How Bit-Vector Logic Can Help Improve the Verification of LTL Specifications over Infinite Domains *

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

ABSTRACT

Propositional Linear Temporal Logic (LTL) is well-suited for describing properties of timed systems in which data belong to finite domains. However, when one needs to capture infinite domains, as is typically the case in software systems, extensions of LTL are better suited to be used as specification languages. Constraint LTL (CLTL) and its variant CLTL-over-clocks (CLTLoc) are examples of such extensions; both logics are decidable, and so-called bounded decision procedures based on Satisfiability Modulo Theories (SMT) solving techniques have been implemented for them. In this paper we adapt a previously-introduced bounded decision procedure for LTL based on Bit-Vector Logic to deal with the infinite domains that are typical of CLTL and CLTLoc. We report on a thorough experimental comparison, which was carried out between the existing tool and the new, Bit-Vector Logic-based one, and we show how the latter outperforms the former in the vast majority of cases.

CCS Concepts

•**Software and its engineering** → **Model checking;**

Keywords

Formal Verification; Constraint LTL; Bounded Satisfiability Checking; Bit-Vector Logic; Logic Integration

1. INTRODUCTION

Propositional Linear Temporal Logic (LTL) has been a staple in computer science for decades [20]. Its uses include, among others, the specification of system properties [22], test case generation [24], run-time verification [6], and the formalization and verification of UML diagrams [4].

Two of the most important factors that still hamper the applicability of LTL-based approaches in practice are the limited efficiency and scalability of the corresponding verification tools, and the lack of expressiveness of the logic, which does not allow one to express, for example, variables with infinite domains (e.g., unbounded integers or reals). Recent work [5] on using Bit-Vector Logic as target formalism in a so-called bounded satisfiability approach for the formal verification of LTL specifications addresses the first issue. To address the second concern, LTL can be extended with infinite-domain variables so that the resulting logic is still decidable, hence verifiable in a fully automated way.

Let us introduce a **first motivating example** for the need for infinite domains in LTL. Consider the classic leader election protocol introduced in [17]. The goal of the protocol is to elect a leader in a ring of processes that exchange messages. Each process in the ring chooses a number, and communicates it to its immediate neighbor on one side. The processes then engage in a sequence of actions (receiving and sending numbers, comparing received numbers with stored ones) until the leader is elected. The selected leader is the one that had initially chosen the highest number among those in the ring. The protocol is guaranteed to elect a unique leader when the initially chosen numbers are all distinct. The protocol is rather simple, and it can be formally verified, for example, through the Spin model checker[1]. However, the Promela[2] model analyzed by Spin can only deal with finite ranges of integer values, so all we can verify through the tool is that, if the numbers assigned to processes are taken from a certain finite domain, the protocol will work correctly. Then, generalizing the result to any combination of integer numbers requires a manual step.

Constraint LTL (CLTL) [16] is a first-order extension of LTL that allows variables to take values from infinite domains such as integers or reals; the values assigned to variables at a time instant can be compared against each other (i.e., one can write constraints such as $x < y$), and against their future values (e.g., the value of a variable in the next instant). Recently, we have developed an effective decision procedure based on a bounded satisfiability approach for

SAC 2016,April 04-08, 2016, Pisa, Italy

http://dx.doi.org/10.1145/2851613.2851833

*Work partially supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869 (DICE).

[1]The protocol is one of the basic examples included in the tool distribution [23].

[2]Promela is the input language of the Spin model checker.

the automated verification of CLTL specifications [7]. To the best of our knowledge, this decision procedure is the first one to be implemented for CLTL; it is available as part of the Zot tool [25]. Through CLTL it is possible to create a model of the leader election protocol that allows for numbers assigned to processes in the ring to take values in an infinite domain such as the set of integers.

When variables can be real-valued, it becomes natural to introduce the possibility that they behave as *clocks* that measure the passing of time, as in Timed Automata [1]; that is, the value of each variable (i.e., clock) does not change through assignment, but it increases with the passing of time. In fact, real-valued clocks are a typical mechanism for quantitatively modeling the passing of time in systems that combine both software components and physical ones, such as embedded and cyber-physical systems. LTL-like temporal logics that include a notion similar to that of time-measuring clocks have been studied for several decades. Timed Propositional Temporal Logic (TPTL) [2] is a classic example of such logics, but it becomes undecidable when clocks are real-valued. Then, as a **second motivating example**, we would like to have a *decidable*, real-time, extension of LTL that allows us to capture the passing of time in a quantitative way, where time quantities are real-valued.

CLTL-over-clocks (CLTLoc) [9, 12] is the variant of CLTL where variables behave as real-valued clocks. It is decidable, expressively equivalent to Timed Automata [10], and a decision procedure for solving it is implemented in the Zot tool. Also, CLTLoc is the basis for the first implemented decision procedure that solves the satisfiability of continuous-time Metric Interval Temporal Logic (MITL) [8, 11].

The goal of this work is to bring the gains in time and memory efficiency provided by the bit-vector-based bounded encoding for LTL [5] to the decision procedures implemented for CLTL and CLTLoc. To this end, we introduce a novel Zot plugin that, like the previous tool, is built upon SMT solvers such as Z3 [18], and which exploits a combination of Bit-Vector Logic and Linear Integer/Real Arithmetic. The performance of the new tool is compared against that of the existing one, showing in many cases marked improvements in time and memory consumption.

The rest of this paper is organized as follows: Section 2 introduces CLTL, CLTLoc, and briefly describes the bit-vector-based bounded encoding for LTL; Section 3 introduces the new tool for deciding CLTL and CLTLoc; Section 4 presents and discusses the experimental results; Section 5 concludes the paper.

2. BACKGROUND

In this section we first introduce the CLTL logic and its extension where variables behave as clocks (CLTL-over-clocks). Then, we briefly introduce the Bit-Vector Logic-based encoding of LTL formulae that can be used to efficiently solve bounded satisfiability problems for LTL.

2.1 Constraint LTL (over clocks)

Constraint LTL (CLTL [16, 7]) is a decidable fragment of First-Order LTL. CLTL formulae are defined with respect to

$$
\begin{aligned}
(\pi,\sigma), i \models p &\Leftrightarrow p \in \pi(i) \text{ for } p \in AP \\
(\pi,\sigma), i \models R(\alpha_1,\ldots,\alpha_n) &\Leftrightarrow (\sigma(i+|\alpha_1|, x_{\alpha_1}), \\
&\qquad \ldots, \sigma(i+|\alpha_n|, x_{\alpha_n})) \in R \\
(\pi,\sigma), i \models \neg\phi &\Leftrightarrow (\pi,\sigma), i \not\models \phi \\
(\pi,\sigma), i \models \phi \wedge \psi &\Leftrightarrow (\pi,\sigma), i \models \phi \text{ and } (\pi,\sigma), i \models \psi \\
(\pi,\sigma), i \models \mathbf{X}\phi &\Leftrightarrow (\pi,\sigma), i+1 \models \phi \\
(\pi,\sigma), i \models \mathbf{Y}\phi &\Leftrightarrow (\pi,\sigma), i-1 \models \phi \wedge i > 0 \\
(\pi,\sigma), i \models \phi\mathbf{U}\psi &\Leftrightarrow \exists\, j \geq i : (\pi,\sigma), j \models \psi \ \wedge (\pi,\sigma), \\
&\qquad n \models \phi \ \forall\ i \leq n < j \\
(\pi,\sigma), i \models \phi\mathbf{S}\psi &\Leftrightarrow \exists\, 0 \leq j \leq i : (\pi,\sigma), j \models \psi \wedge \\
&\qquad (\pi,\sigma), n \models \phi \ \forall\ j < n \leq i
\end{aligned}
$$

Figure 1: Semantics of CLTL.

a finite set V of variables and a *constraint system* $\mathcal{D}$, which is a pair $(D, \mathcal{R})$ with D being a specific domain of interpretation for variables and constants and $\mathcal{R}$ being a family of relations on D, such that the set AP of atomic propositions coincides with set $\mathcal{R}_0$ of 0-ary relations. An *atomic constraint* is a term of the form $R(x_1,\ldots,x_n)$, where R is an n-ary relation of $\mathcal{R}$ on domain D and $x_1,\ldots,x_n$ are variables. A *valuation* is a mapping $v : V \to D$, i.e., an assignment of a value in D to each variable. A constraint is *satisfied* by v, written $v \models_{\mathcal{D}} R(x_1,\ldots,x_n)$, if $(v(x_1),\ldots,v(x_n)) \in R$. Given a variable $x \in V$ over domain D, *temporal terms* are defined by the syntax: $\alpha := c \mid x \mid \mathrm{X}\alpha$, where c is a constant in D and x denotes a variable over D. Operator X is very similar to the classic "next" operator $\mathbf{X}$ of LTL, but it only applies to temporal terms, with the meaning that $\mathrm{X}\alpha$ is the *value* of temporal term α in the next time instant. Notice that, to differentiate the "next" operator that is applied to formulae from the one that is applied to terms we write the former in bold, and the latter in plain font. Well-formed CLTL formulae are defined as follows:

$$\phi := R(\alpha_1,\ldots,\alpha_n) \mid \phi \wedge \phi \mid \neg\phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi$$

where α_i's are temporal terms, $R \in \mathcal{R}$, $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{U}$ and $\mathbf{S}$ are the usual "next", "previous", "until" and "since" operators of LTL, with the same meaning. Operators "eventually" $\mathbf{F}$, and "globally" $\mathbf{G}$ are defined as usual, i.e., $\mathbf{F}\psi$ is $\top\mathbf{U}\psi$ and $\mathbf{G}\psi$ is $\neg\mathbf{F}(\neg\psi)$.

The semantics of CLTL formulae is defined with respect to a strict linear order representing time $(\mathbb{N}, <)$. The truth values of propositions in AP, and values of variables belonging to V are defined by a pair (π,σ) where $\sigma : \mathbb{N} \times V \to D$ is a function that defines the value of variables at each position in $\mathbb{N}$ and $\pi : \mathbb{N} \to \wp(AP)$ is a function associating a subset of the set of propositions with each element of $\mathbb{N}$. The value of terms is defined with respect to σ as follows:

$$\sigma(i,\alpha) = \sigma(i+|\alpha|, x_\alpha)$$

where x_α is the variable in V occurring in term α and $|\alpha|$ is the *depth* of a temporal term, namely the total amount of temporal shift needed in evaluating α: $|x| = 0$ when x is a variable, and $|\mathrm{X}\alpha| = |\alpha| + 1$. The semantics of a CLTL formula ϕ at instant $i \geq 0$ over a linear structure (π,σ) is recursively defined as in Figure 1. A formula $\phi \in$ CLTL is

satisfiable if there exists a pair (π, σ) such that $(\pi, \sigma), 0 \models \phi$.

We are particularly interested in the cases where $\mathcal{D} = (\mathbb{Z}, \{<, =\})$ and $\mathcal{D} = (\mathbb{R}, \{<, =\})$, which are known to be decidable [16], and a decision procedure based on bounded satisfiability checking mechanisms has been defined in [7]. This decision procedure has been implemented in the `ae2zot` plugin of the Zot tool [25]. To illustrate the features of the language, the next CLTL formula states that, each time predicate *swap_a_and_b* holds, the values of variables a and b are swapped, that is, the next value of variable a is equal to the current value of variable b, and vice-versa:

$$\mathbf{G}(swap\_a\_and\_b \Rightarrow (\mathrm{X}a = b \wedge \mathrm{X}b = a)) . \qquad (1)$$

CLTL-over-clocks (CLTLoc for short) is a variant of CLTL, where $\mathcal{D} = (\mathbb{R}, \{<, =\})$, arithmetic variables are evaluated as *clocks*, and the arithmetic "next" operator X is not allowed. A clock "measures" the time elapsed since the last time the clock was "reset" (i.e., the variable was equal to 0). By definition, in CLTLoc each $i \in \mathbb{N}$ is associated with a "time delay" $\delta(i)$, where $\delta(i) > 0$ for all i, which corresponds to the "time elapsed" between i and the next state $i+1$. More precisely, for all clocks $x \in V$, $\sigma(i+1, x) = \sigma(i, x) + \delta(i)$, unless it is "reset" (i.e., $\sigma(i+1, x) = 0$). It is shown in [12] that CLTLoc is decidable. In addition, [10] shows that CLTLoc is equivalent to Timed Automata, so it is well suited for capturing timed specifications.

For example, the following CLTLoc formula states that, when predicate *turn_on* holds, a clock x is reset (i.e., it is equal to 0), and then predicate *on* holds until x hits value 5 (i.e., the light stays on for at least 5 time units):

$$\mathbf{G}(turn\_on \;\Rightarrow\; (x = 0 \wedge \mathbf{X}(\; (x > 0 \wedge on)\mathbf{U}(x = 5 \wedge on)\;))) . \qquad (2)$$

2.2 LTL Bounded Satisfiability Checking through Bit-Vector Logic

Let us briefly recall the bounded encoding of LTL formulae into Bit-Vector Logic formulae that was introduced in [5].

A bit-vector is an array of bits (Booleans). In Bit-Vector Logic, the size of a bit-vector (number of bits) is finite, and can be any nonzero number in $\mathbb{N}$. We denote by $\overleftarrow{x}_{[n]}$ the bit-vector x with size n; we simply write $\overleftarrow{x}$ when the size is not important or can be inferred from the context. $\overleftarrow{x}^{[i]}_{[n]}$ denotes the i^{th} bit in $\overleftarrow{x}$, 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.

Similarly to the classic Boolean encoding of [13], given an LTL formula ϕ, the goal is to capture models of ϕ that are *ultimately periodic*, i.e., of the form $\alpha(s\beta)^\omega$, where the length of $\alpha s \beta$ is $k+1$. To this end, since to capture the periodic nature of the model we look for bounded models of the form $\alpha s \beta s$, we use a bit-vector of size $k+2$ to represent the truth values of each subformula of ϕ at positions $[0, k+1]$. However, we only introduce as many bit-vectors as the number of atomic propositions in the formula, and describe the values of the non-atomic subformulae as transformations on the former vectors. More precisely, for each $p \in AP$, we introduce a bit-vector, $\overleftarrow{p}_{[k+2]}$, such that $\overleftarrow{p}^{[i]}_{[k+2]}$, with $i \in [0, k+1]$, captures the value of proposition p at instant i. In addition, we introduce a bit-vector, $\overleftarrow{loop}_{[k+2]}$, that contains (encoded in binary) the position of the loop in interval $[0, k+1]$ (i.e., the position of the first state s in $\alpha s \beta s$). The $\overleftarrow{loop}$ bit-vector appears as an integer in the encoding with the notation of l.

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

If the main connective in ϕ is a Boolean one, then we simply apply the corresponding bitwise operation to the bit-vectors of the subformulae of ϕ. For example, if $\phi = \neg\psi$, then $\overleftarrow{\phi} = !\overleftarrow{\psi}$, and if $\phi = \psi_1 \wedge \psi_2$, then $\overleftarrow{\phi} = \overleftarrow{\psi}_1 \& \overleftarrow{\psi}_2$ (where ! and & denote, respectively, the bitwise "not" and "and").

The next table shows the transformations in the case of both future ($\mathbf{X}$, $\mathbf{U}$) and past ($\mathbf{Y}$, $\mathbf{S}$) temporal operators, where :: and $\ll$ are, respectively, the concatenation and the "shift left" operators, | is the bitwise "or", and + represents the bitwise sum.

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

The encoding of the $\mathbf{U}$ operator uses the $\mathbf{U}_{nl}$ operation on bit-vectors that is defined as:
$\overleftarrow{x} \mathbf{U}_{nl} \overleftarrow{y} = \overleftarrow{y} \mid (\overleftarrow{x} \;\&\; !\mathrm{Rev}(\mathrm{Rev}(\overleftarrow{x} \mid \overleftarrow{y}) + \mathrm{Rev}\overleftarrow{y}))$, where $\mathrm{Rev}\overleftarrow{x}$ is the operation that reverses the bit-vector $\overleftarrow{x}$.

To complete the encoding, we need to introduce, for each past subformula and for each propositional letter, the so-called "last state constraints", and for each propositional letter the "loop constraints" (see also [13]). More precisely, for each formula $\mathbf{Y}\psi$, we add the constraint $(\ll \overleftarrow{\psi})^{[l]} = (\ll \overleftarrow{\psi})^{[k+1]}$, and similarly, *mutatis mutandis*, for each formula $\psi_1 \mathbf{S} \psi_2$. Also, for each $p \in AP$ we include the constraint $\overleftarrow{p}^{[l]} = \overleftarrow{p}^{[k+1]}$. Notice that "last state constraints" are built-in in the encoding of Boolean connectives and of temporal operators $\mathbf{X}$ and $\mathbf{U}$. Finally, to impose the "loop constraints" we simply add, for each $p \in AP$, the constraint $\overleftarrow{p}^{[l-1]} = \overleftarrow{p}^{[k]}$.

We refer the reader to [5] for all details of the encoding and the proof of its correctness.

3. COMBINING BIT-VECTORS AND ARITHMETIC TO SOLVE CLTL

The Zot tool [25] includes a plugin, called `ae2zot` (which stands for "arithmetic enhanced zot"), which is capable of deciding the satisfiability of both CLTL and CLTLoc formulae. To achieve this, the plugin implements the bounded approach described in [7, 12, 9]. To solve the satisfiability problem for a formula ϕ that belongs to either CLTL or

CLTLoc, the `ae2zot` plugin unfolds ϕ over a finite model of length $k+2$, where the last state of the model is the repetition of a previous one, and translates the unfolded formula into a formula of a suitable decidable logic. The target logic depends on the nature of ϕ. If ϕ is a CLTL formula where the domain of the variables is $\mathbb{N}$ (i.e., $\mathcal{D} = (\mathbb{N}, \{<, =\})$) or $\mathbb{Z}$, then the target logic is Quantifier-Free Linear Integer Arithmetic with Uninterpreted Functions (QFUFLIA); if ϕ is either a CLTL formula where the values of variables are in $\mathbb{R}$, or it is a CLTLoc formula, then the target logic is QFUFLRA (i.e., the arithmetic part is over the reals). The resulting QFUFLIA/QFUFLRA formula is fed to an off-the-shelf SMT solver such as Z3 [18].

To improve with respect to the `ae2zot` plugin, we have separated the encoding of the temporal operators, which is now done through the bit-vector-based approach presented in Section 2.2, from the representation of the arithmetic variables. We have called the resulting plugin, which mixes Quantifier-Free Bit-Vector Logic (QFBV) with QFUFLIA/QFUFLRA, `ae2bvzot`[3]. Let us briefly illustrate how the separation is carried out through some examples.

To separate the arithmetic layer from the temporal one, each arithmetic constraint is expressed through linear integer/real arithmetic logic formulae at each time instant; each constraint is then replaced by a fresh atomic proposition (essentially, a Boolean abstraction of the arithmetic constraint), which acts as placeholder in the temporal formula. As a result, we obtain a temporal logic formula that is free from any arithmetic constraints, which is conjoined with the assertions capturing these constraints in the corresponding logic (i.e., the concrete representation of the constraints). For example, consider formula $(\mathrm{X}x > x)\mathbf{S}(v = 1)$ — where x and v are integer-valued, time-dependent variables — that states that the value of x strictly increased since an instant in the past when the value of v was equal to 1. To encode it, according to [7] we introduce two sets of integer variables: $k+2$ integer variables $v_0, \ldots v_{k+1}$, which capture the value of v at every time instant in $[0..k+1]$, and $k+3$ integer variables $x_0, \ldots x_{k+2}$, which capture the value of x and $\mathrm{X}x$ at each position in $[0..k+1]$ (for example, the value of $\mathrm{X}x$ at position $k+1$ is given by x_{k+2}). These variables are used to impose the constraints that are necessary to capture the semantics of arithmetic variables/-clocks in CLTL and CLTLoc, as defined in [7, 12, 9]. For example, for clocks in CLTLoc, we need to introduce constraints that state that all clocks advance of the same quantity, unless they are reset. In addition, we introduce two bit-vectors, $\overleftarrow{bv}_{\mathrm{X}x>x}$ and $\overleftarrow{bv}_{v=1}$, which represent the value of the corresponding atomic formulae in $[0..k+1]$. Then, $\bigwedge_{i=0}^{k+1}(\overleftarrow{bv}^{[i]}_{\mathrm{X}x>x} \Leftrightarrow x_{i+1} > x_i)$ and $\bigwedge_{i=0}^{k+1}(\overleftarrow{bv}^{[i]}_{v=1} \Leftrightarrow v_i = 1)$ are asserted, and the value of formula $(\mathrm{X}x > x)\mathbf{S}(v = 1)$ is given by bit-vector $\overleftarrow{bv}_{v=1} \mid (\overleftarrow{bv}_{\mathrm{X}x>x}\ \&\ !((\overleftarrow{bv}_{\mathrm{X}x>x} | \overleftarrow{bv}_{v=1}) + \overleftarrow{bv}_{v=1}))$, as defined in Section 2.2.

The efficiency of our encoding mainly owes to the word-level simplification of the Bit-Vector Logic formulae that capture the temporal operators of the original CLTL formula. In fact, in the classic Boolean-based encodings there are groups of Boolean variables capturing the value of atomic propositions, much like our bit-vectors, but the solver is blind to their interrelations, because constraints are asserted at the bit-level. Therefore, in Boolean-based encodings there are no simplifications attempted at the level of the whole word, whereas SMT solvers efficiently handle such simplifications when atomic propositions are introduced as bit-vectors. For example, in the case of the **S** and **U** operators, we use binary additions and bitwise operations to provide a very concise encoding of the temporal operators.

To maximize the efficiency of an SMT solver, one needs to configure it properly by indicating what tactics should be used to solve the given problem. Some preliminary experiments we carried out, and discussed below, showed that this is especially true when the problem to be solved is expressed through the combination of different logics. There are numerous configuration parameters in SMT solvers, which in many cases are documented rather briefly, and trying all of them is almost infeasible. Choosing the most efficient configuration out of the many possible ones was a trial-and-error process guided by our intuition of what reasonable tactics could be. We do not claim that this process led us to the absolute best possible configuration of the SMT solver for checking CLTL/CLTLoc specifications. It is possible that even better configurations can be found by further studying the shape of the SMT problems that are produced by the bounded decision procedure for CLTL/CLTLoc, but we leave this for future work.

In the set-up phase of our experiments, then, we tried many combinations of different tactics to be used by the `ae2zot` and `ae2bvzot` plugins when invoking the SMT solver (3 in our case), to find the best ones in the two cases. The result was that we could hardly improve on the default configuration of Z3 when a single logic was involved (i.e., in the `ae2zot` case), but that the efficiency of the verification could be increased significantly with respect to the default configuration when multiple logics were used (i.e., for the `ae2bvzot` plugin).

Finally, we configured the `ae2bvzot` plugin so that, when Z3 is invoked, the tactics are applied in the following order: first we perform simplification and elimination of variables through the solving of equations; then, the solver performs bit-blasting to reduce the bit-vector expression into a Boolean satisfiability problem; and finally the solver uses a SAT-based tactic on this problem.

4. EXPERIMENTAL RESULTS

In this section we first briefly present the CLTL and CLTLoc case studies over which we compared the performance of the `ae2zot` and `ae2bvzot` plugins; then, we show the experimental results, and finally we draw some considerations on the results of the comparison.

4.1 Case Studies

We performed our comparison over four groups of examples, two concerning CLTL, and two concerning CLTLoc. For both CLTL and CLTLoc the corresponding two groups of examples differ in the way the CLTL/CLTLoc models have been produced: in one group, the models have been

[3]The `ae2bvzot` plugin is also available as part of the Zot distribution [25].

produced by hand from an informal description; in the second group, the temporal logic model has been automatically generated from another, formal or semi-formal, description.

More precisely, in the case of CLTL, we built by hand the models for two well-known examples, a bubblesort-style sorting algorithm, and the leader election protocol introduced in Section 1. We have also used specifications automatically generated from multi-diagram UML models using the approach described in [3].

In the case of CLTLoc, the model built by hand is a standard timed lamp which has been used many times for testing the performance of verification tools (see, e.g., [21]), and which has been given a CLTLoc description in [9]. The bulk of the CLTLoc experiments, however, used models that have been created using the transformation from continuous-time MITL specifications that has been defined in [8, 11]. In effect, these are experiments in verification of continuous-time MITL models, which exploit CLTLoc as intermediate language and use the corresponding decision procedure.

We remark that in all experiments the approach is entirely logic-based, and the verification is always an instance of the bounded satisfiability checking problem. That is, in all our examples both the system being analyzed and the property to be checked (if any) are expressed in temporal logic. This differs from so-called bounded model checking mechanisms, where the system is expressed in some kind of operational formalism, typically labeled transition systems.

In general, we perform two kinds of experiments: consistency checking ones (SAT), where we feed the verification tool with only the system model, without any property to be verified, and ask for an execution trace that witnesses the feasibility of the model (i.e., we check that the system has at least one admissible execution, hence it is not inconsistent); and classic property verification experiments, where we feed the tool with the system and the property to be verified (both described through temporal logic formulae), and we check whether the latter holds for the former or not (in which case the tool returns a trace witnessing the violation).

Let us briefly introduce the case studies we used in our experiments.

Sorting. This model specifies a sorting process of an array of fixed size N, using CLTL over $\mathcal{D} = (\mathbb{Z}, \{<, =\})$. This model is introduced in [7]. We indicate by $b, a \in \mathbb{Z}^N$ the array we want to sort and the array during each step of the sorting process, respectively, and by b_i the i-th element in b (similarly for a_i). The model consists in a sorting process that nondeterministically chooses an index $1 \leq s \leq N-1$ such that $a_s > a_{s+1}$ and swaps a_s with a_{s+1}. The sorting process keeps swapping unsorted adjacent elements until the whole array is sorted (Formula (1) is an example of CLTL formula capturing the swapping mechanism). The following is a sample property to be checked that says that eventually the array gets sorted:

$$\mathbf{F}\left(\bigwedge_{i=1}^{N-1}(a_i \leq a_{i+1}) \wedge \bigwedge_{i=1}^{N}\bigvee_{j=1}^{N} a_i = b_j\right). \tag{3}$$

In addition to the model in which the elements to be sorted are arbitrary integer numbers, we also performed experiments on a model which is built upon the same CLTL formulae, but where elements are real-valued; that is, in this second case we have that that $b, a \in \mathbb{R}^N$, and $\mathcal{D} = (\mathbb{R}, \{<, =\})$.

We also use a generalized version of this sorting process, in which instead of swapping only adjacent values a_s, a_{s+1}, the algorithm swaps a_s with possibly any a_z, provided that $z > s$ and $a_s > a_z$. In other words, any pair of unsorted elements can be nondeterministically selected for swapping.

Leader Election Protocol. This case study consists in the CLTL model (with $\mathcal{D} = (\mathbb{Z}, \{<, =\})$) of the leader election protocol described in Section 1. It is a CLTL version of the Promela model included in the Spin distribution [23].

Car Collision Avoidance System (CCAS). This example is taken from [3]. It is originally described in UML, then translated into CLTL through the technique presented in [3]. The example concerns a system that detects the distance of the vehicle on which it is installed, with respect to other objects such as cars and pedestrians. The distance between the car and the external objects is read by a sensor, which sends the data to the CCAS main module every 100 ms through the system bus. When the distance between the car and the external objects is greater than or equal to 2 meters the CCAS should perform no action. When the distance becomes strictly less than 2 meters the CCAS switches to the warning state. If the CCAS remains in the warning state for more than 300 ms and the distance is still less than 2 meters, the CCAS must brake the car. In this case, we use CLTL with $\mathcal{D} = (\mathbb{Z}, \{<, =\})$ to capture the data that is sent by the sensor to the main module, and that triggers the action. On this system, we want to prove the property that "if the distance remained less than 2 meters for T time units, then the system would brake within those same T time units", where T is a fixed positive integer, and each (discrete) time unit corresponds to 10 ms.

Leader Election Protocol - UML version. This is again the leader election protocol of Section 1, but first modeled in UML, then translated into CLTL.

Timed Lamp - CLTLoc version. This example is taken from [9, 12]. It consists of a lamp that is controlled by two buttons, *ON* and *OFF*, which cannot be pressed simultaneously. The lamp can be either on or off. When *ON* (resp. *OFF*) is pressed, the lamp is immediately turned on (resp. off). After *ON* is pressed, if no more buttons are pressed, it will automatically turn off with a delay Δ, a positive real constant. If the *ON* button is pressed again before the timeout expires, then the timeout is extended by a new delay Δ. Formula (2) is an example of CLTLoc formula for the timed lamp, where $\Delta = 5$. In this case, we check properties such as "the light never stays on for longer than Δ time units" and "if at some point the light stays on for longer than Δ time units, then *ON* is eventually pressed, and it is pressed again before Δ time units".

Timed Lamp - Continuous time MITL specification. This example is also taken from [9, 12]. It is a pure MITL specification of the previously described behavior of the timed lamp over so-called continuous time signals. In this example,

we exploit the MITL-to-CLTLoc satisfiability-preserving translation described in [8, 11] to carry out the verification.

Continuous time MITL specifications. These examples from [8, 11] exploit the aforementioned MITL-to-CLTLoc satisfiability-preserving translation. They consider "events" occurring in single instants over the real line (for example, predicate p occurring exactly when the current instant is a multiple of 100). We impose constraints such as "q must occur within 1 time unit (in the future or in the past) of p, then check properties such as "after each q there is another q within 100 time units". The examples include also the so-called "counting" operators, which allow users to state properties such as "q will hold at least n times in the next interval of length 1" (with n a constant, for example 2).

4.2 Experimental Setup and Results

Tables 1 and 2 show the result (**R**) of the verification, which can be satisfiable (S) or unsatisfiable (U), the time (**T**) in seconds and memory (**M**) in MBs consumed in each of the experiments we performed[4]. Table 1 shows the results for the experiments carried out with CLTL, whereas Table 2 presents those where the logic used was CLTLoc.

To help the reader get a quick overview of the results, we formatted the cells related to the `ae2bvzot` tool according to the following scheme. If tr (resp., mr) is the ratio between the time taken (resp., memory used) by `ae2zot` and that taken by `ae2bvzot`, then the format of the corresponding cell is the following:

- $tr \geq 2$ or $mr \geq 1.5$ (i.e., `ae2bvzot` is at least twice as fast as `ae2zot`, or occupies less than 2/3 of the memory): good;
- $1.1 \leq tr < 2$ or $1.1 \leq mr < 1.5$: moderately good;
- $0.91 < tr < 1.1$ or $0.91 < mr < 1.1$: comparable;
- $0.5 < tr \leq 0.91$ or $0.66 < mr < 0.91$: moderately bad;
- $tr < 0.5$ or $mr \leq 0.66$: bad.

Let us briefly explain the meaning of the identifiers used in the tables.[5] In Table 1, *S** rows capture the experiments with the model of the sorting algorithm where the elements are 5 integers ($b, a \in \mathbb{Z}^5$) and the bound (K) is 25, whereas in *S1-R-** rows the elements are real-valued ($b, a \in \mathbb{R}^5$) and K=30. The *S2-N-** identifier, instead, stands for the generalized version of the algorithm that can swap arbitrary numbers, where K=25. In all cases the postfix (*) is a number that identifies the check that was performed (e.g., pure SAT checking to see if the model is feasible, or checking of a specific property), whereas N (with $N \in \{5, 6, 7, 8\}$) is the length of the array. Similarly, *LN-** and *ULN-** identify the experiments performed using the model of leader election protocol described, respectively, "natively" in CLTL and through UML diagrams first; N corresponds to the number of elements in the ring, the postfix identifies the check performed, and K=70. Finally, rows labeled *CCN-** contain the results (pure satisfiability, verification of property p1 and p2) for the experiments with the CCAS example (K=200). There are 5 versions of this model, identified by number N, that differ from one another in the values of some temporal bounds, such as the maximum duration that the system stays in the warning state, the delay with which the brakes are activated, and the time constants T in the property checked.

Table 1: Comparison between `ae2zot` and `ae2bvzot` on CLTL specifications.

Model \ Tool	R	ae2zot		ae2bvzot		Model \ Tool	R	ae2zot		ae2bvzot	
		T(s)	M(MB)	T(s)	M(MB)			T(s)	M(MB)	T(s)	M(MB)
S1-1	S	2	155	**1**	167	CC5-sat	S	16	463	32	590
S1-2	U	99	181	**48**	199	CC5-p1	S	29	539	18	641
S1-3	S	10	175	**3**	191	CC5-p2	U	34	503	41	650
S1-R-1	S	6	180	**1**	177	L5-sat	S	28	381	**2**	**254**
S1-R-2	U	90	209	**6**	162	L5-p1	U	59	309	**2**	237
S1-R-3	S	13	209	**2**	183	L5-p2	U	19	333	**5**	257
S2-5-1	S	13	196	**5**	225	L10-sat	S	4997	2882	**7**	**412**
S2-5-2	U	34	184	**4**	203	L10-p1	U	1016	1084	**67**	**612**
S2-5-3	S	67	220	**19**	228	L10-p2	U	19837	1650	**161**	**667**
S2-5-4	S	38	208	21	235	L12-sat	S	15776	5535	**51**	**792**
S2-5-5	S	11	193	15	217	L12-p1	U	3686	1660	**400**	**1025**
S2-5-6	U	TO	TO	**1233**	**270**	L12-p2	U	TO	TO	**938**	**1131**
S2-6-1	S	46	227	**20**	253	L14-sat	S	TO	TO	**45**	**789**
S2-6-2	U	1219	227	**342**	253	L14-p1	U	18830	3128	**2183**	**1910**
S2-6-3	S	160	262	**51**	292	L14-p2	U	TO	TO	**7395**	**2930**
S2-6-4	S	204	250	**44**	275	L15-sat	S	TO	TO	**85**	**1282**
S2-6-5	S	47	223	**23**	245	L15-p1	U	TO	TO	**6797**	**3636**
S2-6-6	-	TO	TO	TO	TO	L15-p2	U	TO	TO	**18526**	**4679**
S2-7-1	S	51	264	42	312	L16-sat	S	TO	TO	**135**	**1450**
S2-7-2	-	TO	TO	TO	TO	L16-p1	U	TO	TO	**14558**	**4143**
S2-7-3	S	448	334	**61**	326	L16-p2	-	TO	TO	TO	TO
S2-7-4	S	253	301	**61**	331	UL5-sat	S	96	728	**10**	**434**
S2-7-5	S	270	272	**49**	298	UL5-p1	U	272	822	**11**	**446**
S2-7-6	-	TO	TO	TO	TO	UL5-p2	U	223	798	**11**	**428**
S2-8-1	S	1041	399	**104**	451	UL7-sat	S	559	1233	**51**	**803**
S2-8-2	-	TO	TO	TO	TO	UL7-p1	U	897	1381	**76**	**713**
S2-8-3	S	1338	467	**257**	539	UL7-p2	U	929	1417	**38**	**636**
S2-8-4	S	247	425	171	508	UL9-sat	S	2175	2384	**395**	**1304**
S2-8-5	S	1203	372	**131**	420	UL9-p1	U	4390	2736	**162**	**954**
S2-8-6	-	TO	TO	TO	TO	UL9-p2	U	4933	2900	**76**	**797**
CC1-sat	S	15	461	17	574	UL10-sat	S	5370	2974	**117**	**1228**
CC1-p1	S	30	565	23	635	UL10-p1	U	9918	3822	**452**	**1298**
CC1-p2	U	42	485	60	638	UL10-p2	U	12656	3871	**232**	**987**
CC2-sat	S	14	463	16	591	UL12-sat	S	TO	TO	**1465**	**3398**
CC2-p1	S	39	538	**19**	639	UL12-p1	U	TO	TO	**4049**	**2852**
CC2-p2	U	42	499	57	644	UL12-p2	U	TO	TO	**4264**	**3215**
CC3-sat	S	16	466	17	593	UL13-sat	S	TO	TO	**124**	**1949**
CC3-p1	S	32	548	19	649	UL13-p1	U	TO	TO	**7012**	**3643**
CC3-p2	U	41	511	55	660	UL13-p2	U	TO	TO	**8467**	**4008**
CC4-sat	S	14	460	30	572	UL14-sat	S	TO	TO	**2651**	**4936**
CC4-p1	S	39	535	20	638	UL14-p1	U	TO	TO	**12066**	**4305**
CC4-p2	U	41	492	36	638	UL14-p2	U	TO	TO	**25674**	**5952**

In the case of the CLTLoc experiments of Table 2, rows

[4] The code for all the experiments is available at http://home.deib.polimi.it/pourhashem.kallehbasti/sac-2016.php

[5] An extended, informal, description of each model – and of the verification performed – used in the tables can be found at http://home.deib.polimi.it/pourhashem.kallehbasti/ModelDescriptions.pdf

labeled LC^* correspond to the experiments carried out using the native CLTLoc specification of the timed lamp (K=70), while LM^* (K=200) are those where the model of the timed lamp is originally described through continuous-time MITL formulae translated into equisatisfiable CLTLoc formulae.

Rows labeled with Sp^* (K=30) and W^* (K=30) are verification experiments of MITL models capturing particular behaviors where phenomena behave as events ("spikes") or as rectangular waves. Labels $CX\text{-}^*$ (K=30), with $X \in \{1, 2, 3\}$, identify experiments starting from MITL specifications that also include the "counting" operator. Finally, models labeled with F^* (K=60) and Sq^* (K=20) are experiments that test the assumptions under which the MITL-to-CLTLoc encoding is defined. In particular, the former tests whether it is possible to produce traces in which a signal is *not* finitely variable (i.e., it can change an infinite number of times over a finite interval); and the latter whether it is possible to produce a square wave in which the intervals are left-open and right-closed, when the encoding assumes that intervals are left-closed and right-open.

Table 2: Comparison between `ae2zot` and `ae2bvzot` on CLTLoc specifications.

Model \ Tool	R	ae2zot		ae2bvzot		Model \ Tool	R	ae2zot		ae2bvzot	
		T(s)	M(MB)	T(s)	M(MB)			T(s)	M(MB)	T(s)	M(MB)
LC1	S	1	156	1	148	C3-1	U	65	315	89	309
LC2	U	1	159	2	151	C3-2	U	577	379	1035	356
LC3	S	2	157	**0**	149	Sp1	S	1	180	1	157
LC4	U	19	167	25	157	Sp2	S	13	311	**5**	293
LC5	U	18	160	17	154	Sp3	S	384	424	**117**	337
LM1	S	1	187	1	145	Sp4	U	554	479	654	403
LM2	S	1	185	**0**	156	Sp5	U	644	500	711	420
LM3	S	3	245	2	187	Sp6	U	983	520	757	420
LM4	U	98	282	55	223	Sp7	U	2920	498	1984	375
LM5	S	49	669	26	663	Sp8	U	3132	517	TO	TO
LM6	S	TO	TO	**594**	**1427**	Sp9	S	614	477	1120	384
LM7	U	246	308	235	244	Sp10	S	702	439	393	349
C1-1	S	142	347	78	284	Sp11	S	16	318	**5**	289
C1-2	U	249	406	646	377	W1	S	22	249	12	195
C1-3	U	1222	487	1582	416	W2	S	47	297	**18**	229
C1-4	S	204	363	**66**	312	W3	S	45	283	**1**	201
C1-5	U	849	395	1145	371	W4	S	323	290	259	221
C2-1	S	8	271	**1**	210	W5	S	20	246	14	198
C2-2	U	1487	291	1890	236	W6	S	145	329	**47**	247
C2-3	S	22	283	**5**	218	W7	S	267	406	**10**	**267**
C2-4	S	4	258	**2**	199	W8	S	141	350	**15**	250
C2-5	U	959	287	940	231	W9	S	681	368	**332**	267
C2-6	S	3	247	3	198	W10	S	137	330	**46**	248
C2-7	S	46	305	**3**	226	Sq1	U	6	197	4	172
C2-8	-	TO	TO	TO	TO	Sq2	U	251	289	481	226
C2-9	S	31	306	**5**	230	F1	U	4	242	**1**	183
C2-10	S	9	274	**3**	209	F2	U	32	381	**8**	308
C2-11	-	TO	TO	TO	TO	F3	U	120	467	**22**	422
C2-12	S	23	276	**2**	206						

All the experiments were carried out on a Linux desktop machine with a 3.4 GHz Intel® Core™ i7-4770 CPU and 8 GB RAM. All the reported runs had a timeout of 1 hour, i.e., if the verification took longer than 1 hour, it was aborted (**TO**). The models for the leader election protocol, however, are more time consuming as the number of nodes in the ring increases. Hence in this case, to make the comparison more meaningful, the time limit was set to 10 hours.

4.3 Lessons Learned

From the experimental results shown in Section 4.2 we can draw some considerations on the effectiveness of the new verification tool, and of the kinds of problems for which `ae2bvzot` seems particularly well suited.

First of all, we remark that the main feature of the `ae2bvzot` plugin is that it combines two different logics, Bit-Vector Logic for capturing the behavior of the temporal operators and arithmetic constraints for the first-order variables. Since SMT solvers do not support a logic that combines both the theory of bit-vectors and that of integer/real numbers, they do not have tactics that are specific for the combination of the two logics, so the solver needs to be guided in what tactics to apply to the problem to obtain the best results. This, in turn, suggests that in some cases the interplay between logics makes the solving less efficient than when using one single logic to capture all aspects, both arithmetic and temporal (as it is the case in `ae2zot`, which uses either only QFUFLIA or only QFUFLRA as target logic). For example, the position of the loop back is frequently used in the encoding and acts like a bottleneck in combining different layers/logics, since it appears as an integer variable in the arithmetic layer, and as a bit-vector in the temporal one. This emerges also from our experiments, where it seems that, when the arithmetic part of the model becomes more and more significant, the gains obtained with the `ae2bvzot` plugin decrease, or disappear entirely.

For example, the comparison between `ae2zot` and `ae2bvzot` becomes in general less favorable for the latter in the CLTLoc examples that have been derived by translation from continuous-time MITL specifications, as evidenced in Table 2 with respect to Table 1. In these cases, in fact, the number of arithmetic variables becomes considerable (multiple clocks, i.e., arithmetic variables, are introduced for each subformula of the original MITL formula); in addition, to manage the advancement of time the solver needs to take into account not only comparisons between values, but also more complicated operations such as addition of delays.

When the temporal, propositional part is predominant, instead, as in most of the CLTL case studies shown in Table 1, the gains that have been obtained by the purely bit-vector-based encoding presented in [5] manifest themselves also in the `ae2bvzot` plugin. In these models, especially the sorting and leader election cases, the arithmetic part is simpler, as it is essentially confined to establishing comparisons between values and to perform value assignments.

The CCAS case study requires a separate discussion. In fact, on this example the `ae2zot` plugin consistently outperforms `ae2bvzot`. We remark however that, unlike the sorting and the leader election examples, where the nature of the models is such that the various instances differ in their structures (the size of the array and the size of the ring of processes change, hence the number of arithmetic variables also changes), the various versions of the CCAS all have the same components and variables, and they differ only in the values of the temporal constants involved in the model. Hence, it is natural that, if `ae2zot` is more efficient

in one case (say, *CC-1-**), so is for the other cases. As for the reason why `ae2zot` is the best plugin for this case, we conjecture that it depends on the fact that the behavior of the arithmetic variables in the CCAS is rather rigid, as they are constrained to be piecewise constant, which in turn increases the interplay between the arithmetic and temporal parts of the model.

Finally, we remark that the `ae2bvzot` is, in many CLTL tests, slightly more memory consuming than `ae2zot`. However, the difference is, especially in the sorting case, still rather limited (mostly around 10%), with a steep increase in efficiency.

5. CONCLUSIONS

The ability of handling infinite-domain variables is more and more important in modern verification techniques to be able to express and check, from the early design phases, properties about components exchanging data. In this paper we have combined an efficient mechanism, based on bit-vectors, for handling propositional LTL — a logic suitable for expressing and verifying specifications over finite domains — with arithmetic constraints typical of first-order fragments of LTL, and in particular of CLTL and its extension CLTLoc. Our experimental results show that the resulting plugin of the ℤot tool, `ae2bvzot`, is in many cases an improvement over the previously available decision procedure, which did not exploit bit-vectors. This will allow us to increase the range and size of problems that can be tackled through the CLTL- and CLTLoc-based specification and verification approaches. In particular, the gains obtained through the novel tool will be exploited to bring verification techniques into the domain of so-called data-intensive applications [14], to analyze safety and security properties thereof.

Finally, we plan to investigate the possibility of exploiting a recent evolution of the NuSMV model checker, called nuXmv [15, 19] as the basis to implement the decision procedures for CLTL and CLTLoc. In fact, although nuXmv *per se* cannot handle precisely CLTL and CLTLoc models because it does not natively introduce certain conditions that are necessary for the decision procedures developed in [7, 12], we aim to use it as an engine to develop further, novel techniques for solving such models.

6. REFERENCES

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

[3] L. Baresi, A. Morzenti, A. Motta, and M. Rossi. A Logic-based Semantics for the Verification of Multi-diagram UML Models. *ACM Sw. Eng. Notes*, 37(4):1–8, 2012.

[4] L. Baresi, M. M. Pourhashem Kallehbasti, and M. Rossi. Flexible Modular Formalization of UML Sequence Diagrams. In *Proc. of FormaliSE*, pages 10–16, 2014.

[5] 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.*, pages 711–721. IEEE Press, 2015.

[6] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[7] M. M. Bersani, A. Frigeri, A. Morzenti, M. Pradella, M. Rossi, and P. San Pietro. Constraint LTL satisfiability checking without automata. *Journal of Applied Logic*, 12(4):522 – 557, 2014.

[8] M. M. Bersani, M. Rossi, and P. San Pietro. Deciding the satisfiability of MITL specifications. In *Proc. of GandALF*, pages 64–78, 2013.

[9] M. M. Bersani, M. Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. In *Proc. of TIME*, pages 99–106, 2013.

[10] M. M. Bersani, M. Rossi, and P. San Pietro. A logical characterization of timed (non-)regular languages. In *MFCS*, volume 8634 of *LNCS*, pages 75–86. 2014.

[11] M. M. Bersani, M. Rossi, and P. San Pietro. An SMT-based approach to satisfiability checking of MITL. *Inform. and Comp.*, 245:72–97, 2015.

[12] M. M. Bersani, M. Rossi, and P. San Pietro. A tool for deciding the satisfiability of continuous-time metric temporal logic. *Acta Informatica*, pages 1–36, 2015.

[13] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan. Linear Encodings of Bounded LTL Model Checking. *Log. Meth. in CS*, 2(5):1–64, 2006.

[14] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. Pérez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladušič. DICE: Quality-driven development of data-intensive cloud applications. In *Proc. of MiSE*, pages 78–83, 2015.

[15] 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*, volume 8559 of *LNCS*, pages 334–342, 2014.

[16] S. Demri and D. D'Souza. An automata-theoretic approach to constraint LTL. *Information and Computation*, 205(3):380–415, 2007.

[17] D. Dolev, M. Klawe, and M. Rodeh. An o(n log n) unidirectional distributed algorithm for extrema finding in a circle. *J. of Alg.*, 3(3):245 – 260, 1982.

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

[19] The nuXmv model checker. https://nuxmv.fbk.eu/.

[20] A. Pnueli. The temporal logic of programs. In *Proc. of FOCS*, pages 46–67, 1977.

[21] M. Pradella, A. Morzenti, and P. San Pietro. Bounded Satisfiability Checking of Metric Temporal Logic Specifications. *ACM TOSEM*, 22(3):20:1–20:54, 2013.

[22] K. Y. Rozier. Linear temporal logic symbolic model checking. *Comp. Sci. Review*, 5(2):163–203, 2011.

[23] The spin model checker. http://spinroot.com.

[24] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *Proc. of IEEE IRI*, pages 493–498, 2004.

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