Efficient Scalable Verification of LTL Specifications

Luciano Baresi, Mohammad Mehdi Pourhashem Kallehbasti, and Matteo Rossi
Politecnico di Milano – Dipartimento di Elettronica, Informazione e Bioingegneria
Via Golgi 42 – 20133 Milano, Italy
Emails: {luciano.baresi,mohammadmehdi.pourhashem,matteo.rossi}@polimi.it

***Abstract*—Linear Temporal Logic (LTL) has been used in computer science for decades to formally specify programs, systems, desired properties, and relevant behaviors. This paper presents a novel, efficient technique for verifying LTL specifications in a fully automated way. Our technique belongs to the category of Bounded Satisfiability Checking approaches, where LTL formulae are encoded as formulae of another decidable logic that can be solved through modern satisfiability solvers. The target logic in our approach is Bit-Vector Logic. We present our novel encoding, show its correctness, and experimentally compare it against existing encodings implemented in well-known formal verification tools.**

I. Introduction

Linear Temporal Logic (LTL) has been used in computer science for decades [1]. Its applications include the specification and verification of (possibly safety-critical) programs and systems [2], test case generation [3], run-time verification [4], planning [5], and controller synthesis [6]. In addition, LTL —or its expressively equivalent variants— can be used as underlying formalism to capture the semantics of semi-formal notations like UML to perform formal verification on them, as we showed in [7].

However, for an LTL-based approach to be effective in practice, there must be techniques and tools that allow users to check big LTL specifications in a short amount of time. In this paper we focus on the problem of performing formal verification on systems described through a set of LTL formulae. These LTL descriptions could be the outcome of systematic requirements elicitation or formal specification, but they could also be the output of tools producing LTL formalizations from more informal languages such as UML. In fact, one of the driving factors that led us to pursue the work presented in this paper was the necessity of formally verifying larger UML models than those that can be managed by the tool introduced in [7].

Formal verification of LTL specifications can be carried out through tools capable of determining the satisfiability of sets of LTL formulae. Various techniques have been developed in the past, based for example on automata construction [8]. In this work we pursue an approach based on the notion of Bounded Satisfiability Checking [9], a variant of Bounded Model Checking [10] that focuses on the satisfiability of temporal logic formulae. In Bounded Satisfiability Checking approaches, LTL formulae are suitably translated into formulae of another decidable logic, such as Propositional Logic, which precisely capture ultimately periodic models of the original formulae of length up to a bound k; produced formulae are then fed to a solver for the target logic (e.g., a SAT solver) for verification.

The first question tackled by this paper is thus: *how can we analyze bigger models?* We studied a way to allow a Bounded Satisfiability Checker to work on both more complex formulae and larger bounds k. The proposed solution eschews the usual Propositional Logic as target logic, and encodes LTL formulae into formulae of Bit-Vector Logic, which can be efficiently analyzed by modern Satisfiability Modulo Theories (SMT) solvers such as the well-known Z3 [11]. We implemented the encoding in a plugin, called *bvzot*, of our $\mathbb{Z}$ot tool [12]. Even if the initial results were very encouraging, with respect to both improvements on previous versions of $\mathbb{Z}$ot and our goal of being able to analyze larger UML models, we wanted to move a step further and try to generalize them. Our second research question was then: *how can we compare these results against the state of the art in the field and understand how good in general they are?* This is why we started using NuSMV [13], which is one of the reference tools for Bounded Model Checking techniques. The paper thus presents the promising results we obtained by comparing the performance of NuSMV and *bvzot* on several case studies with different characteristics: two of them are well-known benchmarks, while a third one comes from one of our UML specifications, since we also wanted to relate these results to our original problem.

To summarize, the contributions of this paper are twofold. First, we introduce a novel translation of LTL formulae into formulae of Bit-Vector Logic. Second, we provide a systematic comparison of our verification tool with the reference tools in the field, which shows that *bvzot* compares favorably to them.

The rest of this paper is organized as follows. Section II provides a brief introduction to LTL, and presents the classic Propositional Logic-based encoding of LTL formulae, which is then used to demonstrate the correctness of our own encoding. Section III summarizes the key elements of Bit-Vector Logic and proposes the encoding we devised; in addition, it studies the correctness and complexity of our encoding. Section IV presents the toolset we used for evaluation, the experiments we carried out, and the results we obtained. Section V surveys related approaches, and Section VI concludes the paper.

II. Background

A. Linear Temporal Logic

LTL [1] is a widely-used specification logic. In this paper, we focus on the version with both future and past temporal operators. In fact, although past operators do not increase

the expressiveness of the logic, they are advantageous for compositional reasoning [14]. In addition, LTL with past operators is exponentially more succinct than its future-only counterpart [15].

An LTL formula is defined over a set of atomic propositions AP. The syntax of LTL is defined by the following grammar:

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

where $p \in AP$, X and U are the "next" and "until" future operators, whereas Y ("yesterday") and S ("since") are their past counterparts.

The semantics of LTL is given in terms of infinite sequences of sets of atomic propositions, or *words*. A word $\pi : \mathbb{N} \rightarrow 2^{AP}$ assigns, to every instant of the temporal domain $\mathbb{N}$, the (possibly empty) set of atomic propositions that hold in that instant. We can think of a word as an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$, where each state is labeled with the atomic propositions that hold in it. We say that a word *satisfies formula* ϕ *at instant* i, written $\pi, i \models \phi$, if ϕ holds when evaluated starting from instant i of π. The following is the usual formal semantics of the satisfiability relation for LTL:

$$\begin{array}{lcl}
\pi, i \models p & \Leftrightarrow & p \in \pi(i) \text{ for } p \in AP \\
\pi, i \models \neg\phi & \Leftrightarrow & \pi, i \not\models \phi \\
\pi, i \models \phi_1 \wedge \phi_2 & \Leftrightarrow & \pi, i \models \phi_1 \text{ and } \pi, i \models \phi_2 \\
\pi, i \models \mathrm{X}\phi & \Leftrightarrow & \pi, i+1 \models \phi \\
\pi, i \models \mathrm{Y}\phi & \Leftrightarrow & i > 0 \text{ and } \pi, i-1 \models \phi \\
\pi, i \models \phi_1 \mathrm{U} \phi_2 & \Leftrightarrow & \exists j \geq i \text{ s.t. } \pi, j \models \phi_2 \\
 & & \text{and } \forall n \text{ s.t. } i \leq n < j : \pi, n \models \phi_1 \\
\pi, i \models \phi_1 \mathrm{S} \phi_2 & \Leftrightarrow & \exists j \leq i \text{ s.t. } \pi, j \models \phi_2 \\
 & & \text{and } \forall n \text{ s.t. } j < n \leq i : \pi, n \models \phi_1
\end{array}$$

We say that a word π *satisfies* LTL formula ϕ when it holds in the first instant of the temporal domain, i.e., when $\pi, 0 \models \phi$. In this case we will sometimes write $\pi \models \phi$. A word π that satisfies ϕ is a *model* for ϕ.

Starting from the basic connectives and operators, it is customary to introduce the other traditional Boolean connectives ($\vee$, $\Rightarrow$, ...), and temporal operators as abbreviations. In particular the "eventually in the future" (F), "globally in the future" (G) and "release" (R) operators (and their past counterparts "eventually in the past" P, "historically" H and "trigger" T) are defined as follows:

$$\begin{array}{llll}
\mathrm{F}\phi & = \top\mathrm{U}\phi & \mathrm{P}\phi & = \top\mathrm{S}\phi \\
\mathrm{G}\phi & = \neg\mathrm{F}\neg\phi & \mathrm{H}\phi & = \neg\mathrm{P}\neg\phi \\
\phi_1\mathrm{R}\phi_2 & = \neg(\neg\phi_1\mathrm{U}\neg\phi_2) & \phi_1\mathrm{T}\phi_2 & = \neg(\neg\phi_1\mathrm{S}\neg\phi_2)
\end{array}$$

B. Bounded Satisfiability Checking

The principle that underlies both Bounded Model Checking (BMC) and Bounded Satisfiability Checking (BSC) techniques is, given formalizations of the system S and of the property ϕ that should hold for the system, to look for an infinite, ultimately periodic execution $\pi = s_0 s_1 \ldots s_{l-1}(s_l s_{l+1} \ldots s_k)^\omega$ of S that violates ϕ, where k is a parameter. If a counterexample witnessing the violation of the property exists, then the property does not hold for S. If no counterexample of length up to k is found, then the property holds for S provided that k is "big enough".

Both techniques can be used to check whether an LTL formula ϕ is satisfiable or not. At the core of both BMC and BSC is the idea of translating an LTL formula ϕ into a formula of Propositional Logic that represents ultimately periodic models of ϕ. Then, its verification is performed by feeding the translated formula to a solver.

In the following we briefly describe the classic technique for encoding LTL formulae into Propositional Logic introduced in [16], which is at the core of BSC. To this end, one only needs to represent states $s_0 \ldots s_l \ldots s_k$, and then the fact that the state after s_k, say s_{k+1}, is in fact s_l again. Hence, the bounded encoding captures finite sequences of states of the form $\alpha s \beta s$, where $\alpha = s_0 s_1 \ldots s_{l-1}$, $\beta = s_{l+1} \ldots s_k$, and $s = s_l = s_{k+1}$.

The encoding is defined as Boolean constraints over so-called *formula variables* $|[\psi]|_i$. These are Boolean variables which are used to represent the value of all subformulae of the LTL formula to be checked for satisfiability in instants $0, 1, \ldots k+1$. More precisely, given an LTL formula ϕ and a bound k, the encoding introduces, for each subformula ψ of ϕ, $k+2$ formula variables $|[\psi]|_0, |[\psi]|_1, \ldots |[\psi]|_{k+1}$ which capture whether ψ is true or not at the various instants in $[0, k+1]$.

In addition, the encoding introduces $k+1$ *loop selector variables* $l_0, l_1, \ldots, l_k$, which are fresh Boolean variables such that l_l is true iff the loop starts at position l (hence, if l_l is true, then $s_l = s_{k+1}$); at most one of $l_0, l_1, \ldots, l_k$ can be true. Other Boolean variables are introduced for convenience: the $k+1$ variables $InLoop_i$, with $0 \leq i \leq k$, are such that $InLoop_i$ is true iff position i is in the loop (i.e., $l \leq i \leq k$). Finally, variable $LoopExists$ is true iff the desired loop exists.

In the rest of this section we present the constraints that are imposed on the Boolean variables introduced above to capture the semantics of LTL formulae. The following table defines a set of constraints called $|LoopConstraints|_k$:

Base	$\neg l_0 \wedge \neg InLoop_0$
$1 \leq i \leq k$	$(l_i \Rightarrow s_{i-1} = s_k) \wedge (InLoop_i \Leftrightarrow InLoop_{i-1} \vee l_i)$ $\wedge (InLoop_{i-1} \Rightarrow \neg l_i) \wedge (LoopExists \Leftrightarrow InLoop_k)$

They essentially define the semantics of Booolean variables $\{l_i\}_{i\in[0,k]}$, $\{InLoop_i\}_{i\in[0,k]}$ and $LoopExists$ (e.g., the existence of at most one loop). In addition, as mentioned in [16], they impose that the same atomic propositions that hold in state s_k also hold in state s_{l-1}, which has been shown to improve the efficiency of the model search.

The following table shows $|LastStateConstraints|_k$.

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

They define that the subformulae of ϕ that hold in s_{k+1} are the same as those that hold in state s_l. This effectively defines that after state s_k the bounded trace loops back to state s_l.

The subsequent constraints define the semantics of the propositional connectives and of the temporal operators. The next table introduces the set of constraints $|PropConstraints|_k$, which capture the semantics of propositional connectives;

ϕ	$0 \leq i \leq k+1$
p	$\|[p]\|_i \Leftrightarrow p \in \pi(i)$
$\neg p$	$\|[\neg p]\|_i \Leftrightarrow p \notin \pi(i)$
$\psi_1 \wedge \psi_2$	$\|[\psi_1 \wedge \psi_2]\|_i \Leftrightarrow \|[\psi_1]\|_i \wedge \|[\psi_2]\|_i$
$\psi_1 \vee \psi_2$	$\|[\psi_1 \vee \psi_2]\|_i \Leftrightarrow \|[\psi_1]\|_i \vee \|[\psi_2]\|_i$

For example, $|PropConstraints|_k$ state that the value of $|[p]|_i$ and $|[\neg p]|_i$ capture whether propositional letter p holds at instant i or not. The definitions of $|[\psi_1 \wedge \psi_2]|$ and of $|[\psi_1 \vee \psi_2]|$ are straightforward. Notice that the Boolean encoding was defined for LTL formulae in Positive Normal Form (PNF), i.e., in which negations can only appear next to atomic propositions. This can save some formula variables, but the encoding can be easily generalized to formulae that are not in PNF.

The next tables define the semantics of the temporal operators, both future (X, U and R) and past ones (Y, S and T). We call this whole set of constraints $|TempConstraints|_k$.

The semantics of U and R is defined through their standard fixpoint characterization:

ϕ	$0 \leq i \leq k$
$\mathrm{X}\psi$	$\|[\mathrm{X}\psi]\|_i \Leftrightarrow \|[\psi]\|_{i+1}$
$\psi_1 \mathrm{U} \psi_2$	$\|[\psi_1 \mathrm{U} \psi_2]\|_i \Leftrightarrow \|[\psi_2]\|_i \vee (\|[\psi_1]\|_i \wedge \|[\psi_1 \mathrm{U} \psi_2]\|_{i+1})$
$\psi_1 \mathrm{R} \psi_2$	$\|[\psi_1 \mathrm{R} \psi_2]\|_i \Leftrightarrow \|[\psi_2]\|_i \wedge (\|[\psi_1]\|_i \vee \|[\psi_1 \mathrm{R} \psi_2]\|_{i+1})$

The definition of the semantics of U and R is completed through the introduction of the set of constraints $|Eventualities|_k$, which are presented in the following table:

Base	$\psi_1 \mathrm{U} \psi_2$	$LoopExists \Rightarrow (\|[\psi_1 \mathrm{U} \psi_2]\|_k \Rightarrow \langle\langle F\psi_2 \rangle\rangle_k)$
	$\psi_1 \mathrm{R} \psi_2$	$LoopExists \Rightarrow (\|[\psi_1 \mathrm{R} \psi_2]\|_k \Leftarrow \langle\langle G\psi_2 \rangle\rangle_k)$
	$\psi_1 \mathrm{U} \psi_2$	$\langle\langle F\psi_2 \rangle\rangle_0 \Leftrightarrow \bot$
	$\psi_1 \mathrm{R} \psi_2$	$\langle\langle G\psi_2 \rangle\rangle_0 \Leftrightarrow \top$
$1 \leq i \leq k$	$\psi_1 \mathrm{U} \psi_2$	$\langle\langle F\psi_2 \rangle\rangle_i \Leftrightarrow \langle\langle F\psi_2 \rangle\rangle_{i-1} \vee (InLoop_i \wedge \|[\psi_2]\|_i)$
	$\psi_1 \mathrm{R} \psi_2$	$\langle\langle G\psi_2 \rangle\rangle_i \Leftrightarrow \langle\langle G\psi_2 \rangle\rangle_{i-1} \wedge (\neg InLoop_i \vee \|[\psi_2]\|_i)$

These constraints are used to make sure that, if $\psi_1 \mathrm{U} \psi_2$ holds in s_k, then ψ_2 occurs infinitely often, that is, it occurs somewhere in the loop. Similarly, if $\psi_1 \mathrm{R} \psi_2$ occurs in s_k, then either ψ_2 holds throughout the loop, or at some point of the loop ψ_1 holds. $\langle\langle F\psi_2 \rangle\rangle_i$ and $\langle\langle G\psi_2 \rangle\rangle_i$ are auxiliary variables required for capturing these constraints. $\langle\langle F\psi_2 \rangle\rangle_i$ holds if position i belongs to the loop and ψ_2 holds in at least one position between l and i. Accordingly, $\langle\langle F\psi_2 \rangle\rangle_k$ means that ψ_2 holds somewhere in the loop. Therefore, constraint $LoopExists \Rightarrow (|[\psi_1 \mathrm{U} \psi_2]|_k \Rightarrow \langle\langle F\psi_2 \rangle\rangle_k)$ does not allow $\psi_1 \mathrm{U} \psi_2$ to hold at k, if ψ_2 does not occur infinitely often. Similarly, $\langle\langle G\psi_2 \rangle\rangle_k$ holds iff ψ_2 holds everywhere in the loop. Then, constraint $LoopExists \Rightarrow (|[\psi_1 \mathrm{R} \psi_2]|_k \Leftarrow \langle\langle G\psi_2 \rangle\rangle_k)$ forces $|[\psi_1 \mathrm{R} \psi_2]|_k$ to hold if ψ_2 holds from position l on.

The next table defines the semantics of the past operators Y, S and T, which is symmetrical to their future counterparts.

ϕ	$0 < i \leq k+1$
$\mathrm{Y}\psi$	$\|[\mathrm{Y}\psi]\|_i \Leftrightarrow \|[\psi]\|_{i-1}$
$\mathrm{Z}\psi$	$\|[\mathrm{Z}\psi]\|_i \Leftrightarrow \|[\psi]\|_{i-1}$
$\psi_1 \mathrm{S} \psi_2$	$\|[\psi_1 \mathrm{S} \psi_2]\|_i \Leftrightarrow \|[\psi_2]\|_i \vee (\|[\psi_1]\|_i \wedge \|[\psi_1 \mathrm{S} \psi_2]\|_{i-1})$
$\psi_1 \mathrm{T} \psi_2$	$\|[\psi_1 \mathrm{T} \psi_2]\|_i \Leftrightarrow \|[\psi_2]\|_i \wedge (\|[\psi_1]\|_i \vee \|[\psi_1 \mathrm{T} \psi_2]\|_{i-1})$

It also defines operator Z, which is necessary for formulae in PNF, which is simply a variant of Y such that $\mathrm{Z}\psi$ holds in 0 no matter ψ. Since the temporal domain is mono-infinite (i.e., it is infinite only towards the future), there is no need to impose eventuality constraints over past operators. However, we must define the value of past operators in the origin 0. This is done through the constraints of the following table.

ϕ	Base
$\mathrm{Y}\psi$	$\neg\|[\mathrm{Y}\psi]\|_0$
$\mathrm{Z}\psi$	$\|[\mathrm{Z}\psi]\|_0$
$\psi_1 \mathrm{S} \psi_2$	$\|[\psi_1 \mathrm{S} \psi_2]\|_0 \Leftrightarrow \|[\psi_2]\|_0$
$\psi_1 \mathrm{T} \psi_2$	$\|[\psi_1 \mathrm{T} \psi_2]\|_0 \Leftrightarrow \|[\psi_2]\|_0$

Finally, given an LTL formula ϕ, its Boolean encoding ϕ_{B} is given by the conjunction of the constraints in sets $|LoopConstraints|_k$, $|LastStateConstrants|_k$, $|PropConstraints|_k$, $|TempConstraints|_k$, and $|Eventualities|_k$, plus the statement that ϕ holds in the origin, i.e. $|[\phi]|_0$.

III. Bit-Vector-based Encoding

A. Bit-Vector Logic

This section briefly presents the operations on bit-vectors that are defined in Bit-Vector Logic and that we use in our encoding.

A bit-vector is an array whose elements are bits (Booleans). In Bit-vector Logic, the size of a bit-vector (number of bits) is finite, and can be any nonzero number in $\mathbb{N}$. For the bit-vector $\overleftarrow{x}$ with size n, we use the notation $\overleftarrow{x}_{[n]}$, or simply $\overleftarrow{x}$ when the size is not important or can be inferred from the context. Furthermore, $\overleftarrow{x}^{[i]}_{[n]}$ stands for the ith bit in the bit-vector $\overleftarrow{x}$, where bits are indexed from right to left. Accordingly, $\overleftarrow{x}^{[n-1]}_{[n]}$ is the leftmost and most significant bit, and $\overleftarrow{x}^{[0]}_{[n]}$ is the rightmost and least significant bit. For constants we use the notation $\overleftarrow{c}_{[n]}$, which is the two's complement representation of integer c over n bits, for example, $\overleftarrow{-2}_{[4]}$ is 1110.

Bit-Vector Logic offers a wide range of operators. The two core operators are *Concatenation* and *Extraction*. *Concatenation*: $\overleftarrow{x}_{[n]} :: \overleftarrow{y}_{[m]}$ is a bit-vector $\overleftarrow{z}_{[n+m]}$, such that $\overleftarrow{z}^{[0]} = \overleftarrow{y}^{[0]}$ and $\overleftarrow{z}^{[m+n-1]} = \overleftarrow{x}^{[n-1]}$. For example, $111 :: 0 = 1110$. *Extraction*: $\overleftarrow{x}^{[j:i]}$ is a bit-vector $\overleftarrow{z}_{[j-i+1]}$, where $\overleftarrow{z}^{[0]} = \overleftarrow{x}^{[i]}$ and $\overleftarrow{z}^{[j-i]} = \overleftarrow{x}^{[j]}$, which can be defined through concatenation as $\overleftarrow{x}^{[j:i]} = ::_{k=j}^{i} x^{[k]}$. For example, $1100^{[2:0]} = 100$.

Arithmetic operators *addition* ($+$) and *subtraction* ($-$) throw away the final carry bit and the resulting bit-vector has the same size as the operands. *Unsigned shift to the right/left* ($\gg$/$\ll$) throws away he rightmost/leftmost bit and inserts *zero* from the left/right. For example, $\gg 1100 = 0110$ and $\ll 1100 = 1000$.

We also use bitwise operators like *negation* (!), *conjunction* (&), *disjunction* ($|$), and *reduction and* ($\Downarrow$). The *reduction and* operator is defined as $\Downarrow \overleftarrow{x}_{[n]} = \&_{i=0}^{n-1} \overleftarrow{x}_{[n]}^{[i]}$ (i.e., it is the "and" of all the bits in $\overleftarrow{x}$). The size of the resulting bit-vector is one. The bit corresponds to the minimum value in $\overleftarrow{x}$; in other words, it is equal to *one* if all the bits of the bit-vector $\overleftarrow{x}$ are *one*, *zero* otherwise.

Bit-vectors (or parts thereof) can be compared using the usual relational operators $=$, $<$, and formulae of Bit-Vector Logic can be built using the usual Boolean connectives $\neg$, $\wedge$.

B. Encoding

Similarly to the classic Boolean encoding of Section II-B, our encoding uses a bit-vector of size $k+2$ to represent the truth values of each subformula of ϕ from 0 to $k+1$. However, we only introduce as many bit-vectors as the number of atomic propositions in the formula, and describe the values of the non-atomic subformulae as transformations on the former vectors. More precisely, for each $p \in AP$, we introduce a bit-vector, $\overleftarrow{p}_{[k+2]}$ (i.e., of size $k+2$), such that $\overleftarrow{p}_{[k+2]}^{[i]}$, with $i \in [0, k+1]$, captures the value of proposition p at instant i. Recall that $\overleftarrow{p}^{[0]}$ is the right-most (least significant) bit in $\overleftarrow{p}$, and $\overleftarrow{p}^{[k+1]}$ is the left-most (most significant) one.

In addition, we introduce a bit-vector, $\overleftarrow{loop}_{[k+2]}$, that contains (encoded in binary) the position of the loop in interval $[0, k+1]$ (i.e., the position of the first state s in $\alpha s \beta s$).

For the sake of uniformity in using Bit-Vector Logic operators to capture the semantics of LTL formulae, we encode $\perp$ (false) as $\overleftarrow{0}_{[k+2]}$ (i.e., a sequence of *zeros*) and $\top$ (true) as $\overleftarrow{-1}_{[k+2]}$ (i.e., a sequence of *ones*), so the size of all bit-vectors used in the encoding is $k+2$.

To introduce the bit-vector-based encoding of LTL formulae, it is useful to first define some auxiliary operators on bit-vectors that will be exploited in the following. These operators are defined below.

Operator	Definition
$\mathrm{Rev}\ \overleftarrow{x}$	$::_{i=0}^{k+1} \overleftarrow{x}^{[i]}$
$\overleftarrow{x}\,\mathrm{U_{nl}}\,\overleftarrow{y}$	$\overleftarrow{y} \mid (\overleftarrow{x}\ \&\ !\mathrm{Rev}(\mathrm{Rev}(\overleftarrow{x} \mid \overleftarrow{y}) + \mathrm{Rev}\overleftarrow{y}))$

$Rev\ \overleftarrow{x}$ reverses the order of the bits in bit-vector $\overleftarrow{x}$. $\overleftarrow{x}\,\mathrm{U_{nl}}\,\overleftarrow{y}$ produces a bit-vector, say $\overleftarrow{z}$, such that, if one takes $\overleftarrow{x}$ and $\overleftarrow{y}$ to represent the values of some formulae x, y in $[0, k+1]$, then $\overleftarrow{z}$ corresponds to the value of $x\mathrm{U}y$ when one considers only finite models (i.e., there is no loop, hence the subscript nl). Later in this section we give an example of computation of $\overleftarrow{x}\,\mathrm{U_{nl}}\,\overleftarrow{y}$.

In the bit-vector-based encoding of LTL, the bit-vector capturing the value of a formula ϕ in $[0, k+1]$ is obtained by recursively performing operations on the bit-vectors corresponding to the subformulae of ϕ. The operations performed depend on the structure of ϕ.

The next table shows the case in which the main connective in ϕ is a Boolean one.

ϕ	bit-vector encoding
$\neg\psi$	$!\overleftarrow{\psi}$
$\psi_1 \wedge \psi_2$	$\overleftarrow{\psi}_1 \& \overleftarrow{\psi}_2$
$\psi_1 \vee \psi_2$	$\overleftarrow{\psi}_1 \mid \overleftarrow{\psi}_2$

For example, if ϕ is of the form $\neg\psi$, then its corresponding bit-vector is obtained by applying bit-wise negation to the bit-vector, $\overleftarrow{\psi}$, corresponding to subformula ψ. Similarly for the other cases.

The next table shows the transformations in the case of both future (X, U) and past (Y, S) temporal operators.

ϕ	bit-vector encoding
$\mathrm{X}\psi$	$\overleftarrow{\psi}^{[l+1]}\ ::\ \overleftarrow{\psi}^{[k+1:1]}$
$\psi_1 \mathrm{U} \psi_2$	$\overleftarrow{\psi}_1 \mathrm{U_{nl}}((\overleftarrow{\psi}_1 \mathrm{U_{nl}} \overleftarrow{\psi}_2)^{[l]}\ ::\ \overleftarrow{\psi}_2^{[k:0]})$
$\mathrm{Y}\psi$	$\ll \overleftarrow{\psi}$
$\psi_1 \mathrm{S} \psi_2$	$\overleftarrow{\psi}_2 \mid (\overleftarrow{\psi}_1\ \&\ !((\overleftarrow{\psi}_1 \mid \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2))$

We illustrate the cases for the past operators first, which are a bit simpler, and then we focus on the future ones.

Yesterday. Given the semantics of formula $\mathrm{Y}\psi$, where $\mathrm{Y}\psi$ holds at i iff ψ holds at $i-1$, the bit-vector for $\mathrm{Y}\psi$ is the one for ψ, but shifted "to the left" (from $i-1$ to i, recall that position 0 in bit-vectors is the rightmost one). Note that, by definition of shifting to the left, the rightmost bit of $\ll \overleftarrow{\psi}$ is 0, which is consistent with the semantics of $\mathrm{Y}\psi$ in the origin.

Table I shows an example of calculation of the bit-vector for $\psi_1 \mathrm{S} \psi_2$, given bit-vectors $\overleftarrow{\psi}_1$ and $\overleftarrow{\psi}_2$.

TABLE I
AN EXAMPLE OF CALCULATION OF BIT-VECTOR FOR S.

	bit-vector	11	10	9	8	7	6	5	4	3	2	1	0
	$\overleftarrow{\psi}_1$	**1**	**1**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**
	$\overleftarrow{\psi}_2$	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**0**
	$\overleftarrow{\psi}_1 \mid \overleftarrow{\psi}_2$	1	1	0	1	1	1	1	0	1	1	1	1
$\overleftarrow{bv}_1$	$(\overleftarrow{\psi}_1 \mid \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2$	1	1	1	0	0	0	1	1	1	0	0	1
$\overleftarrow{bv}_2$	$!\overleftarrow{bv}_1$	0	0	0	1	1	1	0	0	0	1	1	0
$\overleftarrow{bv}_3$	$\overleftarrow{bv}_2 \& \overleftarrow{\psi}_1$	0	0	0	1	1	1	0	0	0	1	0	0
$\overleftarrow{\psi_1 \mathrm{S} \psi_2}$	$\overleftarrow{bv}_3 \mid \overleftarrow{\psi}_2$	**0**	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**0**

Since. First of all, recall that, informally, $\psi_1 \mathrm{S} \psi_2$ holds at i iff either ψ_2 holds at i, or ψ_1 holds at $i, i-1, \ldots$ until an instant $i' < i$ in which ψ_2 holds (ψ_1 can hold in i' or not). Said in another way, if ψ_2 holds in i', then $\psi_1 \mathrm{S} \psi_2$ holds there, and in all instants $i'+1, i'+2, \ldots$ in which ψ_1 holds consecutively. We use addition to capture this mechanism through which the truth of S "carries" to the left from when ψ_2 holds, as long as

ψ_1 holds. Consider term $(\overleftarrow{\psi}_1|\overleftarrow{\psi}_2)+\overleftarrow{\psi}_2$; whenever $\overleftarrow{\psi}_2^{[i']}$ is 1, this generates a carry (since both bits are 1), which propagates as long as $\overleftarrow{\psi}_1^{[i]}$ is 1, as between bits 1-2 and 6-8 in Table I. The net effect is that in the sum the bits from i' until ψ_1 stops holding are set to 0, and the others are set to 1 (we do not delve into the details of some special cases, which are covered by the correctness proof of Section III-C). Starting from this basis, the rest of the operations are necessary to set to 1 exactly all the bits in which S holds (notice that, in this case, $\psi_1 \mathrm{S} \psi_2$ holds not only where $(\overleftarrow{\psi}_1|\overleftarrow{\psi}_2)+\overleftarrow{\psi}_2$ is 0, but also at position 3, where ψ_2 holds). More precisely, the result of $(\overleftarrow{\psi}_1|\overleftarrow{\psi}_2)+\overleftarrow{\psi}_2$ is bit-wise complemented to bring the 0s to 1s; it is then filtered against $\overleftarrow{\psi}_1$ with a bitwise "and" to eliminate those cases in which the result of the sum is 0 because both $\overleftarrow{\psi}_1^{[i]}$ and $\overleftarrow{\psi}_2^{[i]}$ are 0 and there is no carry from $i-1$ ($\psi_1 \mathrm{S} \psi_2$ does not hold there); finally, the bit-wise "or" with $\overleftarrow{\psi}_2$ sets to 1 all those positions in which ψ_2 holds, since $\psi_1 \mathrm{S} \psi_2$ holds there.

We now illustrate the encoding of the future operators X, U.

Next. The encoding of formula $\mathrm{X}\psi$ is essentially dual to that of $\mathrm{Y}\psi$, i.e., a bit-wise shift to the right of bit-vector $\overleftarrow{\psi}$. In fact, $\mathrm{X}\psi$ holds at i iff ψ holds at $i+1$. However, a true right-shift would always introduce a 0 at position $k+1$, which would be incorrect. In fact, recall that, in bounded encodings such as the classic one presented in Section II-B, we need the state to repeat at positions $k+1$ and l, with the latter corresponding to the position of the first s in $\alpha s \beta s$, which in the bit-vector encoding is represented by the value of bit-vector $\overleftarrow{loop}$. Hence, the value of $\mathrm{X}\psi$ at positions $k+1$ and l must be the same (this is also true for $\mathrm{Y}\psi$, but it is achieved in a different way than for $\mathrm{X}\psi$, as it will be explained later). Note that the value of $\mathrm{X}\psi$ at position l is the same as ψ at position $l+1$, hence, the bit-vector corresponding to $\mathrm{X}\psi$ is obtained by concatenating $\overleftarrow{\psi}^{[l+1]}$ with $\overleftarrow{\psi}^{[k+1:1]}$.

Until. The bit-vector corresponding to $\psi_1 \mathrm{U} \psi_2$ is computed by exploiting the operator $\mathrm{U_{nl}}$ introduced above. In fact, the operations performed by $\mathrm{U_{nl}}$ are the same as those for the computation of $\psi_1 \mathrm{S} \psi_2$, but carried out left-to-right instead of right-to-left. To achieve this, bit-vectors $\overleftarrow{x}|\overleftarrow{y}$ and $\overleftarrow{y}$ are reversed through Rev before being added together, and the result is also reversed.

As mentioned above, the operations performed by operator $\mathrm{U_{nl}}$ are the same as those for the encoding of $\psi_1 \mathrm{S} \psi_2$, but carried out in the reverse direction. Hence, they produce, from bit-vectors $\overleftarrow{x}$ and $\overleftarrow{y}$, a bit-vector that corresponds to the truth of $x \mathrm{U} y$ evaluated over finite models, whereby $x \mathrm{U} y$ holds at position $k+1$ iff y holds at $k+1$. However, when finite traces of the form $\alpha s \beta s$ are used to represent ultimately periodic models, one must take into account that, in the encoding, $\psi_1 \mathrm{U} \psi_2$ can hold in $k+1$ (i.e., state s) also if ψ_1 holds there, and ψ_2 holds somewhere in β (with ψ_1 holding in the prefix of β until then). Since the states in positions $k+1$ and l are the same (they are both s), this is the same as saying that, in position l, $\psi_1 \mathrm{U} \psi_2$ holds in the finite model, that is, the l-th bit of $\overleftarrow{\psi}_1 \mathrm{U_{nl}} \overleftarrow{\psi}_2$ is 1. If, conversely, the l-th bit of $\overleftarrow{\psi}_1 \mathrm{U_{nl}} \overleftarrow{\psi}_2$ is 0, then there is no point in β in which ψ_2 holds, with ψ_1 holding until then, so $\psi_1 \mathrm{U} \psi_2$ does not hold in state s (at position l or at position $k+1$). Finally, to correctly compute the bit-vector corresponding to $\psi_1 \mathrm{U} \psi_2$, we compute $\overleftarrow{\psi}_1 \mathrm{U_{nl}} \overleftarrow{\psi}_2$, take its l-th bit to determine whether $\psi_1 \mathrm{U} \psi_2$ holds in s, then use bit-vector $(\overleftarrow{\psi}_1 \mathrm{U_{nl}} \overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}$ as the second argument of $\mathrm{U_{nl}}$; in fact, as mentioned above, by definition the value of $\overleftarrow{x} \mathrm{U_{nl}} \overleftarrow{y}$ at a position $k+1$ is 1 iff $\overleftarrow{y}^{[k+1]} = 1$.

Whereas the functions computing bit-vectors for future operators X, U by construction impose that the value of the subformula in positions l and $k+1$ is the same, the same does not happen for the functions that compute the bit-vectors for past operators Y, S. These so-called "last state constraints" (see also Section II-B) are easily included in the encoding by adding, for each formula $\mathrm{Y}\psi$, the constraint $(\ll \overleftarrow{\psi})^{[l]} = (\ll \overleftarrow{\psi})^{[k+1]}$, and similarly, *mutatis mutandis*, for each formula $\psi_1 \mathrm{S} \psi_2$. The "last state constraints" must be added for all subformulae, including propositional letters, so for each $p \in AP$ we also include the constraint $\overleftarrow{p}^{[l]} = \overleftarrow{p}^{[k+1]}$. Notice that it is not necessary to include the "last state constraints" for each subformula of the form $\neg\psi$, $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$, as they are automatically guaranteed recursively. We indicate this set of constraints as $|BVLastStateConstraints|_k$.

Finally, the so-called "loop constraints" (see Section II-B) are easily imposed by adding, for each $p \in AP$, the constraint $\overleftarrow{p}^{[l-1]} = \overleftarrow{p}^{[k]}$. We name this set of constraints $|BVLoopConstraints|_k$.

Then, given an LTL formula ϕ, the complete bit-vector-based encoding, called ϕ_{bv}, is given by:

- the set $|BVLastStateConstraints|_k$, which includes constraints for each $p \in AP$, and for each past formula $\mathrm{Y}\psi$ and $\psi_1 \mathrm{S} \psi_2$;
- the set $|BVLoopConstraints|_k$, which includes a constraint for each $p \in AP$
- constraint $\overleftarrow{\phi}^{[0]} = 1$, where $\overleftarrow{\phi}$ is the bit-vector obtained through the transformations above.

For example, consider formula $\neg \mathrm{X} p \vee (q \mathrm{U} \mathrm{Y} p)$. Its complete encoding is given by the following formula:

$$\begin{gathered}
\overleftarrow{p}^{[l]} = \overleftarrow{p}^{[k+1]} \wedge \overleftarrow{q}^{[l]} = \overleftarrow{q}^{[k+1]} \wedge (\ll \overleftarrow{p})^{[l]} = (\ll \overleftarrow{p})^{[k+1]} \wedge \\
\overleftarrow{p}^{[l-1]} = \overleftarrow{p}^{[k]} \wedge \overleftarrow{q}^{[l-1]} = \overleftarrow{q}^{[k]} \wedge \\
\left(\begin{array}{l} !(\overleftarrow{p}^{[l+1]} \;::\; \overleftarrow{p}^{[k+1:1]}) \;| \\ \overleftarrow{q}\, \mathrm{U_{nl}}((\overleftarrow{q}\, \mathrm{U_{nl}}(\ll \overleftarrow{p}))^{[l]} \;::\; (\ll \overleftarrow{p})^{[k:0]}) \end{array} \right)^{[0]} = 1
\end{gathered} \tag{1}$$

As remarked in Section II-A, it is customary to define the other temperal operators from the basic ones as abbreviations. In some cases we can use the abbreviation to further simplify the encoding of these operators by exploiting the properties of bit-vector operations. First of all, we can introduce operators $\overleftarrow{x} \mathrm{R_{nl}} \overleftarrow{y}$ and $\mathrm{F_{nl}} \overleftarrow{y}$, shown below, whose definitions correspond, respectively, to the simplified versions of $\neg(\neg \overleftarrow{x} \mathrm{U_{nl}} \neg \overleftarrow{y})$ and $\top \mathrm{U_{nl}} \overleftarrow{y}$.

Operator	Definition
$\overleftarrow{x}\,\mathrm{R_{nl}}\,\overleftarrow{y}$	$\overleftarrow{y}\ \&\ (\overleftarrow{x} \mid \mathrm{Rev}(\mathrm{Rev}(!\overleftarrow{x} \mid !\overleftarrow{y}) + \mathrm{Rev}\ !\overleftarrow{y}))$
$\mathrm{F_{nl}}\,\overleftarrow{y}$	$\overleftarrow{y} \mid !\mathrm{Rev}(\mathrm{Rev}\,\overleftarrow{y} - \overleftarrow{1})$

Using these operators, we can in turn simplify the encoding of derived temporal operators R and F as shown below. Similarly for the encoding of operators T and P.

ϕ	bit-vector encoding
$\psi_1 \mathrm{R} \psi_2$	$\overleftarrow{\psi}_1 \mathrm{R_{nl}}((\overleftarrow{\psi}_1 \mathrm{R_{nl}} \overleftarrow{\psi}_2)^{[l]} \ :: \ \overleftarrow{\psi}_2^{[k:0]})$
$\mathrm{F}\psi$	$\mathrm{F_{nl}}((\mathrm{F_{nl}}\overleftarrow{\psi})^{[l]} \ :: \ \overleftarrow{\psi}^{[k:0]})$
$\psi_1 \mathrm{T} \psi_2$	$\overleftarrow{\psi}_2\ \&\ (\overleftarrow{\psi}_1 \mid ((!\overleftarrow{\psi}_1 \mid !\overleftarrow{\psi}_2) + !\overleftarrow{\psi}_2))$
$\mathrm{P}\psi$	$\overleftarrow{\psi} \mid !(\overleftarrow{\psi} - \overleftarrow{1})$
$\mathrm{W}\psi$	$::^{k+2} \Downarrow \overleftarrow{\psi}$

Finally, the encoding of "always ψ" (written W), which is defined as $\mathrm{W}\psi = \mathrm{G}\psi \wedge \mathrm{H}\psi$ can be simplified as shown above by considering that its value is 1 throughout $[0, k+1]$ if $\overleftarrow{\psi}$ has value 1 everywhere, otherwise its value is 0 at every position.

We conclude this section by remarking that if one wants to consider formulae in PNF (see Section II-B), which entails that operator Z be introduced, the semantics of $\mathrm{Z}\psi$ is simply captured by the transformation $\ll \overleftarrow{\psi} \mid \overleftarrow{1}$.

C. Correctness of the encoding

In this section we show that the new, bit-vector-based bounded encoding of LTL formulae is equivalent to the classic Boolean one introduced in Section II-B.

To show the equivalence, it is natural to consider a bit-vector $\overleftarrow{x}_{[n]}$ of size n, whose bits are $\overleftarrow{x}_{[n]}^{[0]}, \ldots, \overleftarrow{x}_{[n]}^{[n-1]}$, as a set of n Boolean variables $|[x]|_0, \ldots, |[x]|_{n-1}$. An operation on bit-vectors (e.g., $\ll$) returns a bit-vector, which corresponds to its own set of Boolean propositions; hence, for example, bit-vector $\ll \overleftarrow{x}$, obtained by shifting $\overleftarrow{x}$ to the left, comprises n bits, $(\ll \overleftarrow{x})^{[0]}, \ldots, (\ll \overleftarrow{x})^{[n-1]}$, which in turn correspond to Boolean variables $|[\ll x]|_0, \ldots, |[\ll x]|_{n-1}$.

Recall that, given an LTL formula ϕ, we indicate by ϕ_{B} the set of formulae that correspond to the Boolean encoding of ϕ, and by ϕ_{bv} the ones of the bit-vector-based encoding of ϕ (see Sections II-B and III-B for the definitions of ϕ_{B} and ϕ_{bv}). We have the following result.

Theorem 1: Given an LTL formula ϕ, the encoding ϕ_{B} is equivalent to the encoding ϕ_{bv}.

Proof: We prove the equivalence by showing that every constraint in ϕ_{B} corresponds to a constraint in ϕ_{bv} and vice-versa. First of all, we remark that the bit-vector encoding assumes the existence of a loop starting at position l, so we focus on this case for the proof. This is without loss of generality, as it is always possible to extend an aperiodic trace with a "dummy" loop at the end, in which nothing happens. Notice also that, since l is the position of the first s in $\alpha s \beta s$, we have that $0 \leq l < k+1$.

Let us first consider the $|LoopConstraints|_k$. It is easy to see that they directly correspond to $|BVLoopConstraints|_k$, which in fact impose that, for each $p \in AP$, in the corresponding bit-vector $\overleftarrow{p}$ it holds that $\overleftarrow{p}^{[l-1]} = \overleftarrow{p}^{[k]}$. Given the correspondence between bits of bit-vectors and propositional letters introduced in the Boolean encoding, this is the same as saying, for each $p \in AP$, that $|[p]|_{l-1} \Leftrightarrow |[p]|_k$.

We will tackle $|LastStateConstraints|_k$ at the end of the proof. We now focus on the encoding of propositional connectives, i.e. $|PropConstraints|_k$. The bit-vector of formula $\psi_1 \wedge \psi_2$ is simply $\overleftarrow{\psi}_1 \& \overleftarrow{\psi}_2$, that is, for each bit $i \in [0, k]$ (we will deal with the $k+1$-th bit in the $|LastStateConstraints|_k$), it is $(\overleftarrow{\psi}_1 \& \overleftarrow{\psi}_2)^{[i]} = 1$ iff $\overleftarrow{\psi}_1^{[i]} = 1 \wedge \overleftarrow{\psi}_2^{[i]} = 1$, which corresponds to Boolean constraints $|[\psi_1 \wedge \psi_2]|_i \Leftrightarrow |[\psi_1]|_i \wedge |[\psi_2]|_i$, i.e., $|PropConstraints|_k$ for the $\wedge$ connective. Similarly for the $\vee$ connective. As far as the $\neg$ connective is concerned, recall that the Boolean encoding assumes formulae in Positive Normal Form (PNF), an optimization for saving intermediate Boolean variables which we do not pursue in the bit-vector-based encoding. Even in the Boolean encoding, the optimization could be eliminated without impacting on the correctness of the encoding, by introducing a Boolean variable $|[\neg\psi]|$ for each formula of the form $\neg\psi$, with the constraint that $|[\neg\psi]|_i \Leftrightarrow \neg|[\psi]|_i$. Then, it is obvious that such a constraint corresponds to the semantics of the bit-wise negation: $(!\overleftarrow{\psi})^{[i]} = 1$ iff $\overleftarrow{\psi}^{[i]} = 0$.

Let us now focus on $|TempConstraints|_k$, starting from those regarding past operators Y and S. The semantics of $\mathrm{Y}\psi$ is captured by the transformation $\ll \overleftarrow{\psi}$, which by definition means that, for all $i \in [1, k+1]$, it holds that $(\ll \overleftarrow{\psi})^{[i]} = \overleftarrow{\psi}^{[i-1]}$. This corresponds to the constraint that, for $1 \leq i \leq k+1$, $|[\mathrm{Y}\psi]|_i \Leftrightarrow |[\psi]|_{i-1}$. In addition, by definition of $\ll$, $(\ll \overleftarrow{\psi})^{[0]} = 0$, which corresponds to the constraint $\neg|[\mathrm{Y}\psi]|_0$ that appears in the Boolean encoding. The encoding of $\psi_1 \mathrm{S} \psi_2$ is trickier. In the Boolean encoding, we have that, for $i \in [1, k+1]$, $|[\psi_1 \mathrm{S} \psi_2]|_i \Leftrightarrow |[\psi_2]|_i \vee (|[\psi_1]|_i \wedge |[\psi_1 \mathrm{S} \psi_2]|_{i-1})$, with the additional constraint that $|[\psi_1 \mathrm{S} \psi_2]|_0 \Leftrightarrow |[\psi_2]|_0$. To show how this is equivalent to the bit-vector encoding, in which the bit-vector representing $\psi_1 \mathrm{S} \psi_2$ is $\overleftarrow{\psi}_2 | (\overleftarrow{\psi}_1 \& !((\overleftarrow{\psi}_1 | \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2))$, we need to show that, for $i \in [1, k+1]$, if $\overleftarrow{\psi}_2^{[i]} = 0$ and $\overleftarrow{\psi}_1^{[i]} = 1$, then $!((\overleftarrow{\psi}_1 | \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2)^{[i]}$ is 1 iff $\psi_1 \mathrm{S} \psi_2$ holds in $i-1$ (if $\overleftarrow{\psi}_2^{[i]} = 1$, i.e., $|[\psi_2]|_i$ is true, or $\overleftarrow{\psi}_1^{[i]} = 0$, i.e., $|[\psi_1]|_i$ is false, the Boolean and bit-vector encodings clearly yield the same result). Then, we need to determine when $((\overleftarrow{\psi}_1 | \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2)^{[i]} = 0$, provided $\overleftarrow{\psi}_2^{[i]} = 0$ and $\overleftarrow{\psi}_1^{[i]} = 1$. Notice that, if $\overleftarrow{\psi}_2^{[i]} = 0$ and $\overleftarrow{\psi}_1^{[i]} = 1$, then $((\overleftarrow{\psi}_1 | \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2)^{[i]} = 0$ iff there is a carry from $i-1$. This occurs, recursively, iff either $\overleftarrow{\psi}_2^{[i-1]} = 0$, $\overleftarrow{\psi}_1^{[i-1]} = 1$ and there is a carry from $i-2$, or $\overleftarrow{\psi}_2^{[i-1]} = 1$. By inductive reasoning, for $((\overleftarrow{\psi}_1 | \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2)^{[i]}$ to be 0, there must be $i' < i$ in which $\overleftarrow{\psi}_2^{[i']} = 1$, and for all $i < i'' < i'$ it is $\overleftarrow{\psi}_2^{[i'']} = 0$ and $\overleftarrow{\psi}_1^{[i'']} = 1$. In other words, $\psi_1 \mathrm{S} \psi_2$ holds in $i-1$. If, on the other hand, $\psi_1 \mathrm{S} \psi_2$ holds in i, there must be $i' < i$ where $\overleftarrow{\psi}_2^{[i']} = 1$ and for all $i < i'' < i'$ it is $\overleftarrow{\psi}_1^{[i'']} = 1$. In this case in i' $((\overleftarrow{\psi}_1 | \overleftarrow{\psi}_2) + \overleftarrow{\psi}_2)^{[i']}$ generates a carry, which propagates to the left, until in i it produces a 0. Notice that, since in position 0 there can be no carry,

$(\overleftarrow{\psi}_2|(\overleftarrow{\psi}_1\&!((\overleftarrow{\psi}_1|\overleftarrow{\psi}_2)+\overleftarrow{\psi}_2)))^{[0]} = 1$ iff $\overleftarrow{\psi}_2^{[0]} = 1$, which corresponds to the Boolean encoding of $\psi_1\mathrm{S}\psi_2$ in 0.

Let us now focus on the future operators X and U. Since the bit-vector for $\mathrm{X}\psi$ is $\overleftarrow{\psi}^{[l+1]} :: \overleftarrow{\psi}^{[k+1:1]}$, for $i \in [0,k]$ (we will deal with the case $i = k+1$ when focusing on $|LastStateConstraints|_k$), we have that $(\overleftarrow{\psi}^{[l+1]} :: \overleftarrow{\psi}^{[k+1:1]})^{[i]} = \overleftarrow{\psi}^{[i+1]}$, which corresponds to the Boolean encoding of $\mathrm{X}\psi$, i.e., $|[\mathrm{X}\psi]|_i \Leftrightarrow |[\psi]|_{i+1}$. The encoding of U relies on operator $\mathrm{U_{nl}}$, whose definition coincides with that of S, except that the arguments are reversed before applying the sum, and reversed again after having applied it. As a consequence, the properties of $\mathrm{U_{nl}}$ are the same as those of the encoding of S, except that the bit vectors are considered in reverse order. Then, we can conclude that, for $i \in [0,k]$, $(\overleftarrow{x}\,\mathrm{U_{nl}}\,\overleftarrow{y})^{[i]} = 1$ iff there is $i \le i' \le k+1$ such that $\overleftarrow{y}^{[i']} = 1$ and, for all $i \le i'' < i'$, $\overleftarrow{x}^{[i'']} = 1$ holds. In addition, $(\overleftarrow{x}\,\mathrm{U_{nl}}\,\overleftarrow{y})^{[k+1]} = 1$ iff $\overleftarrow{y}^{[k+1]} = 1$. The Boolean encoding of U differs from that of S because, unlike the latter, where the value of $\psi_1\mathrm{S}\psi_2$ in 0 depends exclusively on the value of ψ_2 there, in the former the value in k depends on whether there is a position in the loop (i.e., in the part $s\beta$ of the trace) where ψ_2 holds, as defined by constraints $|Eventualities|_k$. Given the properties of operator $\mathrm{U_{nl}}$, and in particular the fact that $(\overleftarrow{x}\,\mathrm{U_{nl}}\,\overleftarrow{y})^{[k+1]} = 1$ iff $\overleftarrow{y}^{[k+1]} = 1$, the value of $(\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[k]}$ is 1 iff either $\overleftarrow{\psi}_2^{[k]} = 1$, or $\overleftarrow{\psi}_1^{[k]} = 1$ and $(\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} = 1$. In both cases, there must be a position in the loop in which ψ_2 holds; the first is evident (that position is k); the second derives from the fact that by the properties of $\mathrm{U_{nl}}$, $\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2$ is 1 in l iff there is $l \le i' \le k$ in which ψ_2 holds (notice that, as it will be shown below, it is guaranteed that $\overleftarrow{\psi}_2^{[l]} = \overleftarrow{\psi}_2^{[k+1]}$).

Finally, we need to show that the $|LastStateConstraints|_k$ are also captured by the bit-vector-based encoding. To this end, recall that for propositional letters and past operators Y and S the following constraints are explicitly introduced: $\overleftarrow{p}^{[l]} = \overleftarrow{p}^{[k+1]}$, $(\ll \overleftarrow{\psi})^{[l]} = (\ll \overleftarrow{\psi})^{[k+1]}$ and $(\overleftarrow{\psi}_2|(\overleftarrow{\psi}_1\&!((\overleftarrow{\psi}_1|\overleftarrow{\psi}_2)+\overleftarrow{\psi}_2)))^{[l]} = (\overleftarrow{\psi}_2|(\overleftarrow{\psi}_1\&!((\overleftarrow{\psi}_1|\overleftarrow{\psi}_2)+\overleftarrow{\psi}_2)))^{[k+1]}$. These are trivially equivalent to Boolean constraints $|[p]|_l \Leftrightarrow |[p]|_{k+1}$, $|[\mathrm{Y}\psi]|_l \Leftrightarrow |[\mathrm{Y}\psi]|_{k+1}$ and $|[\psi_1\mathrm{S}\psi_2]|_l \Leftrightarrow |[\psi_1\mathrm{S}\psi_2]|_{k+1}$. In the case of X and U operators the constraint is enforced in the construction of the corresponding bit-vector. More precisely, we have $(\overleftarrow{\psi}^{[l+1]} :: \overleftarrow{\psi}^{[k+1:1]})^{[l]} = \overleftarrow{\psi}^{[l+1]} = (\overleftarrow{\psi}^{[l+1]} :: \overleftarrow{\psi}^{[k+1:1]})^{[k+1]}$ (recall that $0 \le l < k+1$). As far as the U operator is concerned, $(\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[k+1]} = 1$ iff the $k+1$-th bit of the second argument is 1, i.e., $(\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[k+1]} = (\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]}$. If $(\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} = 1$, then there is $l \le i' \le k+1$ such that $\overleftarrow{\psi}_2^{[i']} = 1$ and $\overleftarrow{\psi}_1^{[i'']} = 1$ for all $l \le i'' < i'$. If $i' < k+1$ it is easy to see that $(\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[l]} = (\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} = (\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[k+1]}$. If, instead, $i' = k+1$, then $(\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} = \overleftarrow{\psi}_2^{[k+1]}$, so $((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}) = \overleftarrow{\psi}_2$, and, again $(\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[l]} = (\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} = (\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]}))^{[k+1]}$. If $(\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} = 0$, then there is no $l \le i' \le k+1$ such that $\overleftarrow{\psi}_2^{[i']} = 1$ and $\overleftarrow{\psi}_1^{[i'']} = 1$ for all $l \le i'' < i'$. Then, *a fortiori* also $(\overleftarrow{\psi}_1\mathrm{U_{nl}}(0 :: \overleftarrow{\psi}_2^{[k:0]}))^{[l]} = 0$, so again we have the equality of the l-th and $k+1$-th bits in $\overleftarrow{\psi}_1\mathrm{U_{nl}}((\overleftarrow{\psi}_1\mathrm{U_{nl}}\overleftarrow{\psi}_2)^{[l]} :: \overleftarrow{\psi}_2^{[k:0]})$. Finally, the fact that $|LastStateConstraints|_k$ hold also for formulae of the form $\neg\psi$, $\psi_1 \wedge \psi_2$ and $\psi_1 \vee \psi_2$ can be easily shown by induction, where the base cases are those already tackled above.

To conclude the proof, we remark that the classic Boolean encoding natively defines the semantics of operators R and T. In our bit-vector-based encoding, instead, the encoding of these operators exploits their definition as abbreviations for formulae involving U and S; hence, the correctness of the encoding in this case derives directly from that of the encoding of formulae $\neg\psi$, $\psi_1\mathrm{U}\psi_2$ and $\psi_1\mathrm{S}\psi_2$. ∎

D. Complexity

As remarked in Section III-B, in the bit-vector-based encoding of an LTL formula ϕ we introduce a number of bit-vectors of length $k+2$ that is equal to the number of propositional letters appearing in ϕ. In the Boolean encoding, instead, $k+2$ propositional letters are introduced for each *subformula* of ϕ, hence the bit-vector encoding is more compact from this point of view. The size of the constraint $\overleftarrow{\phi}^{[0]} = 1$, which relies on the transformations captured in Section III-B to build the expression of $\overleftarrow{\phi}$, is linear in the size of the formula if we introduce one bit-vector for each subformula. In fact, in this case if n is the number of subformulae of ϕ, for each subformula we build a transformation of bit-vectors, according to the tables of Section III-B, whose size is constant, hence the total size of constraint $\overleftarrow{\phi}^{[0]} = 1$ is $O(n)$. Since the number of propositional letters appearing in ϕ is bounded by the size of the latter, and since the size of the loop constraints (of the form $\overleftarrow{p}^{[l-1]} = \overleftarrow{p}^{[k]}$) is constant with respect to n, the size of $|BVLoopConstraints|_k$ is also $O(n)$.

Finally, one element of $|BVLastStateConstraints|_k$ is introduced for each subformula whose principal operator is a past one. For example, if subformula $\mathrm{YY}p$ appears in ϕ, constraints $(\ll \overleftarrow{p})^{[l]} = (\ll \overleftarrow{p})^{[k+1]}$ and $(\ll\ll \overleftarrow{p})^{[l]} = (\ll\ll \overleftarrow{p})^{[k+1]}$ both belong to $|BVLastStateConstraints|_k$. If $m \le n$ is the number of past operators appearing in ϕ, the size of $|BVLastStateConstraints|_k$ is $O(mn)$, hence it is quadratic in the worst case. Regarding the size of $|BVLastStateConstraints|_k$ however, we remark that it can be easily made $O(n)$ as in the Boolean encoding by introducing, for each subformula of ϕ (or at least for each past subformula of ϕ) a fresh bit-vector; in this case, for example, the $|BVLastStateConstraints|_k$ for formula $\mathrm{YY}p$ would be, after having defined $\overleftarrow{bv}_1 = \ll \overleftarrow{p}$ and $\overleftarrow{bv}_2 = \ll \overleftarrow{bv}_1$, $\overleftarrow{bv}_1^{[l]} = \overleftarrow{bv}_1^{[k+1]}$ and $\overleftarrow{bv}_2^{[l]} = \overleftarrow{bv}_2^{[k+1]}$. However, if the number of past subformulae is small with respect to n, then it might be more efficient to avoid introducing additional, intermediate bit-vectors, and exploit the effective simplification algorithms implemented in solvers of Bit-Vector Logic to handle the constraints effectively.

IV. Implementation and Evaluation

The bit-vector-based encoding has been implemented as a plugin in the $\mathbb{Z}$ot [12] tool, called ***bvzot***.

$\mathbb{Z}$ot is an extensible Bounded Model/Satisfiability Checker written in Common Lisp. More precisely, $\mathbb{Z}$ot is capable of performing bounded satisfiability checking of formulae written both in LTL (with past operators) and in the propositional, discrete-time fragment of the metric temporal logic TRIO [17], which is equivalent to LTL, but more concise. In fact, TRIO formulae can straightforwardly be translated into LTL formulae, so we use the two temporal logics interchangeably.

The verification process in $\mathbb{Z}$ot goes through the following steps: (i) the user writes the specification to be checked as a set of temporal logic formulae (these formulae could also be produced automatically as in [7]), and selects the plugin and the time bound (i.e., the value of bound k) to be used to perform the verification; (ii) depending on the input temporal logic (TRIO or LTL) and the selected plugin, $\mathbb{Z}$ot encodes the received specification in a target logic (e.g., Propositional Logic, or Bit-Vector Logic); (iii) $\mathbb{Z}$ot feeds the encoded specification to a solver that is capable of handling the target logic; (iv) the result obtained by the solver is parsed back and presented to the user in a textual representation.

$\mathbb{Z}$ot supports both SAT solvers (e.g., MiniSat [18]) for Propositional Logic, and SMT solvers (e.g., Z3 [11]) for Bit-Vector Logic and decidable fragments of first-order logic.

To evaluate the bit-vector-based encoding we compared it against three other encodings available in the literature: the classic bounded encoding presented in [16]; the optimized encoding presented in [19], which has been further improved in [20] and made incremental in [21]; and the encoding optimized for metric temporal logic presented in [9]. The first two encodings are implemented in the well-known NuSMV model checker [13] (in fact, NuSMV implements an optimized, incremental version of the classic encoding of [16]), whereas the third is implemented in the *meezot* plugin of the $\mathbb{Z}$ot tool.

In the rest of this section we will label the experiments carried out with the classic encoding implemented in NuSMV as *bmc*, those performed with the optimized encoding of [20] as *smbc*, those with the incremental version of *sbmc* presented in [21] as *sbmc_inc*[1], and those performed with the metric encoding implemented in $\mathbb{Z}$ot as *meezot* (all these labels come from the commands used in NuSMV and $\mathbb{Z}$ot to select the encodings). Note that both NuSMV and $\mathbb{Z}$ot support other encodings for LTL/TRIO; we have chosen those mentioned above because further experiments, not reported here, have shown them to be, on average, the most efficient ones for the two tools.

To test the relative efficiency of the four encodings, we applied them to the verification of three case studies, two from the literature and one from previous work of ours. The case studies were chosen mostly for their complexity to highlight the relative strengths and weaknesses of each tool. These three case studies employ a BSC approach, that is, they use temporal logic to describe both the system under verification and the properties to be checked. In all three cases we performed two kinds of checks. First, we took the temporal logic formula ϕ_S describing the system, and we simply checked for its satisfiability. This allowed us to determine whether the specification is realizable or not. As a second type of check, we also provided a logic formula ϕ_P capturing the property that the system should have satisfied, and we fed the BSC algorithm with formula $\phi_S \wedge \neg\phi_P$ to determine whether the property holds for the system or not. We also experimented with different bounds k to analyze how the tools behave when k is increased.

We now briefly introduce the three case studies; interested readers can refer to the cited literature for their details.

Kernel Railway Crossing (KRC). The KRC problem is frequently used for comparing real-time notations and tools [22]. A railway crossing system prevents crossing of the railway by vehicles during passage of a train, by controlling a gate. A temporal logic-based version of the KRC was developed in [9] for benchmarking purposes. It describes one track and one direction of movement of the trains, and it considers an interlocking system. We experimented with two sets of time constants that allow different degrees of nondeterminism, hereafter denoted as `krc1` and `krc2`. The level of nondeterminism is increased by using bigger time constants, e.g., the time for a train to go through the railway crossing, which increase the number of possible combinations of events in the system. We also carried out formal verification with two properties of interest: a safety property that says that as long as a train is in the critical region the gate is closed (`P1`); and a utility property that states that the gate must be open when it is safe to do so (i.e., the gate should not be closed when unnecessary), where the notion of "safe" is captured through suitable time constants (`P2`).

Fischer's Protocol. This is a classic algorithm for granting exclusive access to a resource that is shared among many processes. Fischer's protocol is a typical benchmark for verification tools capable of dealing with real-time constraints. The version we used for our tests, where the specification of the system is described through temporal logic formulae, is taken from [9]; it includes 4 processes, and the delay that a process waits after sending a request, which is the key parameter in Fischer's protocol, is 5 time instants. We performed formal verification of a safety property that states that it is never the case that two processes are simultaneously in their critical sections (`P1`).

Verification of UML Diagrams. *Corretto*[2] is the toolset we developed to perform formal verification of UML models [7]. *Corretto* takes as input a set of UML diagrams and produces their formal representation through formulae of temporal logic. In our tests we used the example diagrams introduced in [7], which describe the behavior of an application that pings two servers, and then sends queries to the server that responds

[1]While conductiong the *sbmc_inc* experiments we did not activate the completeness checking option since it often slows the verification down, as shown in [21].

[2]https://github.com/deib-polimi/Corretto

first. The model comprises a loop, and we performed tests on two versions of the system, called `sdserverl2`, and `sdserverl3`, where the number of iterations in the loops is 2 and 3, respectively. We also performed formal verification on the example system using property `P1` defined in [7], to which we refer the reader for further details on this case study.

To compare the performances of the *bmc*, *sbmc*, *sbmc_inc*, *meezot* and *bvzot* encodings, we built a simple translation tool that converts specifications written in the ℤot input language such as those used in [9] and [7] into the input language of NuSMV.

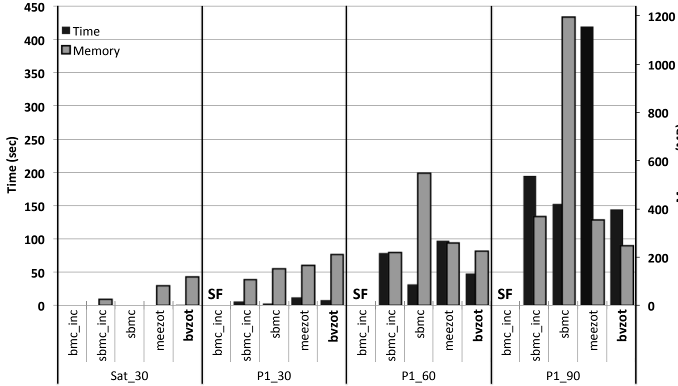

Fig. 1. Time/Memory Comparison for KRC1 (SAT and P1).

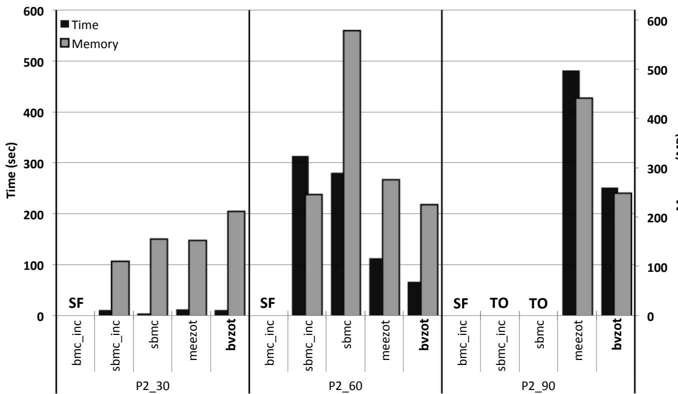

Fig. 2. Time/Memory Comparison for KRC1 (P2).

Figures 1-7 show the time (in seconds) and memory (in MBs) consumed in each of the experiments we performed. Note that if no bar is visible, and no error tag is reported, this means that the number is very small.[3]

For example, Figure 1 shows the time/memory consumption for each encoding (*bmc*, *sbmc*, *sbmc_inc*, *meezot*, and *bvzot*) for the various checks on example `krc1`: simple satisfiability checking with maximum bound $k = 30$ (*sat_30*) and verification of property `P1` with maximum bound $k = 30$ (*P1_30*), $k = 60$ (*P1_60*) and $k = 90$ (*P1_90*), respectively. The role of

[3]Interested readers can refer to http://home.deib.polimi.it/pourhashem.kallehbasti/icse-2015.php for the complete and detailed data about the experiments.

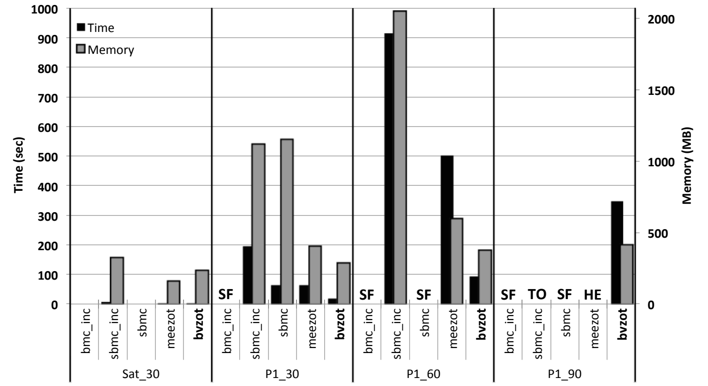

Fig. 3. Time/Memory Comparison for KRC2 (SAT and P1).

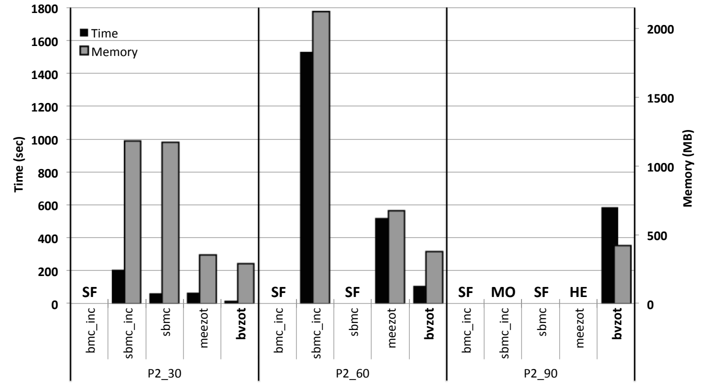

Fig. 4. Time/Memory Comparison for KRC2 (P2).

the "maximum bound" is the following: for a given maximum bound k, the tools iteratively (possibly incrementally) try to find an ultimately periodic model $\alpha\beta^\omega$ where the length of $\alpha\beta$ is $1, 2, \ldots, k$. As soon as a model is found, the search stops, and the model is output; if no model is found for any bound up to k, the search stops at k and the formula is declared unsatisfiable.

All the runs reported in Figures 1-7 had a time limit of 1 hour and a memory limit of 4GB RAM; that is, if the verification took longer than 1 hour or occupied more than 4GB of RAM, it was stopped. Hence, the possible outcomes of a run are **satisfiable**, **unsatisfiable**, **out of time** (**TO**), and **out of memory** (**MO**). In addition, in some cases the tool stopped with a **segmentation fault** (**SF**) error, and in others with **heap exhausted** (**HE**) while pre-processing the specification to produce the encoding.

All the experiments were carried out on a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 8 GB RAM[4]. The NuSMV version was 2.5.4. The SAT and SMT solvers used with ℤot were, respectively, MiniSat version 2.2 and Z3 version 4.3.2.

As the figures show, among the algorithms implemented in NuSMV, *sbmc_inc* is the most memory efficient, while

[4]*bvzot*, along with the code for all the experiments, is available on the ℤot repository [12]

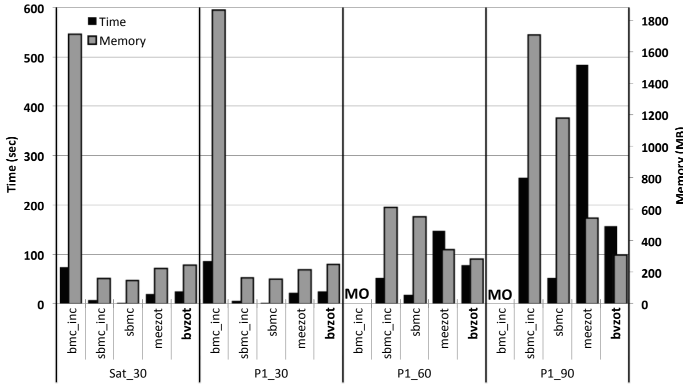

Fig. 5. Time/Memory Comparison for Fischer (SAT and P1).

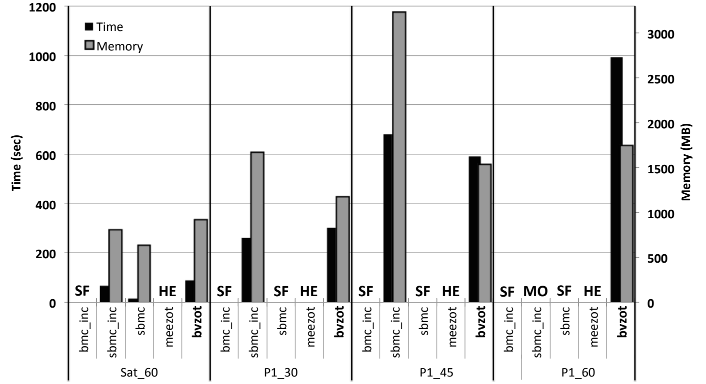

Fig. 7. Time/Memory Comparison for SDServerl3 (SAT and P1).

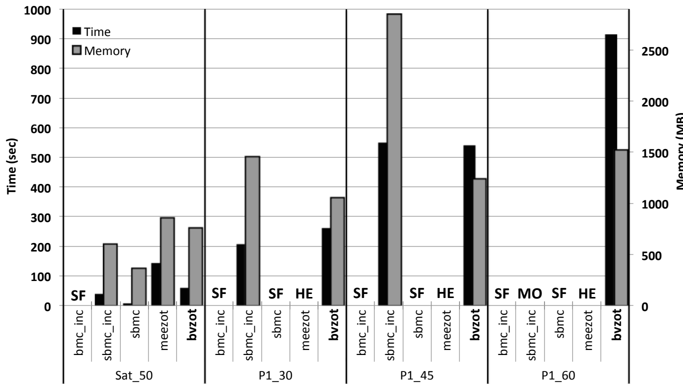

Fig. 6. Time/Memory Comparison for SDServerl2 (SAT and P1).

sbmc is the fastest. There are six models that *sbmc_inc* can afford to verify, while *sbmc* fails. However, for the models both encodings can afford, *sbmc* is faster than *sbmc_inc*.

When performing checks that require small bounds, such as the satisfiability checks, *bvzot* is only occasionally more efficient than the other tools. However, as the models and bounds[5] grow in size, *bvzot* demonstrates its strengths. For example, when proving properties for the KRC version with the highest level of nondeterminism, i.e., `krc2`, *bvzot* is the only tool able to explore all the bounds up to 90, and it is faster than the others when the time bound is kept smaller (30 or 60). Similar results hold for the verification of properties on the UML diagrams, whose formalization in temporal logic is in fact the biggest specification that we have tested due to the necessity of capturing all the possible sequences of events in the Sequence Diagram.

However, we must highlight that in the case of Fischer's protocol, *sbmc* is the most efficient encoding time-wise, whereas *bvzot* is often the one with the least memory consumption.

All in all, we can conclude that the experimental results show a promising ability by *bvzot* to scale up as the size of the specification and of the time bound increase. Further gains could also be obtained by adapting some of the optimizations presented in [20] in *bvzot*.

[5]For each satisfiability experiment, the length of the smallest model found was the same for each tool: 1 for the `krc` examples (the execution in which nothing happens, that is, no train enters the crossing, is admissible), 29 for Fischer's protocol, 10 for `sdserverl2`, and 13 for `sdserverl3`.

V. Related Work

There are essentially two approaches to the problem of satisfiability checking of LTL formulae: bounded and automata-based ones. This paper pursues a bounded approach, and Section IV compares it against similar ones, and in particular those presented in [16], [19], [20], [21] and [9].

Rozier and Vardi [23] carried out a comparison of satisfiability checkers for LTL formulae based on the translation of LTL formulae into Büchi automata. Rozier and Vardi [24] also propose a novel translation of LTL formulae into Transition-based Generalized Büchi Automata, inspired by the translation presented in [25]. Such automata are used by SPOT [26], which is claimed to be the best explicit LTL-to-Büchi automata translator for satisfiability checking purposes based on the experiments carried out in [23]. Li et al. [27] present a novel on-the-fly construction of Büchi automata from LTL formulae that is particularly well suited for finding models of LTL formulae when they exist. Given the different nature of our approach with respect to automata-based ones, however, we did not compare our approach against them, and focused on similar, BSC-based approaches instead.

Finally, an exhaustive evaluation of several techniques and tools (including some that are not based on translation to Büchi automata or on bounded approaches) can be found in [28].

VI. Conclusions and Future Work

This paper presents a novel, efficient technique for verifying LTL specifications implemented in the prototype tool *bvzot*. Obtained results show the ability of this solution to scale up and be usable for analyzing complex LTL specifications efficiently. In classic Boolean encodings, all the constraints are fed to the solver at bit-level, which makes the solver blind to their relations at any higher level. Our encoding introduces the constraints at word-level and exploits the ability of modern solvers to work on word-level simplifications.

As for future work, we plan to keep refining the encoding and identify further challenging case studies and competitors.

References

[1] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS'77)*, 1977, pp. 46–67.

[2] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011.

[3] L. Tan, O. Sokolsky, and I. Lee, "Specification-based testing with linear temporal logic," in *Proceedings of the IEEE International Conference on Information Reuse and Integration*, 2004, pp. 493–498.

[4] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011.

[5] G. Fainekos, H. Kress-Gazit, and G. Pappas, "Temporal logic motion planning for mobile robots," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2005, pp. 2020–2025.

[6] P. Tabuada and G. Pappas, "Linear time logic control of discrete-time linear systems," *IEEE Transactions on Automatic Control*, vol. 51, no. 12, pp. 1862–1877, 2006.

[7] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi, "Flexible Modular Formalization of UML Sequence Diagrams," in *Proc. of the 2nd FME Workshop on Formal Methods in Software Engineering*, ser. FormaliSE 2014, 2014, pp. 10–16.

[8] K. Y. Rozier and M. Y. Vardi, "LTL satisfiability checking," in *Model Checking Software*, ser. Lecture Notes in Computer Science, 2007, vol. 4595, pp. 149–167.

[9] M. Pradella, A. Morzenti, and P. San Pietro, "Bounded Satisfiability Checking of Metric Temporal Logic Specifications," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 20:1–20:54, 2013.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, 1999, vol. 1579, pp. 193–207.

[11] Microsoft Research, "Z3: An efficient SMT solver," http://research.microsoft.com/en-us/um/redmond/projects/z3/.

[12] "The Zot bounded model/satisfiability checker," http://zot.googlecode.com.

[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV 2: An OpenSource Tool for Symbolic Model Checking," in *Computer Aided Verification*, ser. LNCS, 2002, vol. 2404, pp. 359–364.

[14] O. Lichtenstein, A. Pnueli, and L. Zuck, "The Glory of the Past," in *Logics of Programs*, ser. Lecture Notes in Computer Science, 1985, vol. 193, pp. 196–218.

[15] F. Laroussinie, N. Markey, and P. Schnoebelen, "Temporal logic with forgettable past," in *Proc. of the Symposium on Logic in Computer Science*, 2002, pp. 383–392.

[16] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan, "Linear encodings of bounded LTL model checking," *Log. Meth. in Computer Science*, vol. 2, no. 5, pp. 1–64, 2006.

[17] C. Ghezzi, D. Mandrioli, and A. Morzenti, "TRIO: A Logic Language for Executable Specifications of Real-time Systems ," *Journal of Systems and Software*, vol. 12, no. 2, pp. 107 – 123, 1990.

[18] N. Een and N. Sorensson, "An extensible sat-solver," in *Proceedings of the 6th International Conference on Theory and Application of Satisfiability Testing (SAT'03)*, ser. Lecture Notes in Computer Science, 2003, no. 2919, p. 502–518.

[19] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, "Simple bounded LTL model checking," in *Formal Methods in Computer-Aided Design*, ser. Lecture Notes in Computer Science, 2004, vol. 3312, pp. 186–200.

[20] ——, "Simple is better: Efficient bounded model checking for past LTL," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, 2005, vol. 3385, pp. 380–395.

[21] K. Heljanko, T. Junttila, and T. Latvala, "Incremental and complete bounded model checking for full PLTL," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, 2005, vol. 3576, pp. 98–111.

[22] C. Heitmeyer and D. Mandrioli, *Formal Methods for Real-Time Computing*. New York, NY, USA: John Wiley & Sons, Inc., 1996.

[23] K. Y. Rozier and M. Y. Vardi, "LTL satisfiability checking," *International Journal on Software Tools for Technology Transfer*, vol. 12, no. 2, pp. 123–137, 2010.

[24] ——, "A multi-encoding approach for LTL symbolic satisfiability checking," in *FM 2011: Formal Methods*, ser. Lecture Notes in Computer Science, 2011, vol. 6664, pp. 417–431.

[25] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of LTL formulae to büchi automata," in *Formal Techniques for Networked and Distributed Sytems — FORTE 2002*, ser. Lecture Notes in Computer Science, 2002, vol. 2529, pp. 308–326.

[26] A. Duret-Lutz and D. Poitrenaud, "Spot: an extensible model checking library using transition-based generalized büchi automata," in *Proceedings of the Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, 2004, pp. 76–83.

[27] J. Li, L. Zhang, G. Pu, M. Vardi, and J. He, "LTL satisfiability checking revisited," in *Proccedings of the International Symposium on Temporal Representation and Reasoning (TIME)*, 2013, pp. 91–98.

[28] V. Schuppan and L. Darmawan, "Evaluating LTL satisfiability solvers," in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, 2011, vol. 6996, pp. 397–413.