On How Bit-Vector Logic Can Help Verify LTL-based Specifications

Mohammad Mehdi Pourhashem Kallehbasti*, Matteo Rossi†, and Luciano Baresi‡

Abstract—This paper studies how bit-vector logic (bv logic) can help improve the efficiency of verifying specifications expressed in Linear Temporal Logic (LTL). First, it exploits the notion of Bounded Satisfiability Checking to propose an improved encoding of LTL formulae into formulae of bv logic, which can be formally verified by means of Satisfiability Modulo Theories (SMT) solvers. To assess the gain in efficiency, we compare the proposed encoding, implemented in our tool $\mathbb{Z}$ot, against three well-known encodings available in the literature: the classic bounded encoding and the optimized, incremental one, as implemented in both NuSMV and nuXmv, and the encoding optimized for metric temporal logic, which was the "standard" implementation provided by $\mathbb{Z}$ot. We also compared the newly proposed solution against five additional efficient algorithms proposed by nuXmv, which is the state-of-the-art tool for verifying LTL specifications. The experiments show that the new encoding provides significant benefits with respect to existing tools.
Since the first set of experiments only used Z3 as SMT solver, we also wanted to assess whether the benefits were induced by the specific solver or were more general. This is why we also embedded different SMT solvers in $\mathbb{Z}$ot. Besides Z3, we also carried out experiments with CVC4, Mathsat, Yices2, and Boolector, and compared the results against the first and second best solutions provided by either NuSMV or nuXmv. Obtained results witness that the benefits of the bv logic encoding are independent of the specific solver. Bv logic-based solutions are better than traditional ones with only a few exceptions. It is also true that there is no particular SMT solver that outperformed the others. Boolector is often the best as for memory usage, while Yices2 and Z3 are often the fastest ones.

Index Terms—Formal Methods, Linear Temporal Logic, Bounded Satisfiability Checking, Bit-Vector Logic.

1 Introduction

Linear Temporal Logic [1] (LTL) plays a key role in computer science. It has been used for the specification and verification of (possibly safety-critical) programs [2], the generation of test cases [3], the synthesis of controllers [4], the formalization of notations (e.g., UML) [5], the run-time verification of systems [6], and as planning formalism [7]. However, one of the key factors that still hamper the widespread adoption of this formalism in practice is the limited efficiency and scalability of verification tools.

While various techniques have used automata in the past to formally verify LTL models [8], this work exploits the notion of *Bounded Satisfiability Checking* (BSC) [9], a variant of Bounded Model Checking (BMC) [10]. BSC requires that LTL formulae be suitably translated into formulae of another decidable logic, such as propositional logic, that 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 or SMT solver) for verification (up to bound k).

To tackle efficiency, this article presents *bit-vector logic* (bv logic) as means to encode LTL formulae and speed-up their verification. This logic allows SMT solvers to exploit the representation of the different temporal values of variables as vectors and to carry out simplifications and optimizations at word (vector) level. Our initial work [11] demonstrated the feasibility of the approach, proposed an initial encoding, and demonstrated it was able to scale better than the "usual" Boolean-based ones by exploiting Z3 [12] as SMT solver.

This paper moves a step forward and generalizes the outcome. It first proposes a new bv logic-based encoding, which significantly improves the original one [11]. Besides highlighting the novel aspects, we implemented it as additional plug-in of our bounded satisfiability checker $\mathbb{Z}$ot [13]. Its architecture helped us implement different encodings as independent plug-ins and carry out the experiments more easily. To assess the efficiency gain we carried out a first set of experiments, reported in Section 4, to compare the new encoding against solutions already proposed by $\mathbb{Z}$ot and by NuSMV [14] and nuXmv[1] [16], which are the de-facto standard for bounded verification of LTL specifications (we did not consider tools like SPIN [17] because they employ other, different verification techniques).

We used $\mathbb{Z}$ot for reusing the old bv logic-based encoding [11] and the "standard" LTL encoding [9]. We also used both NuSMV and nuXmv to try with three "classical", Boolean logic-based encodings available in the literature: (i) the classic bounded encoding [18]; (ii) the optimized encoding [19], and (iii) the improved and incremental version [12], [20]. We also exploited nuXmv for five additional verification algorithms that both adopt diverse verification techniques and exploit specific optimizations to solve particular problems.

*Corresponding author. Department of Computer Engineering, University of Science and Technology of Mazandaran, Behshahr, Iran. E-mail: pourhashem@mazust.ac.ir
†Politecnico di Milano, Dipartimento di Meccanica, Milano, Italy. E-mail: matteo.rossi@polimi.it
‡Politecnico di Milano, Dipartimento di Elettronica Informazione e Bioingegneria, Milano, Italy. E-mail: luciano.baresi@polimi.it
Part of the work was carried out while the first author was at Politecnico di Milano.

[1] nuXmv is an extension of NuSMV that comes with strong SAT-based algorithms as well as SMT-based verification techniques integrated with MathSAT5 [15].

Obtained results show that the new solution, implemented as $\mathbb{Z}$ot plug-in and based on Z3, is almost always the fastest option and consumes less memory. The most significant exception is the verification of the Fischer protocol, where the *k-live* solution proposed by nuXmv is the best because it is able to subsume the UNSAT result without necessarily iterating up to the maximum bound. Our experiments also suggest that this solution (*k-live*) only works well with a few small models.

The second set of experiments we carried out aimed to assess whether efficiency benefits were independent of the particular SMT solver used —Z3 in the initial set of experiments. This is why we exploited $\mathbb{Z}$ot one more time to implement plug-ins and compare the top five solvers in recent SMT competitions [21]: Boolector [22], Yices2 [23], Mathsat [15], CVC4 [24], and Z3.

In this paper we focus on the verification of LTL specifications, which are finite-state models. The bv logic-based encoding presented here has also been used to improve the efficiency of the verification technique of infinite-state models presented in [25]. We do not present this work in this paper for the sake of brevity.

All these experiments helped us reject the claim that the gain was mainly due to the efficiency of Z3, and clearly highlight the benefits of the bv logic encoding. Obtained results witness that the benefits are independent of the specific solver. Bv logic-based solutions are better than traditional ones with only a few exceptions. There is however no specific solver that outperformed the others. Boolector is often the best as for memory usage, while Yices2 and Z3 are often the fastest options.

To summarize, this article extends the work initially presented in [11] with: (i) an improved, and more efficient, bv logic encoding of LTL formulae; (ii) a new and more thorough set of experiments to compare the efficiency of our $\mathbb{Z}$ot- and Z3-based solution against the best Boolean logic-based approaches and additional algorithms (provided by nuXmv); and (iii) a wider comparison to assess the impact of different SMT solvers on the efficiency of the proposed solution.

The rest of this article is organized as follows. Section 2 introduces LTL, briefly sketches logic-based system verification, and describes the existing bounded Boolean-based encoding for LTL. Section 3 explains the improved bv logic-based encoding for LTL and highlights the differences with respect to the original one [11]. Section 4 describes the tools we used for evaluation, the experiments we carried out, and the results we obtained. Section 5 surveys related approaches and Section 6 concludes the article.

2 Preliminaries

2.1 Linear Temporal Logic

LTL [1] is a widely-used specification logic. In this article, we focus on the version with both future and past temporal operators: although past operators do not increase the expressiveness of the logic, they are advantageous for compositional reasoning [26]. In addition, LTL with past operators is exponentially more concise than its future-only counterpart [27].

An LTL formula ϕ is defined over a set of atomic propositions AP by means of the following grammar:

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

where $p \in AP$, $\neg$ and $\wedge$ have the usual meaning, $\mathbf{X}$ and $\mathbf{U}$ are the "next" and "until" future operators, and $\mathbf{Y}$ ("yesterday") and $\mathbf{S}$ ("since") are their past counterparts. Complex formulae are composed of sub-formulae: for example, $p\mathbf{U}\mathbf{X}(p \wedge q)$ comprises p, q, $p \wedge q$, and $\mathbf{X}(p \wedge q)$.

The semantics of LTL is given in terms of infinite sequences of sets of atomic propositions, or *words*. A word $\pi : \mathbb{N} \to 2^{AP}$ 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 π *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:

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

We say that a word π *satisfies* formula ϕ when it holds at the first instant of the temporal domain, i.e., when $\pi, 0 \models \phi$ holds. 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" ($\mathbf{F}$), "globally in the future" ($\mathbf{G}$) and "release" ($\mathbf{R}$) operators (and their past counterparts "eventually in the past" $\mathbf{P}$, "historically" $\mathbf{H}$ and "trigger" $\mathbf{T}$) are defined as follows: $\mathbf{F}\phi = \top\mathbf{U}\phi$, $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$, $\phi_1\mathbf{R}\phi_2 = \neg(\neg\phi_1\mathbf{U}\neg\phi_2)$, $\mathbf{P}\phi = \top\mathbf{S}\phi$, $\mathbf{H}\phi = \neg\mathbf{P}\neg\phi$, and $\phi_1\mathbf{T}\phi_2 = \neg(\neg\phi_1\mathbf{S}\neg\phi_2)$.

LTL is then often used to model (complex) systems and the properties they must comply with,in a so-called *descriptive* approach [28]. If formulae S and ϕ describe system and property to be checked, respectively, satisfiability checking can help prove if ϕ holds (or fails) for S, since a formula is *valid* iff its negation is unsatisfiable [28]. $S \Rightarrow \phi$, which captures the fact that property ϕ holds for S, can be proven valid if its negation ($S \wedge \neg\phi$) is shown to be unsatisfiable, otherwise a trace that satisfies $S \wedge \neg\phi$ would witness the failure of property ϕ for system S.

For the sake of simplicity, let us introduce a simple running example used throughout the paper to materialize the main concepts. A synchronous shift-register returns every received bit after a delay of two time instants. This system can be specified by the LTL formula $S : \mathbf{G}(in \Leftrightarrow \mathbf{X}\mathbf{X}out)$, which states that in holds at the current time instant iff out will hold at the second time instant from now. Consider property $P_1 : \mathbf{F}\mathbf{G}\neg in$, which asserts that there is a time instant in the future at which in stops occurring; one can easily show that P_1 does not hold for S by producing a counterexample in which in occurs infinitely often. This can be proven by checking the satisfiability of formula $S \wedge \neg P_1$, which leads to a counterexample. On the other hand, property $P_2 : \mathbf{FG}\neg in \Rightarrow \mathbf{FG}\neg out$, which states that, if in ceases to occur after a certain point in time, then out eventually ceases to occur, holds for S. Indeed, there is not a single trace of S in which P_2 is falsified, which means that $S \wedge \neg P_2$ is unsatisfiable.

2.2 Bounded Satisfiability Checking

Bounded Satisfiability Checking (BSC) is a well-known satisfiability checking technique. It is based on the idea of translating a temporal logic formula ψ into a formula of propositional logic that represents infinite, *ultimately periodic* models of ψ—i.e., sequences of states of the form $\pi = s_0 s_1 \dots s_{l-1}(s_l s_{l+1} \dots s_k)^\omega$, where k is a parameter called the *bound* of the model. As discussed in Section 2.1, then, if one wants to validate the specification of a system S against property ϕ using a BSC approach, the formula to be translated is $S \wedge \neg\phi$, and one must look for an ultimately periodic sequence of states $\pi = s_0 s_1 \dots s_{l-1}(s_l s_{l+1} \dots s_k)^\omega$ of S that violates ϕ. If a counterexample that witnesses 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. For example, back to the running example, property P_1 does not hold for S because of the counterexample $\pi = \{\}\{in\}\{\}(\{in, out\}\{\})^\omega$, where we have an in at the second time instant and from the forth time instant onwards both in and out occur every other time instant forever.

BSC can be easily carried out by an SMT solver by translating LTL formulae properly. The classic encoding technique into propositional logic [18] represents states $s_0 \dots s_l \dots s_k$, 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 $\alpha = s_0 s_1 \dots s_{l-1}$, $\beta = s_{l+1} \dots s_k$, 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 that are used to represent the values of all subformulae of the LTL formula to be checked for satisfiability at instants $0, 1, \dots k+1$. More precisely, given an LTL formula ϕ and a bound k, the encoding introduces $k+2$ formula variables $|[\psi]|_0, |[\psi]|_1, \dots |[\psi]|_{k+1}$ for each subformula ψ of ϕ to 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* $l_0, l_1, \dots, l_k$, 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 $l_0, l_1, \dots, l_k$ can be true. Other Boolean variables are introduced for convenience: the $k+1$ variables $InLoop_i$, with $0 \le i \le k$, are such that $InLoop_i$ is true iff position i is in the loop (i.e., $l \le i \le k$). Finally, variable $LoopExists$ is true iff the desired loop exists.

Table 1 introduces the constraints that are imposed on the Boolean variables introduced above to capture the semantics of LTL formulae. Constraints $|LoopConstraints|_k$ formalize the semantics of Boolean 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 [18], 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 satisfiability checking.

TABLE 1
Constraints defined to capture the semantics of LTL formulae.

$|$LoopConstraints$|_k$

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

$|$LastStateConstraints$|_k$

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

$|$PropConstraints$|_k$

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

$|$TempConstraints$|_k$ for future operators

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

$|$Eventualities$|_k$

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

$|$TempConstraints$|_k$ for past operators

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

$|$TempConstraints$|_k$ in the origin.

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

Constraints $|LastStateConstraints|_k$ 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. Constraints $|PropConstraints|_k$ capture the semantics of propositional connectives. For example, they 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. Note that the Boolean encoding was defined for LTL formulae in Positive Normal Form (PNF), that is, 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.

Constraints $|TempConstraints|_k$ define the semantics of the temporal operators, both future ($\mathbf{X}$, $\mathbf{U}$ and $\mathbf{R}$) and past ones ($\mathbf{Y}$, $\mathbf{S}$ and $\mathbf{T}$). The semantics of $\mathbf{U}$ and $\mathbf{R}$ is defined through their standard fixpoint characterization and through the introduction of the set of constraints $|Eventualities|_k$.

The latter constraints are used to ensure that, if $\psi_1 \mathbf{U} \psi_2$ holds in s_k, then ψ_2 occurs infinitely often, that is, it occurs somewhere in the loop. Similarly, if $\psi_1 \mathbf{R} \psi_2$ 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 $LoopExists \Rightarrow (|[\psi_1 \mathbf{U} \psi_2]|_k \Rightarrow \langle\langle F\psi_2\rangle\rangle_k)$ does not allow $\psi_1 \mathbf{U} \psi_2$ 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 $LoopExists \Rightarrow (|[\psi_1 \mathbf{R} \psi_2]|_k \Leftarrow \langle\langle G\psi_2\rangle\rangle_k)$ forces $|[\psi_1 \mathbf{R} \psi_2]|_k$ to hold if ψ_2 holds from position l on.

Similar constraints define the semantics of the past operators $\mathbf{Y}$, $\mathbf{S}$ and $\mathbf{T}$, which is symmetrical to their future counterparts. We also define operator $\mathbf{Z}$, which is necessary for formulae in PNF, which is simply a variant of $\mathbf{Y}$ such that $\mathbf{Z}\psi$ 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 (constraints $|TempConstraints|_k$ in the origin).

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. $|[\phi]|_0$.

2.3 Bit-Vector Logic

A bit-vector is an array whose elements are bits (Booleans). In bit-vector logic (bv logic), the size of a bit-vector (number of bits) is finite, and can be any nonzero number in $\mathbb{N}$. We use the notation $\overleftarrow{x}_{[n]}$ for the bit-vector $\overleftarrow{x}$ with size 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 i^{th} 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.

Bv 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 $\overleftarrow{x}^{[j:i]} = \mathbf{::}_{k=j}^{i} x^{[k]}$. 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 the rightmost/leftmost bit and inserts *zero* from the left/right. For example, $\gg 1100 = 0110$ and $\ll 1100 = 1000$. In general, $\ll^n \overleftarrow{x}$ (resp., $\gg^n \overleftarrow{x}$) is the operation that applies $\ll$ (resp., $\gg$) to $\overleftarrow{x}$ n times.

We also use bitwise operators like *negation* (!), *conjunction* (&), *disjunction* (|), *reduction or* ($\Uparrow$), and *reduction and* ($\Downarrow$). The *reduction and* operator is defined as $\Downarrow \overleftarrow{x}_{[n]} = \&_{i=0}^{n-1} \overleftarrow{x}^{[i]}_{[n]}$ (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 bv logic can be built using the usual Boolean connectives $\neg$, $\wedge$.

3 Bit-vector-based Encoding

Before introducing our new bv logic-based encoding, we want to motivate the choice of this logic.

The truth values of an LTL formula at the time instants from 0 to k are a series of $true$s or $false$s, and the value at a particular time instant is logically related to the values at the other instants. If one adopted a Boolean encoding, each value would be stored in an independent variable and the broader view is disregarded. While a bit-vector is a collection of Boolean values, the key difference lies in the way constraints are managed. If they are asserted on a set of (independent) Boolean values, the solver is blind to their interrelations and no simplifications can be carried out at word level. In contrast, when these values are stored in a single vector (word), SMT solvers can apply simplifications and optimizations (more) efficiently. Essentially, more information is provided to the solver in the latter case.

While a thorough assessment of the impact of these simplifications is out of the scope of this paper [29] (see also Section 4 for our empirical results), we invite the reader to focus on the trivially unsatisfiable LTL formula $((a\mathbf{U}b \vee \neg a \mathbf{R} \neg b)\mathbf{U}c) \wedge \neg \mathbf{F}c$. By definition, $a\mathbf{U}b$ is equivalent to $\neg(\neg a \mathbf{R} \neg b)$, which reduces $a\mathbf{U}b \vee \neg a \mathbf{R} \neg b$ to $\top$. Besides, $\top \mathbf{U} c$ is another form of $\mathbf{F}c$, which reduces the LTL formula to $\mathbf{F}c \wedge \neg \mathbf{F}c$, that is $\bot$. These simplifications are not easy for a solver, especially when the whole formula is asserted at the Boolean level. Since only Z3 shows its intermediate steps, we can report its behavior, but we argue it can be generalized. Z3 simplifies the Boolean formula produced by the classic Boolean encoding into another Boolean formula that then must be solved. In contrast, the bv logic formula produced by ***sbvzot*** is simplified and reduced to $\bot$, and thus the result is UNSAT, without solving any formula. With the Boolean encoding, the solver computes the Boolean variables for time instants i and $i+1$, which are false, by resolving different constraints. It is not aware that they both represent the same sub-formula ($\bot$) at various time instants. In a bv logic-based encoding, the solver knows that bit i and $i+1$ are zero, not by solving constraints at bit level (Boolean values), but by simplifying the formula at vector level since both bits are parts of the same bit vector ($\perp$).

This example shows that bv logic can indeed enable simplifications that Boolean logic does not. However, in this specific example, since the formula is quite small, the solving time is quite small. Section 4 witnesses that the bigger formulae become, the higher the gain is.

3.1 sbvzot

bvzot is the first bv logic-based encoding for LTL we developed [11], ***sbvzot*** (simple ***bvzot***) is the new encoding presented in this paper. ***sbvzot***: (i) does not use binary arithmetic operations (addition and subtraction), (ii) introduces as many bit-vectors as the number of subformulae in a formula (not only for its propositional letters), (iii) and adds "last state constraints" for all operators (not only for past ones). This encoding —which, from a purely syntactic point of view, is usually more concise than ***bvzot***— is the result of diverse experiments that explored different tweaks and solutions. ***sbvzot*** is overall the best one in terms of efficiency.

Similarly to the classic Boolean encoding of Section 2.2, ***sbvzot*** uses bit-vectors to represent the truth value of each subformula in time instants $[0, k+1]$. More precisely, to encode an LTL formula ϕ, for each subformula ψ of ϕ we introduce a bit-vector, $\overleftarrow{\langle\psi\rangle}_{[k+2]}$ (i.e., of size $k+2$), such that $\overleftarrow{\langle\psi\rangle}^{[i]}_{[k+2]}$, with $i \in [0, k+1]$, captures the value of subformula ψ at instant i[2].

In addition to a bit-vector for each subformula ψ, we also introduce a bit-vector, $\overleftarrow{\langle lpos\rangle}_{[k+2]}$, that contains (encoded in binary) position pos of the loop in interval $[0, k+1]$ and a bit-vector, $\overleftarrow{\langle inloop\rangle}_{[k+2]}$, where the bit at position i is 1 iff the position i is inside the periodic part. For the sake of uniformity, 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 $k+2$. Note that, given a formula ϕ, and its vector $\overleftarrow{\langle\phi\rangle}$, $\overleftarrow{\langle\phi\rangle}\&!\overleftarrow{\langle\phi\rangle} = \perp$ and $\overleftarrow{\langle\phi\rangle}|!\overleftarrow{\langle\phi\rangle} = \top$.

To define the value of bit-vector $\overleftarrow{\langle inloop\rangle}_{[k+2]}$ we introduce constraint $\overleftarrow{\langle inloop\rangle}_{[k+2]} = \ll^{pos} \overleftarrow{-1}_{[k+2]}$.

For example, Table 2 shows an exemplar trace, along with $\overleftarrow{\langle lpos\rangle}$, and $\overleftarrow{\langle inloop\rangle}$, where we assume that k is 4 and thus all bit-vectors have length 6 ($k+2$). This trace comes from a counterexample that shows P_1 does not hold for S in the running example. P_1 states that, for all executions of the system, at some point in stops occurring. This property can be trivially falsified by the shown counterexample, in which in occurs infinitely often, to be precise, every other time instant from time instant 3. The first two rows are the actual trace, and the rest shows how bit-vectors represent their corresponding subformulae. $\overleftarrow{\langle lpos\rangle}$ equal to 000011 means that the solver was able to find a loop at position 3. Consequently, $\overleftarrow{\langle inloop\rangle}$ is 111000, that corresponds to 111111 shifted to the left 3 ($lpos$) times. The table shows that in all bit-vectors that represent a subformula, the bit at position 3 (loop position, lpos) is equal to the one at position 5 ($k+1$), because of the last state constraint.

As mentioned in Section 2.2, constraints $|LoopConstraints|_k$, which impose the equality of states s_{l-1} and s_k, are introduced for optimization purposes, but they do not affect the correctness of the encoding. Since in our new encoding we assessed empirically they do not have beneficial effects on the efficiency of the verification, we did not use them, and $|SBVLoopConstraints|_k$ reduce to the definition of bit-vector $\overleftarrow{\langle inloop\rangle}$.

For every subformula ϕ being replaced by a fresh bit-vector, Table 3 introduces the sets of constraints in bv logic that define the value of ϕ. $|SBVPropConstraints|_k$ assume that the main connective in ϕ is a Boolean one. $|SBVTempConstraints|_k$, capture the semantics of temporal operators.

[2] Recall that $\overleftarrow{\psi}^{[0]}$ is the right-most (least significant) bit in $\overleftarrow{\psi}$, and $\overleftarrow{\psi}^{[k+1]}$ is the left-most (most significant) one.

TABLE 2
A counterexample that falsifies property P_1 of the running example.

subformula	**bit-vector**	**5 4 3 2 1 0**
in	$\overleftarrow{\langle in\rangle}$	**1 0 1 0 1 0**
out	$\overleftarrow{\langle out\rangle}$	**1 0 1 0 0 0**
$f_1 : \mathbf{X}out$	$\overleftarrow{\langle \mathbf{X}out\rangle}$	**0 1 0 1 0 0**
$f_2 : \mathbf{XX}out$	$\overleftarrow{\langle \mathbf{X}f_1\rangle}$	**1 0 1 0 1 0**
$in \Leftrightarrow \mathbf{XX}out$	$\overleftarrow{\langle in \Leftrightarrow f_2\rangle}$	**1 1 1 1 1 1**
	$\overleftarrow{\langle lpos\rangle}$	**0 0 0 0 1 1**
	$\overleftarrow{\langle inloop\rangle}$	**1 1 1 0 0 0**

TABLE 3
Constraints in bv logic that define the value of ϕ.

$|\mathbf{SBVPropConstraints}|_\mathbf{k}$

ϕ	bit-vector encoding
$\neg\psi$	$\overleftarrow{\langle\neg\psi\rangle} = !\overleftarrow{\langle\psi\rangle}$
$\psi_1 \wedge \psi_2$	$\overleftarrow{\langle\psi_1 \wedge \psi_2\rangle} = \overleftarrow{\langle\psi_1\rangle}\&\overleftarrow{\langle\psi_2\rangle}$
$\psi_1 \vee \psi_2$	$\overleftarrow{\langle\psi_1 \vee \psi_2\rangle} = \overleftarrow{\langle\psi_1\rangle}\vert\overleftarrow{\langle\psi_2\rangle}$

$|\mathbf{SBVTempConstraints}|_\mathbf{k}$

ϕ	bit-vector encoding
$\mathbf{Y}\psi$	$\overleftarrow{\langle\mathbf{Y}\psi\rangle} = \ll \overleftarrow{\langle\psi\rangle}$
$\psi_1\mathbf{S}\psi_2$	$(\overleftarrow{\langle\psi_1\mathbf{S}\psi_2\rangle}^{[k+1:1]} = (\overleftarrow{\langle\psi_2\rangle}^{[k+1:1]} \vert \overleftarrow{\langle\psi_1\rangle}^{[k+1:1]} \& \overleftarrow{\langle\psi_1\mathbf{S}\psi_2\rangle}^{[k:0]})) \wedge (\overleftarrow{\langle\psi_1\mathbf{S}\psi_2\rangle}^{[0]} = \overleftarrow{\langle\psi_2\rangle}^{[0]})$
$\mathbf{X}\psi$	$\overleftarrow{\langle\mathbf{X}\psi\rangle}^{[k:0]} = \overleftarrow{\langle\psi\rangle}^{[k+1:1]}$
$\psi_1\mathbf{U}\psi_2$	$(\overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k:0]} = \overleftarrow{\langle\psi_2\rangle}^{[k:0]} \vert \overleftarrow{\langle\psi_1\rangle}^{[k:0]} \& \overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1:1]}) \wedge (((\overleftarrow{\langle\psi_1\rangle}^{[k+1]} \vert \overleftarrow{\langle\psi_2\rangle}^{[k+1]} \vert !\overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]}) \& (!\overleftarrow{\langle\psi_2\rangle}^{[k+1]} \vert \overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]})) = 1) \wedge (\overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]} \Rightarrow\Uparrow (\overleftarrow{\langle\psi_2\rangle}\&\overleftarrow{\langle inloop\rangle}) = 1)$

Yesterday. Given the semantics of formula $\mathbf{Y}\psi$, where $\mathbf{Y}\psi$ holds at i iff ψ holds at $i-1$, the bit-vector for $\mathbf{Y}\psi$ 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). Consistent with the origin semantics of $\mathbf{Y}\psi$, the rightmost bit of $\ll \overleftarrow{\langle\psi\rangle}$ is 0.

Since. The encoding of $\mathbf{S}$ is recursively defined based on the fact that $\psi_1\mathbf{S}\psi_2$ holds in i iff either ψ_2 holds in i or ψ_1 holds in i and $\psi_1\mathbf{S}\psi_2$ holds in $i-1$. This recursive definition can be captured by $\bigwedge_{i=1}^{k}(\overleftarrow{\langle\phi\rangle}^{[i]} \Leftrightarrow (\overleftarrow{\langle\psi_2\rangle}^{[i]} \vee \overleftarrow{\langle\psi_1\rangle}^{[i]} \wedge \overleftarrow{\langle\phi\rangle}^{[i-1]}))$, that is equivalent to $\overleftarrow{\langle\phi\rangle}^{[k+1:1]} = (\overleftarrow{\langle\psi_2\rangle}^{[k+1:1]} \,|\, \overleftarrow{\langle\psi_1\rangle}^{[k+1:1]} \,\&\, \overleftarrow{\langle\phi\rangle}^{[k:0]})$. Along with this constraint, $\overleftarrow{\langle\phi\rangle}^{[0]} = \overleftarrow{\langle\psi_2\rangle}^{[0]}$ is asserted to make the encoding compliant with the origin semantics of $\psi_1\mathbf{S}\psi_2$.

Next. The encoding of formula $\mathbf{X}\psi$ is a bit-wise shift to the right of bit-vector $\overleftarrow{\langle\psi\rangle}$, i.e., $\mathbf{X}\psi$ holds at i iff ψ holds at $i+1$. The constraint that bit $\overleftarrow{\langle\phi\rangle}^{[k+1]}$ must be equal to the one at the loop-back position is asserted in the "last state constraints" that are presented later in this section.

Until. Similar to $\mathbf{S}$, the encoding of $\mathbf{U}$ is also defined recursively. $\psi_1\mathbf{U}\psi_2$ holds in i iff either ψ_2 holds in i or ψ_1 holds in i and $\psi_1\mathbf{U}\psi_2$ holds in $i+1$. This recursive definition can be captured by $\bigwedge_{i=1}^{k}(\overleftarrow{\langle\phi\rangle}^{[i]} \Leftrightarrow (\overleftarrow{\langle\psi_2\rangle}^{[i]} \vee \overleftarrow{\langle\psi_1\rangle}^{[i]} \wedge \overleftarrow{\langle\phi\rangle}^{[i+1]}))$, that is equivalent to $\overleftarrow{\langle\phi\rangle}^{[k:0]} = (\overleftarrow{\langle\psi_2\rangle}^{[k:0]} \,|\, \overleftarrow{\langle\psi_1\rangle}^{[k:0]} \,\&\, \overleftarrow{\langle\phi\rangle}^{[k+1:1]})$.

Based on the recursive definition of $\mathbf{U}$ at position $k+1$, two constraints should hold. First, if $\overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]}$ holds, then either $\overleftarrow{\langle\psi_1\rangle}^{[k+1]}$ or $\overleftarrow{\langle\psi_2\rangle}^{[k+1]}$ hold; this constraint, which in the following we indicate as $Constraint_1$, can be represented in bv logic as $(!\overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]} | (\overleftarrow{\langle\psi_1\rangle}^{[k+1]} \mid \overleftarrow{\langle\psi_2\rangle}^{[k+1]})) = 1$. Second, if $\overleftarrow{\langle\psi_2\rangle}^{[k+1]}$ holds, then $\overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]}$ also holds, i.e., $\overleftarrow{\langle\psi_2\rangle}^{[k+1]} \Rightarrow \overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]}$ holds. Therefore, a bv logic representation of this constraint (which we indicate in the following as $Constraint_2$) can be $(!\overleftarrow{\langle\psi_2\rangle}^{[k+1]} | \overleftarrow{\langle\psi_1\mathbf{U}\psi_2\rangle}^{[k+1]}) = 1$. The second and third lines of the encoding are essentially a conjunction of $Constraint_1$ and $Constraint_2$ expressed in bv logic.

If no additional constraints are imposed on the semantics of operator $\mathbf{U}$, $\overleftarrow{\langle\phi\rangle}$ can be true throughout the periodic part (i.e., $s\beta$ in $\alpha s\beta s$) without any position within it in which $\overleftarrow{\langle\psi_2\rangle}$ is true. For example, if we suppose that $k=2$, $\overleftarrow{\langle lpos\rangle} = 0001$, $\overleftarrow{\langle inloop\rangle} = 1110$, $\overleftarrow{\langle\psi\rangle}_2 = 0001$, and $\overleftarrow{\langle\psi\rangle}_1 = 1111$. According to the previous constraint (and the "last state constraint" introduced below), $\overleftarrow{\langle\phi\rangle} = \psi_1\mathbf{U}\psi_2$ can be either 0001 or 1111, but the latter value is not correct. In the classic encoding, this is fixed through the introduction of constraints $|Eventualities|_k$ (see Section 2.2). To avoid this problem, we add a constraint that asserts that $\overleftarrow{\langle\phi\rangle}^{[k+1]}$ is true only if there is at least one position in the periodic part where ψ_2 is true, that is, ψ_2 holds infinitely often. More precisely, we add constraint $\overleftarrow{\langle\phi\rangle}^{[k+1]} \Rightarrow \Uparrow (\overleftarrow{\langle\psi\rangle}_2 \& \overleftarrow{\langle inloop\rangle}) = 1$ to the encoding of operator $\mathbf{U}$. Consequently, incorrect values are ruled out, and in fact in the previous example $\overleftarrow{\langle\phi\rangle}$ cannot be 1111, since $\Uparrow(0001\&1110) = 0$.

The "last state constraints" ($|SBVLastStateConstraints|_k$), which must be added for all subformulae ψ of ϕ (including propositional letters), state that $\overleftarrow{\langle\psi\rangle}^{[lpos]} = \overleftarrow{\langle\psi\rangle}^{[k+1]}$.

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

I $|SBVLastStateConstraints|_k$;

II $|SBVLoopConstraints|_k$ to capture the definition of $\overleftarrow{\langle inloop\rangle}$;

III The constraints that define each subformula ($|SBVPropConstraints|_k$ and $|SBVTempConstraints|_k$);

IV Constraint $\overleftarrow{\langle\phi\rangle}^{[0]} = 1$, where $\overleftarrow{\langle\phi\rangle}$ is the bit-vector defined based on its subformulae.

For example, if we consider formula $\neg\mathbf{X}p \vee (q\mathbf{U}\mathbf{Y}p)$, its complete encoding $(\neg\mathbf{X}p \vee (q\mathbf{U}\mathbf{Y}p))_{\mathrm{sbv}}$ is given by the following formula:

$$
\begin{array}{c|c}
\mathrm{I} & \overleftarrow{\langle p\rangle}^{[lpos]} = \overleftarrow{\langle p\rangle}^{[k+1]} \quad\wedge\quad \overleftarrow{\langle q\rangle}^{[lpos]} = \overleftarrow{\langle q\rangle}^{[k+1]}\wedge \\
 & \overleftarrow{\langle \mathbf{Y}p\rangle}^{[lpos]} = \overleftarrow{\langle \mathbf{Y}p\rangle}^{[k+1]} \quad\wedge\quad \overleftarrow{\langle \mathbf{X}p\rangle}^{[lpos]} = \overleftarrow{\langle \mathbf{X}p\rangle}^{[k+1]}\wedge \\
 & \overleftarrow{\langle q\mathbf{U}\mathbf{Y}p\rangle}^{[lpos]} = \overleftarrow{\langle q\mathbf{U}\mathbf{Y}p\rangle}^{[k+1]}\wedge \\
\hline
\mathrm{II} & \overleftarrow{\langle inloop\rangle} = \ll^{lpos} \overleftarrow{-1}\wedge \\
\hline
\mathrm{III} & (\overleftarrow{\langle \mathbf{Y}p\rangle} = \ll \overleftarrow{\langle p\rangle})\wedge \\
 & (\overleftarrow{\langle q\mathbf{U}\mathbf{Y}p\rangle}^{[k:0]} = \overleftarrow{\langle \mathbf{Y}p\rangle}^{[k:0]} \,|\overleftarrow{\langle q\rangle}^{[k:0]} \,\&\overleftarrow{\langle q\mathbf{U}\mathbf{Y}p\rangle}^{[k+1:1]})\wedge \\
 & (\overleftarrow{\langle q\mathbf{U}\mathbf{Y}p\rangle}^{[k+1]} \Rightarrow\Uparrow (\overleftarrow{\mathbf{Y}p}\&\overleftarrow{\langle inloop\rangle}) = 1)\wedge \\
 & (\overleftarrow{\langle \mathbf{X}p\rangle}^{[k:0]} = \overleftarrow{\langle p\rangle}^{[k+1:1]}) \wedge (\overleftarrow{\langle \neg\mathbf{X}p\rangle} = !\overleftarrow{\langle \mathbf{X}p\rangle})\wedge \\
 & (\overleftarrow{\langle \neg\mathbf{X}p \vee (q\mathbf{U}\mathbf{Y}p)\rangle} = \overleftarrow{\langle \neg\mathbf{X}p\rangle}|\overleftarrow{\langle q\mathbf{U}\mathbf{Y}p\rangle})\wedge \\
\hline
\mathrm{IV} & \overleftarrow{\langle \neg\mathbf{X}p \vee (q\mathbf{U}\mathbf{Y}p)\rangle}^{[0]} = 1
\end{array}
$$

Similar to the classic Boolean encoding, the semantics of the other temporal operators is defined from the basic ones as abbreviations. In fact, based on our experiments, in the case of ***sbvzot***, introducing direct encodings for the derived temporal operators—as done in ***bvzot***—does not impact on the efficiency of the encoding, therefore we simply define the following: $\mathbf{F}\phi = \top\mathbf{U}\phi$, $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$, $\phi_1\mathbf{R}\phi_2 = \neg(\neg\phi_1\mathbf{U}\neg\phi_2)$, $\mathbf{P}\phi = \top\mathbf{S}\phi$, $\mathbf{H}\phi = \neg\mathbf{P}\neg\phi$, and $\phi_1\mathbf{T}\phi_2 = \neg(\neg\phi_1\mathbf{S}\neg\phi_2)$.

As for ***bvzot***, we also add constraint $\overleftarrow{\langle\phi\rangle} = \ll \overleftarrow{\langle\psi\rangle} | \overleftarrow{1}$ to capture the semantics of $\phi = \mathbf{Z}\psi$, in order to support PNF formulae (see Section 2.2).

3.1.1 Correctness and Complexity

We show the correctness of the encoding by proving a pair of results. First, we show that, when the encoding of a formula ϕ is satisfiable, the original formula is also satisfiable (*soundness* of the encoding); then, we prove that, if an ultimately periodic model of ϕ exists, then the encoding is satisfiable, provided that a sufficiently long bound k has been defined (which shows, to a certain extent, the *completeness* of the encoding).

To help the reader follow the proofs presented in this section, we exemplify some relevant cases through pictures showing some example bit-vectors and corresponding LTL models.

Theorem 1. Let ϕ be an LTL formula, and let $k \in \mathbb{N}$ be the bound for the encoding ϕ_{sbv}. If formula ϕ_{sbv} is satisfiable, then there is a model $\pi = \alpha s(\beta s)^{\omega}$ of ϕ such that $k+1 = |\alpha s\beta|$.

Proof: To show the result, we first define how α, s and β are defined from the bit-vectors satisfying ϕ_{sbv}, and then we show that $\pi \models \phi$ holds.

Figure 1 provides a graphical depiction of the correspondence between bit-vectors related to atomic propositions and words. Notice that, in all figures shown in this section, bit-vectors are depicted with the least significant bit on the

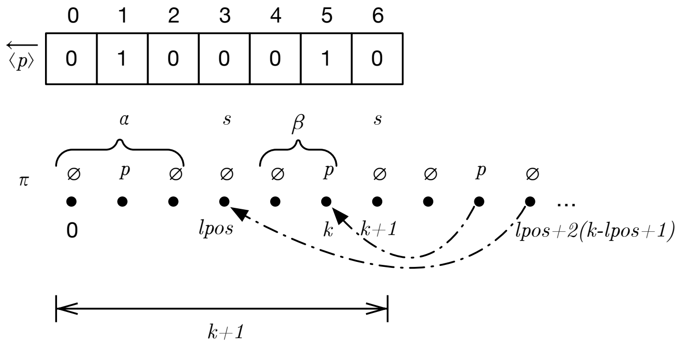

Fig. 1. Example of model π built from bit-vector $\overleftarrow{\langle p\rangle}$.

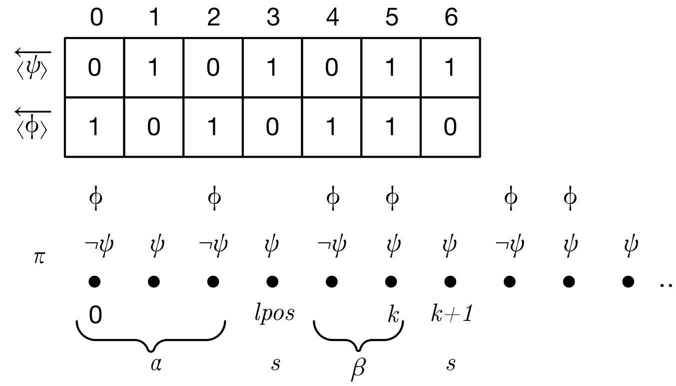

Fig. 2. Exemplification of case $\phi = \mathbf{X}\psi$.

left, instead of on the right, to facilitate the correspondence with words. Recall that $lpos$ is the loop-back position in π (where the first position in the bit-vector is 0), so we define $|\alpha| = lpos$ and $|\beta| = k - lpos$, and the length of the loop is $k - lpos + 1$. Word $\pi : \mathbb{N} \to 2^{AP}$ is defined in the following way: (i) for all $i \in \mathbb{N}$ such that $i \leq k$ holds, then $p \in \pi(i)$ (where $p \in AP$) if, and only if, $\overleftarrow{\langle p\rangle}^{[i]} = 1$ holds; (ii) for all i such that $i > k$, then $p \in \pi(i)$ holds if, and only if, $p \in \pi(j)$ also holds, where j is the unique value such that $lpos \leq j \leq k$ holds and there exists $m \in \mathbb{N}$ such that $i = j + m(k - lpos + 1)$ holds.

To show that $\pi \models \phi$ holds we prove, by induction on the structure of formula ϕ, that: (i) for all $i \in \mathbb{N}$ such that $i \leq k$ holds, then $\pi, i \models \phi$ holds if, and only if, $\overleftarrow{\langle \phi\rangle}^{[i]} = 1$ holds; and (ii) for all i such that $i > k$, $\pi, i \models \phi$ holds if, and only if, $\overleftarrow{\langle \phi\rangle}^{[j]} = 1$ also holds, where, as above, j is the unique value $lpos \leq j \leq k$ such that there is $m \in \mathbb{N}$ such that $i = j + m(k - lpos + 1)$ holds.

The base case $\phi = p$, with $p \in AP$, is trivial from the definition of π.

If $\phi = \neg\psi$, by definition we have that, for all $i \leq k$, $\pi, i \models \phi$ holds if, and only if $\pi, i \not\models \psi$, which, by induction, holds if, and only if, $\overleftarrow{\langle \psi\rangle}^{[i]} = 0$; by the definitions of Table 3, this occurs if, and only if, $\overleftarrow{\langle \phi\rangle}^{[i]} = 1$ holds. The cases for $i > k$ and for the propositional connectives $\wedge$ and $\vee$ are similar.

If $\phi = \mathbf{X}\psi$, then $\pi, i \models \phi$ if, and only if, $\pi, i+1 \models \psi$. Figure 2 exemplifies this case. If $i < k$ (say, $i = 2$ in Figure 2), then by induction hypothesis $\overleftarrow{\langle \psi\rangle}^{[i+1]} = 1$ holds and, by the definitions of Table 3, $\overleftarrow{\langle \psi\rangle}^{[i+1]} = \overleftarrow{\langle \phi\rangle}^{[i]} = 1$ holds. If $i = k$, then $i + 1 = k + 1 = lpos + (k - lpos + 1)$ (that is, $j = lpos$ and $m = 1$); then, by induction hypothesis, $\overleftarrow{\langle \psi\rangle}^{[lpos]} = 1$ holds and, by constraints $|SBVLastStateConstraints|_k$ and Table 3, $\overleftarrow{\langle \phi\rangle}^{[k]} = \overleftarrow{\langle \psi\rangle}^{[k+1]} = \overleftarrow{\langle \psi\rangle}^{[lpos]} = 1$. If $i > k$, we separate the case where $i \neq k + m(k - lpos + 1)$ (e.g., $i = 7$ in Figure 2, where $k = 5$ and $k - lpos + 1 = 3$) from the one where $i = k + m(k - lpos + 1)$ (e.g., $i = 8$ in Figure 2), which are shown in a similar manner as cases $i < k$ and $i = k$ above.

If $\phi = \mathbf{Y}\psi$, $\pi, i \models \phi$ holds if, and only if, $i > 0$ and $\pi, i-1 \models \psi$. If $i = 0$, then by definition $\pi, 0 \not\models \phi$; by Table 3, $\overleftarrow{\langle \phi\rangle}^{[0]} = 0$ (recall that the bit of index 0 is the right-most one, and the unsigned left shift operation $\ll$ inserts a 0 to the right), which shows the desired result. If $0 < i \leq k$ holds, then by induction hypothesis $\overleftarrow{\langle \psi\rangle}^{[i-1]} = 1$ holds and, by the definitions of Table 3, $\overleftarrow{\langle \psi\rangle}^{[i-1]} = \overleftarrow{\langle \phi\rangle}^{[i]} = 1$ holds. If $i > k$, we separate the cases $i = lpos + m(k - lpos + 1)$ and $i \neq lpos + m(k - lpos + 1)$. The latter is shown in a similar manner as case $0 < i \leq k$ above. If $i = lpos + m(k - lpos + 1)$, then $i - 1 = k + (m-1)(k - lpos + 1)$, so, by induction hypothesis, $\overleftarrow{\langle \psi\rangle}^{[k]} = 1$ holds; then, by constraints $|SBVLastStateConstraints|_k$ and Table 3, $\overleftarrow{\langle \phi\rangle}^{[lpos]} = \overleftarrow{\langle \phi\rangle}^{[k+1]} = \overleftarrow{\langle \psi\rangle}^{[k]} = 1$ holds.

If $\phi = \psi_1 \mathbf{U} \psi_2$, then $\pi, i \models \phi$ holds if, and only if, either $\pi, i \models \psi_2$ holds, or both $\pi, i \models \psi_1$ and $\pi, i+1 \models \psi_1 \mathbf{U} \psi_2$ hold. This case is exemplified in Figure 3. Consider the **case $\mathbf{i \leq k}$**. If $\pi, i \models \psi_2$ holds (in which case $\pi, i \models \phi$ also holds, as for $i = 1$ in Figure 3), by induction hypothesis $\overleftarrow{\langle \psi_2\rangle}^{[i]} = 1$ holds and, by the definitions of Table 3, $\overleftarrow{\langle \phi\rangle}^{[i]} = 1$ also holds. Otherwise, if $\pi, i \models \psi_1$ does not hold (in which case $\pi, i \models \phi$ does not hold, as for $i = 2$ in Figure 3), by induction hypothesis $\overleftarrow{\langle \psi_1\rangle}^{[i]} = 0$ holds and, by Table 3, $\overleftarrow{\langle \phi\rangle}^{[i]} = 0$ holds. If, instead, $\pi, i \models \psi_1$ holds (and $\pi, i \models \psi_2$ does not hold), then $\pi, i \models \phi$ holds if, and only if, $\pi, i+1 \models \psi_1 \mathbf{U} \psi_2$ holds; in addition, in this case, by Table 3 we have that $\overleftarrow{\langle \phi\rangle}^{[i]} = \overleftarrow{\langle \phi\rangle}^{[i+1]}$ holds. We separate two cases: $i < k$ and $i = k$. If $i < k$ (e.g., in position $i = 3$ in Figure 3), the previous considerations apply also at position $i + 1$, and we iterate them (notice that $\pi, i' \not\models \psi_2$, $\pi, i' \models \psi_1$, $\overleftarrow{\langle \psi_2\rangle}^{[i']} = 0$, $\overleftarrow{\langle \psi_i\rangle}^{[i']} = 1$ and $\overleftarrow{\langle \phi\rangle}^{[i']} = \overleftarrow{\langle \phi\rangle}^{[i'+1]}$ all hold for all positions $i \leq i' < k$ in which we iterate the reasoning). If $i = k$, we have that $\pi, k \models \phi$ holds if, and only if, $\pi, k+1 \models \psi_1 \mathbf{U} \psi_2$ holds; also, $\overleftarrow{\langle \phi\rangle}^{[k]} = \overleftarrow{\langle \phi\rangle}^{[k+1]}$ holds by Table 3. We show that either $\overleftarrow{\langle \phi\rangle}^{[k+1]} = 0$ and $\pi, k \not\models \phi$ both hold, or $\overleftarrow{\langle \phi\rangle}^{[k+1]} = 1$ and $\pi, k \models \phi$ do.

- If $\overleftarrow{\langle \phi\rangle}^{[k+1]} = 0$ holds then, by constraints $|SBVLastStateConstraints|_k$, $\overleftarrow{\langle \phi\rangle}^{[lpos]} = 0$ also holds. Then, by Table 3, $\overleftarrow{\langle \psi_2\rangle}^{[lpos]} = 0$ holds and at least one of $\overleftarrow{\langle \psi_1\rangle}^{[lpos]}$ and $\overleftarrow{\langle \phi\rangle}^{[lpos+1]}$ is also 0. If $\overleftarrow{\langle \psi_1\rangle}^{[lpos]}$ is 0, then, by inductive hypothesis, $\pi, k+1 \not\models \psi_2$ and $\pi, k+1 \not\models \psi_1$ hold (notice that $k + 1 = lpos + (k - lpos + 1)$), hence $\pi, k \not\models \phi$ also holds. If, instead, $\overleftarrow{\langle \psi_1\rangle}^{[lpos]}$ is 1, then $\overleftarrow{\langle \phi\rangle}^{[lpos]} = \overleftarrow{\langle \phi\rangle}^{[lpos+1]} = 0$, and we iterate the reasoning until either there is $lpos < i' \leq k$ such that $\overleftarrow{\langle \psi_1\rangle}^{[i']}$ is

0, or we conclude that for all $lpos \leq i' \leq k$ both $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 0$ and $\overleftarrow{\langle \psi_1 \rangle}^{[i']} = 1$ hold (this is the case exemplified in Figure 3). In both cases, by inductive hypothesis we conclude that $\pi, k \not\models \phi$ holds (notice that if, as in Figure 3, throughout interval $[lpos, k]$ $\overleftarrow{\langle \psi_2 \rangle}$ is 0 and $\overleftarrow{\langle \psi_1 \rangle}$ is 1, then by inductive hypothesis ψ_1 holds forever after k, but ψ_2 never does, so ϕ does not hold).

- If, instead, $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 1$ holds, then, by Table 3, $\overleftarrow{\langle \psi_1 \rangle}^{[k+1]} = 1$, or $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]} = 1$ hold. In the latter case, by inductive hypothesis, $\pi, k+1 \models \psi_2$ holds, so $\pi, k \models \phi$ also holds. In the former case, by constraints $|SBVLastStateConstraints|_k$, both $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} = 1$ and $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 1$ hold. By the constraints of Table 3, $\overleftarrow{\langle \psi_2 \rangle}^{[lpos]} = 1$ or $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} \& \overleftarrow{\langle \phi \rangle}^{[lpos+1]} = 1$ hold. The case $\overleftarrow{\langle \psi_2 \rangle}^{[lpos]} = 1$ (which is the same as $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]} = 1$) was handled previously. If $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} \& \overleftarrow{\langle \phi \rangle}^{[lpos+1]} = 1$ holds, then we iterate the reasoning. By constraint $(\overleftarrow{\langle \psi_1 \mathbf{U} \psi_2 \rangle}^{[k+1]} \Rightarrow\Uparrow (\overleftarrow{\langle \psi_2 \rangle} \& \overleftarrow{\langle inloop \rangle}) = 1)$ of Table 3, there must be an index $lpos \leq i' \leq k$ such that $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 1$ holds. Then, by inductive hypothesis $\pi, i' \models \psi_2$ and $\pi, i' + (k - lpos + 1) \models \psi_2$ hold (and $\pi, j \models \psi_1$ for all $k \leq j \leq i' + (k - lpos + 1)$), so $\pi, k \models \phi$ also holds.

	0	1	2	3	4	5	6
$\overleftarrow{\langle \psi_1 \rangle}$	0	0	0	1	1	1	1
$\overleftarrow{\langle \psi_2 \rangle}$	0	1	0	0	0	0	0
$\overleftarrow{\langle \phi \rangle}$	0	1	0	0	0	0	0

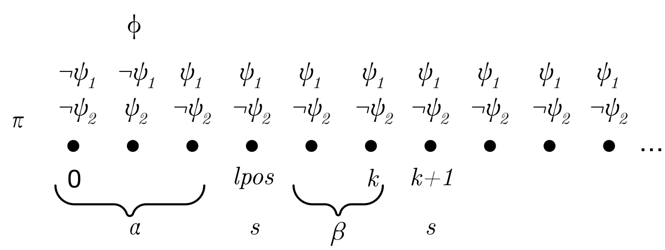

Fig. 3. Exemplification of case $\phi = \psi_1 \mathbf{U} \psi_2$ when $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 0$ holds.

Case $\mathbf{i > k}$, with $i = j + m(k - lpos + 1)$ is similar to the previous one, when one considers index j (for which $lpos \leq j \leq k$ holds) in place of i.

If $\phi = \psi_1 \mathbf{S} \psi_2$, then $\pi, i \models \phi$ holds if, and only if, either $\pi, i \models \psi_2$ holds, or both $\pi, i \models \psi_1$ and $\pi, i-1 \models \psi_1 \mathbf{S} \psi_2$ hold, provided that $i > 0$ holds. Notice that $\pi, 0 \models \phi$ holds if, and only if, $\pi, 0 \models \psi_2$ also holds. The proof for the case $i \leq k$ is similar to the one for subformula $\psi_1 \mathbf{U} \psi_2$, with the simplification given by the fact that, at position 0, the truth of $\psi_1 \mathbf{S} \psi_2$ is the same as that of ψ_2. The proof for the case $i > k$, with $i = j + m(k - lpos + 1)$, is similar to the case $i \leq k$, using $lpos \leq j \leq k$ instead of i. One only needs to consider that, if $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 1$ holds (which, by constraints $|SBVLastStateConstraints|_k$, entails that $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 1$ also holds), and if $\overleftarrow{\langle \psi_1 \rangle}^{[i']} = 1$ and $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 0$ hold for all $lpos \leq i' \leq k$ then, by inductive hypothesis, $\pi, t \models \psi_1$ and $\pi, t \not\models \psi_2$ hold for all $lpos \leq t \leq i$. However, since $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 1$ holds, using a similar reasoning as in the case of subformula $\psi_1 \mathbf{U} \psi_2$, one can show that there must be a position $0 \leq j' < lpos$ such that $\overleftarrow{\langle \psi_2 \rangle}^{[j']} = 1$ holds, and for all $j' < t' < lpos$ also $\overleftarrow{\langle \psi_1 \rangle}^{[t']} = 1$ holds. Then, by inductive hypothesis, $\pi, t' \models \psi_1$ holds for all $j' < t' < lpos$, $\pi, j' \models \psi_2$ holds, and $\pi, i \models \phi$ finally holds.

Finally, from the fact that $\overleftarrow{\langle \phi \rangle}^{[0]} = 1$, we have that $\pi, 0 \models \phi$ holds, that is, formula ϕ is satisfiable. □

In the following result, given a formula ϕ we indicate by $\delta(\phi)$ the nesting depth of past operators $\mathbf{Y}$ and $\mathbf{S}$. More precisely, if $\phi = p$ (with $p \in AP$), then $\delta(\phi) = 0$; if $\phi = \neg(\psi)$ or $\phi = \mathbf{X}\psi$, then $\delta(\phi) = \delta(\psi)$; if $\phi = \psi_1 \wedge \psi_2$, $\phi = \psi_1 \vee \psi_2$, or $\phi = \psi_1 \mathbf{U} \psi_2$, then $\delta(\phi) = \max(\delta(\psi_1), \delta(\psi_2))$; if $\phi = \mathbf{Y}\psi$, then $\delta(\phi) = \delta(\psi) + 1$; finally, if $\phi = \psi_1 \mathbf{S} \psi_2$, then $\delta(\phi) = \max(\delta(\psi_1), \delta(\psi_2)) + 1$. For example $\delta(\mathbf{YY}p) = 2$. We have the following result.

Theorem 2. Let ϕ be an LTL formula, whose depth of past operators is $\delta(\phi)$. Let $\pi = \alpha s(\beta s)^\omega$ be a model of ϕ and $k+1 = |\alpha(s\beta)^{\delta(\phi)+1}|$; then, ϕ_{sbv} is satisfiable, with bound for the encoding ϕ_{sbv} equal to k.

Before proving the result let us remark that, in this case, we are considering a bound k that is long enough to encode a sufficient number of iterations of the loop $s\beta$ (as evidenced by the condition $k + 1 = |\alpha(s\beta)^{\delta(\phi)+1}|$). This is due to the presence of past temporal operators $\mathbf{Y}$ and $\mathbf{S}$, which entail that $\delta(\phi) > 0$ holds; for a formula ϕ that does not include past temporal operators (for which $\delta(\phi) = 0$ holds), the result could be proved with simply $k + 1 = |\alpha s\beta|$. For example, consider formula $\bar{\phi} = \mathbf{GF}(\mathbf{YY}p)$ whose depth is $\delta(\bar{\phi}) = 2$. Word $\pi = p^\omega$ is a model for $\bar{\phi}$, but we need to encode at least 3 iterations of the loop to make $\bar{\phi}_{\text{sbv}}$ satisfiable.

Proof: To prove the result, we first define the values of the bit-vectors that appear in formula ϕ_{sbv}, and then we show that they satisfy the formulae of the encoding. More precisely, for every subformula ψ of ϕ, for every position $0 \leq i \leq k+1$, we define that $\overleftarrow{\langle \psi \rangle}^{[i]} = 1$ if, and only if, $\pi, i \models \psi$. Notice that, since we are requiring that $k + 1 = |\alpha(s\beta)^{\delta(\phi)+1}|$ holds, we are essentially considering model π to be $\pi = \alpha' s(\beta s)^\omega$, where $\alpha' = \alpha(s\beta)^{\delta(\phi)}$. Hence, we define $lpos = |\alpha'|$ (i.e., bit-vector $\overleftarrow{\langle lpos \rangle}$ is the binary encoding, over $k + 2$ bits, of value $|\alpha'|$), so that position $lpos$ corresponds to the start of the $\delta(\phi) + 1$-th iteration of the loop in π. Finally, we define $\overleftarrow{\langle inloop \rangle}^{[i]} = 1$ if, and only if, $i \geq lpos$. Figure 4 shows an example of bit-vector and parameters $lpos$, k defined from a word $\pi = \alpha s(\beta s)^\omega$, in the case where subformula ψ is a propositional letter and the depth is 2. Notice that, in the shown example, word π is a model for formula $\bar{\phi} = \mathbf{GF}(\mathbf{YY}p)$.

First of all, constraints $|SBVLoopConstraints|_k$ trivially hold by construction. Similarly for constraint $\overleftarrow{\langle \phi \rangle}^{[0]} = 1$, since by definition $\pi, 0 \models \phi$ holds.

The constraints of Table 3 ($|SBVPropConstraints|_k$) also obviously hold. Consider, for example, a subformula $\psi = \neg\psi'$. By definition, $\pi, i \models \psi$ holds if, and only if, $\pi, i \models \psi'$ does not hold. By construction, then, for all $0 \leq i \leq k+1$, $\overleftarrow{\langle \psi \rangle}^{[i]} = 1$ holds if, and only if, $\overleftarrow{\langle \psi' \rangle}^{[i]} = 0$.

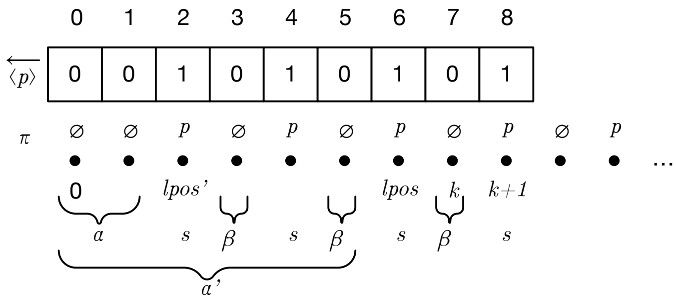

Fig. 4. Example of bit-vector $\overleftarrow{p}$ built from word π in a case where the depth δ is 2.

Consider now the constraints $|SBVTempConstraints|_k$ of Table 3. It is easy to see that, if $\psi = \mathbf{X}\psi'$ holds, constraint $\overleftarrow{\langle\psi\rangle}^{[k:0]} = \overleftarrow{\langle\psi'\rangle}^{[k+1:1]}$ also holds. In fact, by definition, $\pi, i \models \psi$ holds if, and only if, $\pi, i+1 \models \psi'$ does. Then, by construction, for all $0 \le i \le k$, $\overleftarrow{\langle\psi\rangle}^{[i]} = 1$ holds if, and only if, $\overleftarrow{\langle\psi'\rangle}^{[i+1]} = 1$ holds. Similarly if $\psi = \mathbf{Y}\psi'$; in this case, by definition $\pi, 0 \not\models \psi$ holds, and in fact constraint $\overleftarrow{\langle\psi\rangle} = \ll \overleftarrow{\langle\psi'\rangle}$ imposes that $\overleftarrow{\langle\psi\rangle}^{[0]} = 0$ holds due to the $\ll$ operator. The constraints of case $\psi = \psi_1 \mathbf{U} \psi_2$ also hold. Indeed, by definition $\pi, i \models \psi$ holds if, and only if, either $\pi, i \models \psi_2$ holds, or both $\pi, i \models \psi_1$ and $\pi, i+1 \models \psi_1 \mathbf{U} \psi_2$ hold. By construction, then, constraint $\overleftarrow{\langle\psi\rangle}^{[i]} = \overleftarrow{\langle\psi_2\rangle}^{[i]} \mid \overleftarrow{\langle\psi_1\rangle}^{[i]} \,\&\, \overleftarrow{\langle\psi\rangle}^{[i+1]}$ holds for all $0 \le i \le k$. At position $k+1$, either $\pi, k+1 \models \psi$ holds, or $\pi, k+1 \not\models \psi$ holds. If $\pi, k+1 \models \psi$ holds, by construction $\overleftarrow{\langle\psi\rangle}^{[k+1]} = 1$ holds, which means that $(!\overleftarrow{\langle\psi_2\rangle}^{[k+1]} \mid \overleftarrow{\langle\psi\rangle}) = 1$ also holds. In addition, since $\pi, k+1 \models \psi$ holds, either $\pi, k+1 \models \psi_2$ holds, or $\pi, k+1 \models \psi_1$ does, which assures that $(\overleftarrow{\langle\psi_1\rangle}^{[k+1]} \mid \overleftarrow{\langle\psi_2\rangle}^{[k+1]} \mid !\overleftarrow{\langle\psi\rangle}) = 1$ holds by construction. In addition, since $\pi = \alpha' s (\beta s)^\omega$ and $k+1 = |\alpha' s \beta|$ (so $k+1$ is the position of the second s in $\alpha' s \beta s$), $\pi, i' \models \psi_2$ must hold for some $lpos \le i' \le k$, or ψ_2 would never be true throughout suffix $(\beta s)^\omega$, so ψ would not hold at position $k+1$. Then, constraint $\overleftarrow{\langle\psi\rangle}^{[k+1]} \Rightarrow \Uparrow (\overleftarrow{\langle\psi_2\rangle} \& \overleftarrow{\langle inloop\rangle}) = 1$ holds by construction. If $\pi, k+1 \not\models \psi$ holds, $(\overleftarrow{\langle\psi_1\rangle}^{[k+1]} \mid \overleftarrow{\langle\psi_2\rangle}^{[k+1]} \mid !\overleftarrow{\langle\psi\rangle}) = 1$ holds by construction. In addition, $\pi, k+1 \models \psi_2$ cannot hold, so constraint $(!\overleftarrow{\langle\psi_2\rangle}^{[k+1]} \mid \overleftarrow{\langle\psi\rangle}^{[k+1]}) = 1$ holds. The proof for the constraints of case $\psi = \psi_1 \mathbf{S} \psi_2$ is similar (notice that $\pi, 0 \models \psi$ holds if, and only if, $\pi, 0 \models \psi_2$ does).

To conclude the proof, we need to show that constraints $|SBVLastStateConstraints|_k$ hold. To this end we first prove—by induction—something stronger. Let us call $lpos'$ the position of the first loop in $\pi = \alpha(s\beta)^\omega$, as depicted in Figure 4—that is, $lpos' = |\alpha|$ (recall that, instead, by construction $lpos$ is the position of the $\delta(\phi)+1$-th loop in π; also, notice that $k - lpos + 1 = |s\beta|$ holds). We show that, for each subformula ψ of ϕ, whose depth of past operators is $\delta(\psi)$, for all position $lpos' + \delta(\psi)(k - lpos + 1) \le i \le lpos' + (\delta(\psi)+1)(k - lpos + 1) - 1$, $\pi, i \models \psi$ holds if, and only if, $\pi, i + m(k - lpos + 1) \models \psi$, for all $m \in \mathbb{N}$. For example, with reference to Figure 4 (where $lpos' = 2$, $k - lpos + 1 = 2$), subformula $\mathbf{YY}p$, whose depth is 2, holds (resp., does not hold) at position 6 (resp., 7), and at all positions $6 + m2$ (resp., $7 + m2$); similarly, subformula $\mathbf{Y}p$, whose depth is instead 1, does not hold (resp., holds) at position 4 (resp., 5), and at all positions $4 + m2$ (resp., $5 + m2$)
The base case $\psi = p$ (with $p \in AP$) is trivial, since by definition $\pi(i) = \pi(i + m(k - lpos + 1))$ for all $i \le lpos'$. The inductive cases for propositional connectives and for future temporal operators are straightforward. For example, if $\psi = \psi_1 \mathbf{U} \psi_2$, then there is $i' \ge i$ such that $\pi, i' \models \psi_2$ holds, and $\pi, i'' \models \psi_1$ holds for all $i \le i'' < i'$. By inductive hypothesis, since $\delta(\psi) \ge \delta(\psi_1)$ and $\delta(\psi) \ge \delta(\psi_2)$ hold, this holds if $\pi, i' + m(k - lpos + 1) \models \psi_2$ holds, and $\pi, i'' + m(k - lpos + 1) \models \psi_i$ holds for all $i \le i'' < i'$, which corresponds to $\pi, i + m(k - lpos + 1) \models \psi$ holding.
If $\psi = \mathbf{Y}\psi'$, then $\pi, i \models \psi$ holds if, and only if, $\pi, i - 1 \models \psi'$ holds. Since $\delta(\psi) > \delta(\psi')$ holds, then $i > lpos' + \delta(\psi')(k - lpos + 1)$ holds so, by inductive hypothesis, $\pi, i - 1 \models \psi'$ holds if, and only if, $\pi, i - 1 + m(k - lpos + 1) \models \psi'$ holds, which in turn corresponds to $\pi, i + m(k - lpos + 1) \models \psi$ holding.
If $\psi = \psi_1 \mathbf{S} \psi_2$, then there is $i' \le i$ such that $\pi, i' \models \psi_2$ holds, and $\pi, i'' \models \psi_1$ holds for all $i' < i'' \le i$. If $i' \ge lpos' + (\delta(\psi) - 1)(k - lpos + 1)$ holds then, since both $\delta(\psi) - 1 \ge \delta(\psi_1)$ and $\delta(\psi) - 1 \ge \delta(\psi_2)$ hold, by inductive hypothesis both $\pi, i' + m(k - lpos + 1) \models \psi_2$ and $\pi, i'' \models \psi_1$ hold for all $i' + m(k - lpos + 1) < i'' \le i + m(k - lpos + 1)$, which entails that $\pi, i + m(k - lpos + 1) \models \psi$ holds. If, instead, $i' < lpos' + (\delta(\psi) - 1)(k - lpos + 1)$ holds, then $\pi, i'' \models \psi_1$ holds for all $lpos' + (\delta(\psi) - 1)(k - lpos + 1) \le i'' < lpos' + \delta(\psi)(k - lpos + 1)$, which, by inductive hypothesis since $\delta(\psi) > \delta(\psi_1)$ holds, entails that $\pi, \bar{i} \models \psi_1$ holds for all $\bar{i} \ge i'$; hence, $\pi, i + m(k - lpos + 1) \models \psi$ holds for all $m \in \mathbb{N}$.

Since, obviously, $\delta(\phi) \ge \delta(\psi)$ for all subformulae ψ of ϕ, and since, by construction, $k + 1 > lpos' + (\delta(\phi) + 1)(k - lpos + 1) - 1$ holds, then, for all subformulae ψ of ϕ, $\pi, k+1 \models \psi$ holds if, and only if, $\pi, lpos \models \psi$ holds, which by construction entails that $|SBVLastStateConstraints|_k$ hold. □

Concerning the size of the encoding ϕ_{sbv}, it is easy to see that, since we introduce a bit-vector constraint of constant size for each subformula ψ of ϕ, the total size is $O(n)$, with n the number of subformulae of ϕ—notice that the number n of subformulae of ϕ is, in the worst case, $O(l)$, with l the length of the formula, defined for example as the number of connectives and temporal operators appearing in ϕ (at worst, each subformula appears only once in ϕ).

4 EXPERIMENTAL EVALUATION

This section summarizes how we evaluated the efficiency of the encoding presented in this paper by comparing it against different state-of-the-art tools. Most of the experiments exploit our checker $\mathbb{Z}$ot, which 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 [30], which is equivalent to LTL, but more concise. The user feeds $\mathbb{Z}$ot with the specification to be checked and selects the plugin and the time bound (i.e., the value of bound k) to be used to perform the verification. $\mathbb{Z}$ot encodes the received specification in a target logic (e.g., propositional logic, or bv logic) and provides the result to a solver that is capable of handling the target logic. The result obtained by the solver is parsed back and presented to the user in a textual representation.

To assess the new encoding, we selected five benchmark specifications, two from the literature and two from our previous work. We wanted to work with complex specifications to better highlight the strengths and weaknesses of each tool. What follows is a brief presentation of the five case studies, but we refer the reader to cited literature for more details. These studies employ a BSC approach, that is, they use temporal logic to describe both the system under verification and the properties to be checked (Section 2.2).

Kernel Railway Crossing (KRC). This problem is frequently used for comparing real-time notations and tools [31]. A railway crossing system prevents vehicles from crossing the railway while trains are passing through it by controlling a gate. A temporal logic-based version of the KRC problem was developed in [9] for benchmarking purposes. It only considers one track, trains can only move in a direction, and uses an interlocking system. We experimented with two sets of time constants that allow different degrees of non-determinism, denoted as `krc2` and `krc3` in our experiments. The level of non-determinism is increased by using bigger time constants—e.g., the time a train takes to go through the railway crossing—that 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. It 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 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 the protocol, is 5 time instants. We then formally verified a safety property that states that it is never the case that two processes are simultaneously in their critical sections (`P1`). We identify the models of this case study through prefix `fischer`.

Ping Application. *Corretto*[3] is the toolset we developed to perform formal verification of UML models [5]. *Corretto* takes as input a set of UML diagrams and produces their formal representation through temporal logic formulae. In our tests we used the example diagrams introduced in [5] (a Class Diagram, an Object Diagram, and a Sequence Diagram with various combined fragments), which describe the behavior of an ping application that pings two servers and then sends queries to the server that responds 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 loop is 2 and 3, respectively. Property `P1` states that the search request is always sent to the server that replies earlier.

On Board Radar System. *Corretto* was also used in the EU-funded project MADES for the verification of two example Radar Systems, one on board the airplane and a ground-based one, provided by two industrial partners. In our tests, we used the on board system, and more precisely a component that carries out the delivery of the flight data from the environment to the User Interface (UI) of the pilot. Such a delivery is performed by a number of periodic tasks. The UML model (whose corresponding LTL formalization is identified by prefix `txt4` in our experiments) comprises a Class Diagram with five clocks, five Sequence Diagrams, and five State Machine Diagrams. The model identified by prefix `txt8` is similar, but larger, as it includes four more tasks—hence four more Sequence Diagrams and as many State Machine Diagrams. The different Sequence Diagrams illustrate how the data are read and processed by the different periodic tasks.

Human Robot Collaboration. This model (which is taken from [32]) formalizes the main elements a collaborative robotic system: a robot, a physical working area, a human operator, and a job executed by both the human and the robot. The model also includes definitions of hazardous physical contacts between the human and the robot based on the definitions of a few adopted ISO standards. Whenever the state of the model conforms with one of those definitions, a risk value that belongs to set {0,1,2} is assigned to the relevant hazard based on its attributes to estimate its harmfulness. Then, a risk reduction measure is activated when risk is 1 or 2 in order to reduce it to 0 in an acceptable amount of time. We use prefix `hrc` to identify the models of this case study.

4.1 Efficiency of the encoding

To evaluate the efficiency of ***sbvzot***, we implemented it as new $\mathbb{Z}$ot plugin and ran a first set of experiments to check the aforementioned benchmark by means of different tools. These first experiments exploit, in addition to *sbvzot*, the *meezot* and *bvzot* $\mathbb{Z}$ot plugins presented in [9] and [11], respectively: *meezot* implements an optimized encoding of LTL formulae into propositional logic, while *bvzot* implements our first bv logic-based encoding.

We also ran both NuSMV and nuXmv to test their implementations of the classic bounded encoding (*bmc*) [18], the corresponding optimized encoding (*sbmc*) [19], and its incremental version (*sbmcinc*[4]) [33]. We also used nuXmv for five additional, significant verification algorithms that mainly differ in the way they check LTL properties. *coisat* employs an incremental cone of influence reduction [34] to eliminate unrelated variables with respect to a given property. The flags used in the command specify that a SAT engine is used for both verification and trace execution. *coismt* is the same as *coisat*, but it uses an SMT engine. *klive* performs a K-Liveness algorithm with the IC3 engine, and produces a counterexample using the *bmc* algorithm. Note that this algorithm also checks the completeness bound. For example, at a given point it may conclude that the LTL formula is UNSAT and there is no need to check for larger bounds.

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

[4]While running this verification procedure we did not activate the completeness checking option since it often slows the verification down, as shown in [33].

msatcoi employs an SMT-based incremental cone of influence. *msat* is an SMT-based incremental *sbmc*.

Note that NuSMV, nuXmv and $\mathbb{Z}$ot also support other encodings for LTL/TRIO; we have chosen to show the results for the ones above because further experiments, not reported here for the sake of brevity, shown them to be, on average, the fastest ones for the tools. We also use `S` and `X` before the labels identified above to distinguish between NuSMV and nuXmv. To compare the performance of the different algorithms, we built a simple translator to convert specifications written in the $\mathbb{Z}$ot input language—such as those used in [9] and [5]—into the SMV language (the input language of NuSMV and nuXmv).

All experiments[5] were carried out on a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 16 GB RAM. In all 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 considered the logic formula ϕ_P that captures the property of interest, and we fed the verification tool 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.

Since NuSMV and nuXmv adopt a BMC (Bounded Model Checking) approach, we fed them with an empty system model (for which any trace is possible), together with either $\neg\phi_S$ or $\neg(\phi_S \wedge \neg\phi_P)$ as property to check [35]. Indeed, a BMC tool that receives a property ψ to be verified, first builds $\neg\psi$, then looks for a trace that satisfies $\neg\psi$. As a consequence, by feeding it $\neg\phi_S$ (resp., $\neg(\phi_S \wedge \neg\phi_P)$) as a property, the tool looks for a trace satisfying ϕ_S (resp., $\phi_S \wedge \neg\phi_P$), just like our tool does.

Table 4 shows the time (**T**) in seconds and memory (**M**) in MBs consumed in each of the experiments we performed[6]. Column **Model** concatenates the name of the particular model with the verification type (either SAT or property checking) and the maximum bound. For example, the first row (`krc2Sat_30`) shows time/memory consumption of each tool for the simple satisfiability checking of model `krc2` with the maximum bound $k = 30$. The two subsequent rows (`krc2P1_60` and `krc2P1_90`) are the results for the verification of property `P1` with maximum bound $k = 60$ and $k = 90$, respectively. The last row (**Solved Instances**) is the percentage of solved verification problems (models) by each tool on the five benchmarks. To help the reader rank the tools at a first glance, cell background colors indicate the best, second best, and third best tools.

For each experiment, we set a *maximum bound* k and the tools iteratively (possibly incrementally) tried 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 Table 4 had a time limit of 1 hour and a memory limit of 10GB RAM; that is, if the verification has taken longer than 1 hour or occupied more than 10GB of RAM, it would have been 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 because of **heap exhaustion** (**HE**) while pre-processing the specification to produce the encoding.

Table 4 suggests that the combination of ***sbvzot***/Z3 is not the fastest for 6 models, but altogether it only needs 53 more seconds to perform the verification of those 6 models. ***sbvzot***, however, is the fastest for the remaining 28 models and saves two hours in those experiments.

As Table 4 shows, among the algorithms implemented in NuSMV and nuXmv, *X-sbmcinc* is the one with the highest number of solved instances and mostly the one with the lowest memory consumption, whereas *X-sbmc* is the fastest on average. Indeed, *X-sbmcinc* solved 10 more models than *X-sbmc*; however, if one considers only the models on which both encodings are applicable, *X-sbmc* is usually faster than *X-sbmcinc*. Note that in the case of Fischer's protocol *X-klive* is the most efficient encoding, but overall it was able to solve only 11 models out of 34 (32%). All in all, we can conclude that the experimental results show a promising ability of ***sbvzot*** to scale as the size of the specification and the time bound increase.

We also carried out some additional experiments with the idea of letting ***sbvzot*** reach the 3600-second time limit. Figure 5 shows what happened for `txt4P1`, `txt8P1`, `sdserverl2P1`, and `sdserverl3P1`. ***sbvzot*** reached the limit at bounds 241 and 228 for `sdserverl2P1` and `sdserverl3P1`, respectively, and at bounds 115 and 105 for `txt4P1` and `txt8P1`, respectively. These values witness that the boundaries are very application-specific and give an idea of what the limits of ***sbvzot*** are.

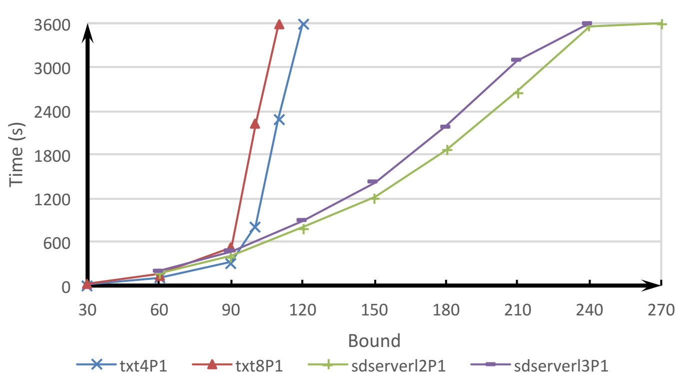

Fig. 5. Excerpts of how ***sbvzot*** behaves given a one-hour time window.

4.2 Independence of the SMT solver

One might claim that efficiency of our tool comes mainly from the underlying SMT solver (Z3), rather than from the encoding itself. To reject this claim, we examined the top five solvers in SMT competitions [21] in recent years, and thus, besides Z3 (version 4.8), we selected four additional SMT

[5]We used version 2.6 of NuSMV and version 1.1.1 of nuXmv. The SAT and SMT solvers used with $\mathbb{Z}$ot were, respectively, MiniSat version 2.2 and Z3 version 4.8. The code for all the experiments is available, along with all $\mathbb{Z}$ot plugins, from the $\mathbb{Z}$ot repository [13].

[6]Interested readers can refer to the $\mathbb{Z}$ot repository [13] for the complete and detailed data about the experiments.

TABLE 4
Time/memory comparison over the five benchmarks.

Tool / Model	sbvzot		bvzot		meezot		S-bmcinc		S-sbmc		S-sbmcinc		X-bmcinc		X-sbmc		X-sbmcinc		X-coisat		X-coismt		X-klive		X-msatcoi		X-msat	
	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M
krc2Sat_60	2	130	13	144	25	245	TO		48	988	159	2108	TO		24	906	147	2048	169	2750	17	777	58	108	18	777	178	1740
krc2P1_60	2	143	40	176	349	498			282	4988	651	3903			158	4525	589	3823	805	7046	367	3226	788	264	392	3226	2541	4225
krc2P1_90	95	175	649	213	1897	665			MO		2108	5665			482	9533	2160	5449	MO		TO		1118	264	TO		TO	
krc2P2_60	18	145	40	176	549	501			362	4979	739	3937			227	4500	641	3853	906	7194	457	3433	TO		457	3384	2600	4329
krc2P2_90	704	212	875	215	TO				MO		TO				2924	9757	TO		MO		TO				TO		TO	
krc3Sat_60	7	153	35	177	134	506			292	4888	1319	6552			172	4498	1346	6347	1419	9330	85	2364			85	2364	1504	6317
krc3P1_60	19	166	56	207	629	850			MO		2927	9639			MO		2814	9235	MO		265	5660			244	5660	MO	
krc3P1_90	102	151	565	261	HE						MO						MO				MO				MO			
krc3P2_60	18	169	60	218	566	988					TO						2953	9369			466	5904			441	5904		
krc3P2_90	229	187	392	259	HE						MO		MO				TO				MO				MO			
fischerSat_30	4	128	12	155	19	182	77	1531	4	153	7	298	24	1467	2	159	7	302	12	406	11	462	4	73	12	462	9	219
fischerP1_30	2	128	10	156	19	197	88	1656	5	160	4	306	24	1588	2	162	5	313	8	427	8	475	1	58	8	475	3	203
fischerP1_60	8	140	38	186	145	378	MO		66	557	13	539	MO		13	519	11	536	43	1006	52	1825	0	58	52	1801	11	389
fischerP1_90	19	155	86	217	457	554			58	1185	24	768			34	1076	24	758	131	1799	179	4711	1	58	179	4711	23	624
hrcSat_20	14	462	116	855	HE		235	1599	271	1015	265	1955	94	1591	132	999	126	1924	204	1970	212	1503	721	1537	212	1503	162	1550
hrcP1_30	144	1110	995	2066			MO		1235	5607	260	921	MO		1053	5202	231	4047	415	5015	1432	7390	930	1934	1434	7463	342	3907
hrcP1_60	596	1931	TO						MO		921	7651			MO		835	7356	MO		MO		1801	2536	MO		1646	7856
hrcP1_90	1832	2809									MO						MO						TO				MO	
txt4Sat_20	5	186	17	303	89	524			55	1543	62	1445	1826	9750	26	1189	58	1416	65	1621	30	886			30	886	56	929
txt4P1_30	18	236	85	425	373	1078			194	4650	137	2293	MO		101	3612	128	2204	158	2885	112	2293			112	2299	77	1527
txt4P1_60	114	338	623	737	HE				MO		435	3993			MO		411	3880	618	6824	MO				MO		289	3440
txt4P1_90	325	455	TO								931	5840					885	5494	MO								618	5769
txt8Sat_20	9	226	29	416	197	762	TO		114	2369	96	1797	TO		52	1859	88	1784	116	2041	68	1271			66	1271	115	1215
txt8P1_30	26	294	121	561	418	1078			431	7269	210	2844			189	5555	226	2777	245	3658	237	3545			239	3463	143	2080
txt8P1_60	150	460	1868	1015	HE				MO		665	5125			MO		630	4862	1044	8741	MO				MO		532	4768
txt8P1_90	528	661	TO								1421	7591					1352	7068	MO								1229	8460
sdserverl2Sat_50	11	212	87	547	115	603	145	2130	8	367	32	1104	104	1794	4	310	31	1085	32	1156	60	823	848	344	60	823	45	655
sdserverl2P1_60	169	649	1204	1244	HE		MO		572	9444	420	3910	MO		256	6899	396	3816	TO		TO		TO		TO		621	4464
sdserverl2P1_90	407	840	3059	1797					MO		945	5598			MO		883	5320									1806	8160
sdserverl2P1_120	791	1136	TO								1929	7521					1790	7097									MO	
sdserverl3Sat_50	16	255	135	558	269	832	1119	6254	17	639	55	1332	819	4984	9	517	50	1324	56	1466	102	1340			108	1340	79	924
sdserverl3P1_60	203	692	1401	1516	HE		MO		MO		551	4414	MO		324	7631	726	4190	TO		TO				TO		829	4968
sdserverl3P1_90	474	924	TO								1216	6442			MO		1128	6070									1682	9014
sdserverl3P1_120	896	1249									2232	8388					2081	8049									MO	
Solved Instances	**100%**		**79%**		**50%**		**14%**		**50%**		**85%**		**17%**		**58%**		**88%**		**52%**		**52%**		**32%**		**52%**		**73%**	

solvers. **Boolector** [22] (version 3) supports the quantifier-free theories of fixed-size bit vectors and arrays. This SMT solver won first place in divisions QF_ABV (main and application track), QF_BV (main track) and QF_UFBV (main and application track) in the 2018 SMT competition [36]. **Yices2** [23] (version 2.6) decides the satisfiability of formulae that contain uninterpreted function symbols with equality, real and integer arithmetic, bit vectors, scalar types, and tuples. It also supports nonlinear arithmetic, and has its own specification language (apart from SMT languages). **Mathsat** [15] (version 5.5) supports equality and uninterpreted functions, linear arithmetic, and bit vectors. It also provides additional features like extraction of unsatisfiable cores, generation of models and proofs, and the ability of working incrementally. **CVC4** [24] (version 1.6) is an automatic theorem prover for SMT problems. It supports first-order formulae in a large number of theories and combinations thereof. CVC4 is intended to be an extensible SMT engine.

Table 5 compares the five implementations of ***sbvzot***, that is, based on the five SMT solvers, against the first two best options provided by NuSMV or nuXmv. If no data is reported for NuSMV/nuXmv, these tools were not able to complete the verification process within the given time/memory limit. Again, cell background colors follow the same convention as before to ease the comprehension of the table. When one considers ***sbvzot*** in general, that is, with any underlying SMT solver, it is, on average, 2 times faster and 8 times more memory-efficient than the best algorithms of NuSMV and nuXmv (column **1st best**).

4.3 Lessons Learned

The results above allow us to draw some conclusions on the effectiveness of ***sbvzot***, and on the kinds of problems for which it seems particularly well suited.

We noticed a trade-off, at the level of the SMT solver, between the use of bit-blasting, which transforms bit-vector constraints into Boolean constraints, and the simplifications that can be obtained by using bit-vector arithmetic. For example, *bvzot* exploits greater simplifications at the bit-vector level because the encoding heavily depends on arithmetic operators (binary addition in the encoding of **U** and **S**). This results in more complex, heavier-to-handle Boolean formulae produced after bit-blasting. ***sbvzot*** mainly employs bit-wise operators, instead of bit-vector level arithmetic, and the Boolean formulae that are ultimately solved after bit-blasting are easier to handle. Although there are some simplification gains at the bit-vector level, the trade-off seems to favor bit-blasting over arithmetic simplification.

We also want to highlight that when we use MathSAT with our encoding, and NuSMV, that uses MathSAT itself, our use of MathSAT is actually faster. This is another evidence of how the use of bv logic and our encoding help verify (complex) LTL specifications.

5 Related Work

There are essentially two approaches to the problem of satisfiability checking of LTL formulae: bounded and complete ones. This paper pursues a bounded approach, and Section 4 compares ***sbvzot*** against similar ones, and in particular those presented in [18], [19], [20], [33] and [9].

TABLE 5
sbvzot, with the five SMT solvers, against the two bests results produced by NuSMV/nuXmv on the five benchmarks.

Tool / Model	sbvzot										NuSMV and nuXmv					
	CVC4		Mathsat		Yices2		Boolector		Z3		1st best			2nd best		
	T	M	T	M	T	M	T	M	T	M	T	M	Tool	T	M	Tool
krc2Sat_60	6	139	4	155	3	131	2	114	2	130	17	777	X-coismt	18	777	X-msatcoi
krc2P1_60	22	166	21	219	75	159	14	132	2	143	158	4525	X-sbmc	282	4988	S-sbmc
krc2P1_90	109	238	98	311	506	194	71	150	95	175	482	9533	X-sbmc	1118	264	X-klive
krc2P2_60	24	163	36	224	119	164	26	135	18	175	227	4500	X-sbmc	362	4979	S-sbmc
krc2P2_90	579	761	657	349	TO		383	187	704	212	2924	9757	X-sbmc	-----	-----	-----
krc3Sat_60	16	169	13	230	9	172	6	136	7	153	85	2364	X-coismt	85	2364	X-msatcoi
krc3P1_60	31	187	35	311	77	201	17	150	19	166	244	5660	X-msatcoi	265	5660	X-coismt
krc3P1_90	140	265	151	425	800	212	95	138	102	151	-----	-----	-----	-----	-----	-----
krc3P2_60	34	194	46	312	72	207	27	158	18	169	441	5904	X-msatcoi	466	5904	X-coismt
krc3P2_90	263	454	460	438	2452	231	268	158	229	187	-----	-----	-----	-----	-----	-----
fischerSat_30	5	140	3	136	2	123	3	117	4	128	2	159	X-sbmc	4	73	X-klive
fischerP1_30	4	139	3	141	1	122	2	114	2	128	1	58	X-klive	2	162	X-sbmc
fischerP1_60	15	159	15	177	6	145	8	128	8	140	0	58	X-klive	11	389	X-msat
fischerP1_90	35	184	47	226	21	171	20	140	19	155	1	58	X-klive	23	624	X-msat
hrcSat_20	25	461	15	534	7	388	11	341	14	462	94	1591	X-bmcinc	132	999	X-sbmc
hrcP1_30	273	940	159	1095	73	929	123	771	144	1110	231	4047	X-sbmcinc	260	921	S-sbmcinc
hrcP1_60	1831	1575	716	2031	452	1620	797	1360	596	1931	835	7356	X-sbmcinc	921	7651	S-sbmcinc
hrcP1_90	TO		2137	3227	2449	2506	2226	1937	1832	2809	-----	-----	-----	-----	-----	-----
txt4Sat_20	18	244	9	237	4	193	6	171	5	186	26	1189	X-sbmc	30	886	X-coismt
txt4P1_30	63	315	25	324	10	264	19	218	18	236	77	1527	X-msat	101	3612	X-sbmc
txt4P1_60	333	484	141	588	95	433	127	343	114	338	289	3440	X-msat	411	3880	X-sbmcinc
txt4P1_90	1176	650	427	943	446	646	432	459	325	455	618	5769	X-msat	885	5494	X-sbmcinc
txt8Sat_20	29	312	13	306	6	244	11	206	9	226	52	1859	X-sbmc	66	1271	X-msatcoi
txt8P1_30	115	408	44	428	14	341	31	271	26	294	143	2080	X-msat	189	5555	X-sbmc
txt8P1_60	789	650	217	917	144	602	206	471	150	460	532	4768	X-msat	630	4862	X-sbmcinc
txt8P1_90	2830	929	647	1528	630	875	666	632	528	661	1229	8460	X-msat	1352	7068	X-sbmcinc
sdserverl2Sat_50	23	306	18	280	8	218	10	193	11	212	4	310	X-sbmc	8	367	S-sbmc
sdserverl2P1_60	254	912	339	1114	255	817	232	598	169	649	256	6899	X-sbmc	396	3816	X-sbmcinc
sdserverl2P1_90	736	1273	895	1842	782	1277	566	811	407	840	883	5320	X-sbmcinc	945	5598	S-sbmcinc
sdserverl2P1_120	1270	1671	1818	2921	1511	1641	1196	1070	791	1136	1790	7097	X-sbmcinc	1929	7521	S-sbmcinc
sdserverl3Sat_50	37	369	19	346	10	262	17	233	16	255	9	517	X-sbmc	17	639	S-sbmc
sdserverl3P1_60	414	1014	387	1224	310	918	272	649	203	692	324	7631	X-sbmc	551	4414	S-sbmcinc
sdserverl3P1_90	858	1467	1013	2077	1001	1428	644	917	474	924	1128	6070	X-sbmcinc	1216	6442	S-sbmcinc
sdserverl3P1_120	1478	1836	2125	3208	1907	1817	1360	1199	896	1249	2081	8049	X-sbmcinc	2232	8388	S-sbmcinc
Solved Instances	**97%**		**100%**		**97%**		**100%**		**100%**		**91%**			**88%**		
Total T (s)	12853		9348		8556		6922		5090		12259			14907		
Total M (MB)	17694		24385		16702		12387		13978		117575			105568		

Common complete techniques include automata-based and tableaux-based approaches. An exhaustive evaluation of several techniques and tools (including some that are not based on translation to Büchi automata or on bounded approaches) for LTL satisfiability checking can be found in [37]. Although, given their difference in nature, we did not not compare our tools against complete ones, in this section we also provide a brief overview of the latter.

As for automata-based approaches (e.g., SPIN [17]), Rozier and Vardi [38] carried out a comparison of satisfiability checkers for LTL formulae based on the translation of LTL formulae into Büchi automata. Rozier and Vardi [39] also propose a novel translation of LTL formulae into Transition-based Generalized Büchi Automata, inspired by the translation presented in [40]. Such automata are used by SPOT [41], which is claimed to be the best explicit LTL-to-Büchi automata translator for satisfiability checking based on the experiments carried out in [38]. Li et al. [42] 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.

In tableau-based approaches, the LTL formula is analyzed on a tableau—that is, a set of nodes. The root node is labeled by the main LTL formula, and it is repeatedly decomposed based on the tableau rules that create successors labeled by a set of formulae. The LTL formula is satisfiable if, and only if, there exists at least one successful branch. Goranko et al. [43] report on the implementation and experimental evaluation of two well-known tableau-based approaches: Wolper's multi-pass, LTL tableau presented in [44], and Schwendimann's one-pass LTL tableau procedure presented in [45], with an evident superior performance to the latter.

Reynolds [46] introduces a novel traditional-style, one-pass, tree-shaped tableau for LTL. The fact that branches can be explored down independently makes this approach particularly suitable for parallel implementation, whereas Schwendimann's approach [45] requires the full development of branches.

Given the different nature of our approach with respect to automata- and tableaux-based ones we did not compare our tools against them, and focused on similar, BSC-based approaches instead.

A simple translation of LTL formulae to CNF (Conjunctive Normal Form) formulae is presented in [19], which deals with the semantic equivalence of LTL and Computation Tree Logic (CTL) when each step has only one successor in the Kripke structure. Another bounded encoding is presented in [20], which virtually unrolls the path up to the maximum depth of past operators (d) in the LTL formula. Unlike other bounded approaches (with bound k), this encoding unfolds the LTL formula up to $d * k$ steps, instead of k.

NuSMV [47] is a symbolic model checker that supports both BDD-based and SAT-based model checking. NuSMV can check LTL and CTL properties against finite state system models, so it can be used as a satisfiability checker for LTL and CTL formulae. Several algorithms are implemented in NuSMV for the satisfiability checking of LTL formulae. nuXmv [48] is an extension of NuSMV that supports both finite and infinite-state synchronous transition systems. nuXmv extends NuSMV by augmenting basic verification algorithms for finite-state systems and providing new data types and advanced SMT-based model checking techniques for infinite-state systems. Furthermore, nuXmv is the basis for various tools for requirements analysis, contract-based design, model checking of hybrid systems, safety assessment, and software model checking [16]. nuXmv offers more algorithms for checking the satisfiability of LTL formulae than NuSMV.

6 Conclusions

This paper presents a new encoding of LTL formulae in bit-vector logic. The encoding is used to solve the satisfiability problem for LTL formulae through a bounded approach. Besides demonstrating the benefits of the proposed encoding by comparing it against the original bv logic-based encoding and some well-known, more "classical" solutions, the paper also investigates the gains provided by the specific SMT checker adopted. While the original proposal exploits Z3, we also carried out experiments with Boolector, Yices2, Mathsat, and CVC4. Obtained results show that the benefits are mainly independent of the specific solver. All proposed checkers are implemented as dedicated plugins of ℤot, our bounded satisfiability checker.

References

[1] A. Pnueli, "The temporal logic of programs," in *Proc. of FOCS*, 1977, pp. 46–67.

[2] K. Y. Rozier, "Linear temporal logic symbolic model checking," *Comp. Sci. 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 *Proc. of IEEE IRI*, 2004, pp. 493–498.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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 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] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi, "Efficient Scalable Verification of LTL Specifications," in *Proc. of the 37th Int. Conf. on Soft. Eng.* IEEE Press, 2015, pp. 711–721.

[12] Microsoft Research, "Z3: An efficient SMT solver," https://github.com/Z3Prover/z3.

[13] "The Zot bounded model/satisfiability checker," https://github.com/fm-polimi/zot.

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

[15] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, "The mathsat5 smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, pp. 93–107.

[16] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *Proc. of CAV*, ser. LNCS, vol. 8559, 2014, pp. 334–342.

[17] G. J. Holzmann, *The SPIN model checker: Primer and reference manual*. Addison-Wesley Reading, 2004, vol. 1003.

[18] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan, "Linear Encodings of Bounded LTL Model Checking," *Log. Meth. in CS*, vol. 2, no. 5, pp. 1–64, 2006.

[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] "The annual smtcomp competition website." http://www.smtcomp.org.

[22] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 9, pp. 53–58, 2015.

[23] B. Dutertre and L. De Moura, "The yices smt solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[24] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 171–177.

[25] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi, "How Bit-vector Logic Can Help Improve the Verification of LTL Specifications over Infinite Domains," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, ser. SAC '16. New York, NY, USA: ACM, 2016, pp. 1666–1673. [Online]. Available: http://doi.acm.org/10.1145/2851613.2851833

[26] 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.

[27] 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.

[28] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling time in computing*. Springer Science & Business Media, 2012.

[29] L. De Moura and G. O. Passmore, "The strategy challenge in smt solving," in *Automated Reasoning and Mathematics*. Springer, 2013, pp. 15–44.

[30] 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.

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

[32] F. Vicentini, M. Askarpour, M. Rossi, and D. Mandrioli, "Safety assessment of collaborative robotics through automated formal verification," *IEEE Transactions on Robotics*, pp. 1–20, 2019, early access. [Online]. Available: https://doi.org/10.1109/TRO.2019.2937471

[33] 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.

[34] Kurshan, Robert P, *Computer-aided Verification of Coordinating Processes: the Automata-Theoretic Approach*. Princeton university press, 2014.

[35] A. Cimatti, M. Roveri, and D. Sheridan, "Bounded verification of past ltl," in *International Conference on Formal Methods in Computer-Aided Design*. Springer, 2004, pp. 245–259.

[36] "Smtcomp competition 2018." http://smtcomp.sourceforge.net/2018.

[37] V. Schuppan and L. Darmawan, "Evaluating LTL satisfiability solvers," in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, T. Bultan and P.-A. Hsiung, Eds. Springer Berlin Heidelberg, 2011, vol. 6996, pp. 397–413.

[38] 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.

[39] ——, "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.

[40] 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.
[41] 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.
[42] 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.
[43] V. Goranko, A. Kyrilov, and D. Shkatov, "Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis," *Electronic Notes in Theoretical Computer Science*, vol. 262, pp. 113–125, 2010.
[44] P. Wolper, "The tableau method for temporal logic: An overview," *Logique et Analyse*, no. 110–111, pp. 119–136, 1985.
[45] S. Schwendimann, "A new one-pass tableau calculus for PLTL," in *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 1998, pp. 277–291.
[46] M. Reynolds, "A traditional tree-style tableau for LTL," *arXiv preprint arXiv:1604.03962*, 2016.
[47] "The NuSMV model checker," http://nusmv.fbk.eu/.
[48] "The nuXmv model checker," https://nuxmv.fbk.eu/.

Mohammad Mehdi Pourhashem Kallehbasti is an assistant professor at University of Science and Technology of Mazandaran, Behshahr. He received his PhD degree in Software Engineering in 2015 from Politecnico di Milano. His research interests are in software engineering and formal methods.

Matteo Rossi is an associate professor at Politecnico di Milano. His research interests are in formal methods for safety-critical and real-time systems, architectures for real-time distributed systems, and transportation systems both from the point of view of their design, and of their application in urban mobility scenarios.

Luciano Baresi is a full professor at the Politecnico di Milano. Luciano was visiting professor at University of Oregon (USA) and visiting researcher at University of Paderborn (Germany). His research interests are in the broad area of software engineering and include formal approaches for modeling and specification languages, distributed systems, service-based applications and mobile, self-adaptive, and pervasive software systems.