(p) \rightarrow \Diamond_{>0} \mathrm{gr}(p)) \end{array}\right)$$

By careful inspection, however, it can be found that in the mono-infinite case $p2$ is only *vacuously true*, i.e., it corresponds to a run where no event occurs. In fact, the property of unconstrained rotation, in the simple form of the above UNROT formula, implies that any nonempty sequence of request events (and corresponding grant and release) goes back indefinitely towards the past. Therefore it can be satisfied non vacuously (i.e., with reference to behaviors that effectively include some events) only over a structure which is infinite in the past.

5.6 Summary of experimental results

We report here and comment on the results of applying the tool to the selected benchmarks and case studies. The experiments were run on a PC equipped with AMD Athlon 64 X2 4600+, 2 GB RAM, Linux OS. The SAT solver was MiniSat [7], version 2, along with SAT2CNF, part of the Alloy Analyzer (http://alloy.mit.edu).

For most examples we considered both a mono-infinite and a bi-infinite time structure, trying various bounds T on the size of the structure: 30, 60, 120, and 240 time units. For every example, the first basic experiment is checking satisfiability (non-emptiness) of the specification, without considering any property. This operation is useful as a sort of "sanity check" for a temporal logic specification, since it ensures that at least the formula is not contradictory. For all examples, except for the simplest In/Out

Case	Prop	Translation time (s)				Ratio b/m		SAT time (s)				Ratio b/m		SAT memory (MB)				Ratio b/m		Kilo-Clauses (#)				Ratio b/m	
		T=30	T=60	T=120	T=240	AVER	STDEV	T=30	T=60	T=120	T=240	AVER	STDEV	T=30	T=60	T=120	T=240	AVER	STDEV	T=30	T=60	T=120	T=240	AVER	STDEV
io5d	sat-m	0.84	0.95	2.32	9.10	2.07	0.81	0.06	0.18	0.52	1.70	2.63	0.35	5.71	9.22	16.89	31.78	1.73	0.09	22.5	44.7	89.0	177.5	1.88	0.00
	sat-b	0.83	1.84	6.28	24.26			0.15	0.41	1.30	5.20			9.15	16.35	28.99	57.48			42.2	83.9	167.3	334.1		
io5s	sat-m	0.93	1.77	6.71	27.13	1.86	0.38	0.15	0.38	1.20	4.29	2.10	0.21	8.39	13.95	26.49	51.04	1.64	0.08	37.7	74.7	148.8	296.9	1.67	0.00
	sat-b	1.24	3.82	14.35	48.39			0.27	0.80	2.54	10.07			12.84	23.58	43.11	86.63			62.9	125.0	249.2	497.6		
io15d	sat-m	0.78	1.80	7.40	27.43	2.91	0.48	0.14	0.40	1.78	5.32	2.76	0.39	8.26	13.81	26.34	50.80	2.05	0.17	37.2	73.8	146.9	293.0	2.20	0.01
	sat-b	1.81	6.29	21.83	78.81			0.38	1.17	3.95	16.61			14.98	28.36	56.43	111.55			81.6	162.3	323.7	646.5		
io15s	sat-m	2.63	9.93	38.72	151.28	1.95	0.09	0.52	1.66	6.07	27.10	2.32	0.17	17.25	30.59	59.58	121.95	1.69	0.07	90.4	179.0	356.3	710.7	1.71	0.00
	sat-b	5.40	19.72	71.73	286.63			1.12	3.72	14.88	67.02			27.44	53.78	103.29	206.52			154.1	306.2	610.4	1218.8		
KRC1	sat-m	11.17	41.45	154.87	649.42	2.24	0.05	2.11	8.48	30.95	134.37	2.46	0.25	40.29	78.69	151.80	301.27	1.61	0.03	211.9	420.3	837.2	1670.8	1.61	0.00
	sat-b	25.14	90.70	356.96	1433.83			4.79	18.77	82.36	359.66			63.91	125.57	243.41	497.14			341.4	678.2	1351.9	2699.2		
	safety-m	11.98	41.39	153.15	605.17	2.25	0.13	0.17	0.36	0.70	1.39	na	na	u.p.	u.p.	u.p.	u.p.	na	na	215.0	426.5	849.5	1695.3	na	na
	safety-b	25.00	91.74	363.32	1408.93			0.27	0.55	1.04	2.39			u.p.	u.p.	u.p.	u.p.			346.8	689.0	1373.4	2742.3		
	utility-m	12.99	46.60	174.97	697.04	2.30	0.09	2.85	19.23	137.56	1582.58	1.00	0.73	42.95	85.96	170.70	416.45	1.51	0.17	232.1	460.5	917.3	1830.8	1.58	0.00
	utility-b	28.43	105.39	411.40	1658.37			5.54	21.70	91.67	397.14			69.07	135.78	271.64	525.92			367.1	729.4	1454.0	2903.1		
KRC2	sat-m	42.11	161.26	622.03	2425.10	2.51	0.02	7.74	32.47	133.44	588.03	2.49	0.50	78.11	151.51	304.27	610.79	1.49	0.40	415.9	824.9	1643.0	3279.2	1.70	0.00
	sat-b	106.67	402.95	1555.55	6110.06			22.35	83.45	365.99	1044.31			133.29	250.52	516.16	543.24			706.2	1403.4	2797.8	5586.6		
	safety-m	41.89	155.59	600.79	2365.76	2.58	0.02	0.38	0.70	1.43	2.81	na	na	u.p.	u.p.	u.p.	u.p.	na	na	419.0	831.1	1655.3	3303.8	na	na
	safety-b	107.30	404.06	1563.68	6098.64			0.56	1.14	2.34	2.75			u.p.	u.p.	u.p.	u.p.			711.7	1414.3	2819.4	5629.7		
	utility-m	51.72	185.95	731.44	2873.39	2.55	0.03	10.52	44.93	201.68	1628.47	1.92	0.77	86.20	172.32	337.07	686.50	1.44	0.41	465.6	923.8	1840.4	3673.5	1.66	0.00
	utility-b	129.53	478.12	1887.22	7360.83			26.73	100.00	430.40	1297.75			142.22	283.35	551.00	568.61			770.7	1531.6	3053.3	6096.7		
Fischer1	sat-m	11.52	38.11	150.29	565.13	2.06	0.11	2.12	6.72	27.77	114.49	2.36	0.26	35.30	66.98	132.55	271.89	1.54	0.01	194.9	386.2	768.9	1534.1	1.56	0.00
	sat-b	21.91	79.54	314.89	1227.55			4.18	16.78	68.79	285.19			53.93	103.33	205.23	421.84			302.6	600.7	1196.9	2389.4		
	safety-m	11.39	41.26	154.91	612.97	2.18	0.07	0.17	0.34	0.66	1.38	na	na	u.p.	u.p.	u.p.	u.p.	na	na	208.4	413.0	822.2	1640.6	na	na
	safety-b	24.02	88.45	348.48	1361.90			4.76	18.29	78.31	333.12			58.74	116.37	225.58	450.66			323.1	641.4	1277.9	2551.0		
	alive-m	11.39	42.80	156.82	625.52	2.12	0.06	2.04	7.29	30.25	125.93	2.39	0.11	38.02	74.49	143.47	286.62	1.53	0.01	205.5	407.2	810.7	1617.5	1.56	0.00
	alive-b	24.04	87.52	344.21	1338.79			4.54	17.26	74.43	312.82			57.78	113.88	220.54	440.45			320.0	635.4	1266.0	2527.4		
Fischer2	sat-m	98.47	373.82	1447.75	5793.60	2.30	0.03	19.11	77.60	339.66	431.07	1.98	0.91	113.47	220.43	441.06	465.79	1.40	0.36	639.5	1267.3	2522.9	5034.2	1.59	0.00
	sat-b	228.20	867.98	3342.28	13130.04			48.68	192.61	212.68	971.33			177.14	346.37	378.32	752.75			1017.8	2020.8	4026.7	8038.7		
	safety-m	100.65	387.13	1513.45	5944.03	2.34	0.02	0.55	1.09	2.20	3.04	na	na	u.p.	u.p.	u.p.	u.p.	na	na	663.6	1315.0	2617.9	5223.7	na	na
	safety-b	238.81	908.40	3518.50	13884.41			52.54	212.86	516.70	2223.07			52.54	363.19	386.17	809.12			1053.5	2091.7	4168.1	8320.8		
	alive-m	98.39	393.15	1578.25	6030.94	2.29	0.06	23.69	84.98	348.01	1077.19	1.72	0.75	117.71	229.44	461.21	474.39	1.41	0.39	655.3	1298.6	2585.3	5158.7	1.60	0.00
	alive-b	231.08	905.03	3483.97	13779.06			51.77	199.56	234.09	1814.90			181.88	356.96	384.31	805.50			1045.0	2075.0	4134.8	8254.5		
Mutex1	sat-m	0.61	1.09	2.71	8.91	1.17	0.08	0.07	0.17	0.45	1.38	1.33	0.07	6.07	9.83	17.09	33.07	1.15	0.05	29.0	57.6	114.9	229.5	1.22	0.00
	sat-b	0.65	1.30	3.23	11.16			0.08	0.22	0.61	1.92			6.66	11.42	20.72	36.99			35.3	70.2	140.2	280.1		
	safety-m	0.84	1.78	6.14	22.96	1.49	0.10	0.11	0.31	0.98	3.42	1.67	0.09	9.45	16.96	30.54	59.73	1.31	0.06	45.5	90.4	180.3	359.9	1.35	0.00
	safety-b	1.14	2.63	9.84	34.85			0.18	0.51	1.62	6.10			12.00	21.07	40.48	82.95			61.4	122.2	243.8	486.9		
	safetyb-b	1.49	4.57	16.58	59.62	na	na	0.41	1.13	3.36	12.27	na	na	17.70	34.02	64.37	123.90	na	na	79.2	157.4	313.9	626.9	na	na
Mutex2	sat-m	0.90	1.94	5.52	19.63	1.16	0.04	0.20	0.64	8.16	38.34	0.94	0.42	11.77	22.00	41.35	82.94	1.16	0.04	47.8	95.1	189.7	379.0	1.18	0.00
	sat-b	1.08	2.27	6.35	21.62			0.25	0.87	4.66	22.70			13.23	25.63	49.86	94.80			56.4	112.5	224.6	448.7		
	safety-m	1.37	3.43	12.67	46.75	1.48	0.11	0.90	3.54	15.85	121.80	1.53	0.32	16.28	31.43	61.98	148.55	1.27	0.03	71.7	142.5	284.1	567.3	1.31	0.00
	safety-b	1.86	5.51	19.19	67.86			1.02	5.55	30.33	184.39			20.62	38.66	80.62	193.47			93.9	186.8	372.7	744.5		
	safetyb-b	2.72	9.09	31.75	118.62	na	na	3.38	6.50	27.41	139.18	na	na	26.01	51.04	100.16	191.29	na	na	119.9	238.5	475.6	949.9	na	na
Alloc1	sat-m	9.04	20.25	55.21	152.30	1.02	0.03	3.15	5.35	14.63	35.58	0.96	0.24	74.32	142.14	289.20	587.30	1.05	0.02	292.3	583.8	1166.8	2332.9	1.05	0.00
	sat-b	9.06	21.14	55.11	159.94			2.26	5.17	12.78	45.91			77.89	151.51	298.32	608.94			306.3	611.9	1223.2	2445.8		
	p1-m	10.45	29.10	83.05	268.08	1.47	0.16	17.25	118.30	130.66	333.55	1.60	1.45	80.61	164.05	318.28	633.13	1.13	0.03	329.0	656.4	1311.1	2620.7	1.17	0.00
	p1-b	13.37	41.75	126.56	445.51			16.45	55.68	164.06	1242.65			91.18	179.84	365.02	732.25			386.2	770.7	1539.7	3077.6		
	p2-m	13.52	38.95	123.12	428.37	1.54	0.13	3.06	7.82	23.69	81.61	2.25	0.66	88.73	173.63	352.50	688.97	1.15	0.02	373.2	744.3	1486.5	2970.9	1.19	0.00
	p2-b	18.52	59.22	195.62	721.61			4.70	14.49	69.10	219.49			100.05	203.07	405.83	800.21			443.9	885.3	1768.2	3533.9		
Alloc2	sat-m	20.35	42.66	102.33	264.48	1.02	0.05	7.76	16.83	35.70	1843.08	1.00	0.07	176.34	348.55	690.48	550.97	1.02	0.00	704.4	1407.7	2814.4	5627.8	1.03	0.00
	sat-b	19.60	44.18	103.55	285.31			7.17	16.58	38.69	1838.81			179.62	356.57	706.50	557.92			721.9	1443.0	2885.1	5769.5		
	p1-m	22.71	55.09	143.09	429.54	1.39	0.15	294.33	329.13	2608.76	2427.64	0.90	0.65	183.03	362.61	753.39	599.47	1.08	0.02	750.2	1498.5	2994.9	5987.8	1.10	0.00
	p1-b	27.79	72.55	208.52	668.03			117.82	402.99	803.27	3999.73			196.53	398.84	800.46	660.96			822.5	1642.8	3283.4	6564.7		
	p2-m	25.13	64.36	185.29	600.01	1.48	0.11	8.23	18.81	47.73	67.54	4.90	6.25	190.95	382.60	738.41	533.55	1.09	0.03	791.6	1580.8	3159.2	6315.9	1.10	0.00
	p2-b	34.23	92.07	289.16	949.60			10.58	31.25	114.95	962.54			202.80	411.22	824.39	597.55			872.4	1742.0	3481.2	6959.5		

Table 1. Summary of collected experimental data.

channel, we also proved a few selected properties: the property holds if and only if the tool answers UNSAT when applied to the specification conjoined with the negation of the property.

The entire collection of results is displayed in Table 1, which includes: translation time (dominated by the conjunctive normal form translation performed by SAT2CNF), SAT time and SAT memory (time and space taken by the SAT solver only), and the number of clauses in the formula generated by SAT2CNF. Translation time, closely related to the size of the original specification, changes in a quite regular way, and in our experiments it appears to be quadratically related to the bound T. SAT time is much less regular and predictable, as it may depend, in a very involved way, on the semantics of the specification and of the property being checked and on the details of the solver algorithms.

Certain data in Table 1 are denoted by **u.p.**. In these cases, MiniSat is able to determine unsatisfiability already during the parsing phase, using so-called unit propagation technique. SAT memory in this case cannot be computed, since the SAT solver has not really started a computation, and also SAT time is not very meaningful (it corresponds only to parsing time, which is negligible and only related to the size of the boolean formula fed to the solver). In two experiments, unit propagation occurred for both the bi-infinite and the mono-infinite case; in one experiment it occurred only in the mono-infinite case.

Table 1 also contains four columns labeled “ratio b/m”, which are more closely focused on the comparison between the figures for the mono and the bi-infinite case: for each pair of such data (for the same example and the same property) it reports the average, over the four values of T, of the ratio between the bi-infinite figure and the corresponding mono-infinite one, together with its standard deviation.

A few values were left out of these columns, since they correspond to cases where the comparison is not possible or would give misleading results:

– The property safety-b-b for both Mutex1 and Mutex2 is bi-infinite only, since it does not hold on a mono-infinite domain (and hence it is difficult to be compared).
– All occurrences of unit propagation were ignored for SAT time and SAT memory, since no meaningful measure can be used to make a comparison.

Also, Property $p2$ for the Allocator, as already pointed out, is only vacuously true on a mono-infinite structure, so the SAT-solver can very easily prove unsatisfiability. This explains the relatively large b/m ratio for SAT time.

As one can notice, standard deviations are typically small for all measures except SAT time, which, as expected, shows more volatility. Hence, for all measures, except for SAT time, the ratio is close to be a constant for the same case study, when considering different bounds.

Overall, results are satisfactory. All measures, including SAT time, show a ratio between 1 and 3, except in the above reported special cases. SAT time shows more volatility than other measures, but it is still bounded and occasionally the ratio can even go below 1, with 2 being the most typical value. Also, there does not appear to be any significant difference in the ratios between cases where the specification is purely operational (Allocator, Mutex, io5d, io15d) or purely logical (Fischer, KRC, io5s, io15d).

6 Conclusions

In this paper we have argued that bi-infinite time in specifications is a useful abstraction, allowing one to ignore the complexity of system initialization, and to express fairness properties also in the past.

Bi-infinite time has certainly been used before in specification. For instance, our own requirement specifications of industrial systems using TRIO temporal logic language [9] most often adopted bi-infinite time. However, we are not aware of any other work extending model checking to deal with bi-infinity, apart from our encoding of automata and PLTL formulae [23] that includes additional information to represent bi-infinite structures by means of finite ones having two cycles of states, one that unfolds in the future and one for the past.

Our Zot tool incorporates the bi-infinite encoding and, by relying on standard satisfiability checkers, supports a variety of analysis and verification activities.

This paper investigated the tool and its application to many case studies, ranging from simple to complex, in order to assess the feasibility of the approach, by comparing the performance of the same case when using a mono-infinite and a then a bi-infinite structure. The experimental results show that, on these examples, tool performance on bi-infinite structures is comparable to that on mono-infinite ones, suggesting that adopting a bi-infinite notion of time does not impose very significant penalties to the efficiency of bounded model checking and bounded satisfiability checking. On the other hand, bi-infinite time is more natural than mono-infinite time in many cases and it avoids subtle semantics problem with PLTL formulae.

Further work might consider various optimizations, such as incremental encodings, and also deal with completeness issues [24, 3]. These were ignored in this paper, where we applied a standard, relatively simple encoding technique, since we were mainly interested in comparing the performance of mono- and bi-infinite model checking.

References

1. M. Benedetti and A. Cimatti. Bounded model checking for past LTL. In *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. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.
3. 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.
4. 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.
5. 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.
6. 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.

7. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT Conference*, volume 2919 of *LNCS*, pages 502–518. Springer-Verlag, 2003.
8. C. A. Furia, M. Pradella, and M. Rossi. Dense-time MTL verification through sampling. In *Proceedings of FM'08*, volume 5014 of *LNCS*, 2008.
9. 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.
10. F. Gire and M. Nivat. Langages algébriques de mots biinfinis. *Theoret. Comput. Sci.*, 86(2):277–323, 1991.
11. C. Heitmeyer and D. Mandrioli. *Formal Methods for Real-Time Computing*. John Wiley & Sons, Inc., New York, NY, USA, 1996.
12. J. A. W. Kamp. *Tense Logic and the Theory of Linear Order (Ph.D. thesis)*. University of California at Los Angeles, 1968.
13. L. Lamport. A fast mutual exclusion algorithm. *ACM TOCS-Transactions On Computer Systems*, 5(1):1–11, 1987.
14. T. Latvala, A. Biere, K. Heljanko, and T. Junttila. Simple is better: Efficient bounded model checking for past LTL. In *Proceedings of VMCAI'2005*, volume 3385 of *Lecture Notes in Computer Science*, pages 380–395. Springer-Verlag, January 2005.
15. M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT solving. In *12th Asia and South Pacific Design Automation Conference*, 2007.
16. 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.
17. S. Morasca, A. Morzenti, and P. San Pietro. A case study on applying a tool for automated system analysis object oriented logic specification of time-critical systems. based on modular specifications written in TRIO. *Autom. Softw. Eng.*, 7(2):125–155, 2000.
18. A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14(4):521–573, 1992.
19. 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.
20. A. Morzenti and P. San Pietro. Object-oriented logical specification of time-critical systems. *ACM Trans. Softw. Eng. Methodol.*, 3(1):56–98, 1994.
21. 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.
22. D. Perrin and J.-É. Pin. *Infinite Words*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004. ISBN 0-12-532111-2.
23. M. Pradella, A. Morzenti, and P. San Pietro. The symmetry of the past and of the future: Bi-infinite time in the verification of temporal properties. In *Proc. of The 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ESEC/FSE*, Dubrovnik, Croatia, September 2007.
24. 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.