Extending NUXMV with Timed Transition Systems and Timed Temporal Properties

Alessandro Cimatti, Alberto Griggio,
Enrico Magnago, Marco Roveri(✉),
and Stefano Tonetta

Fondazione Bruno Kessler, Trento, Italy
roveri@fbk.eu

Abstract. NUXMV is a well-known symbolic model checker, which implements various state-of-the-art algorithms for the analysis of finite- and infinite-state transition systems and temporal logics. In this paper, we present a new version that supports timed systems and logics over continuous super-dense semantics. The system specification was extended with clocks to constrain the timed evolution. The support for temporal properties has been expanded to include $\text{MTL}_{0,\infty}$ formulas with parametric intervals. The analysis is performed via a reduction to verification problems in the discrete-time case. The internal representation of traces has been extended to go beyond the lasso-shaped form, to take into account the possible divergence of clocks. We evaluated the new features by comparing NUXMV with other verification tools for timed automata and $\text{MTL}_{0,\infty}$, considering different benchmarks from the literature. The results show that NUXMV is competitive with and in many cases performs better than state-of-the-art tools, especially on validity problems for $\text{MTL}_{0,\infty}$.

1 Introduction

NUXMV [1] is a symbolic model checker for the analysis of synchronous finite- and infinite-state transition systems. For the finite-state case, NUXMV features strong verification engines based on state-of-the-art SAT-based algorithms. For the infinite-state case, NUXMV features SMT-based verification techniques, implemented through a tight integration with the MATHSAT5 solver [2]. NUXMV has taken part to recent editions of the hardware model checking competition, where it has shown to be very competitive with the state-of-the-art. NUXMV also compares well with other model checkers for infinite-state systems. Moreover, it has been successfully used in several application domains both in research and industrial settings. It is currently the core verification engine for many other tools (also industrial ones) for requirements analysis, contract based design, model checking of hybrid systems, safety assessment, and software model checking.

In this paper, we put emphasis on the novel extensions to NUXMV to support timed synchronous transition systems, which extend symbolically-represented infinite-state transition systems with clocks. The main novelties of this new version are the following. The NUXMV input language was extended to enable the description of symbolic

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11561, pp. 376–386, 2019.
https://doi.org/10.1007/978-3-030-25540-4_21

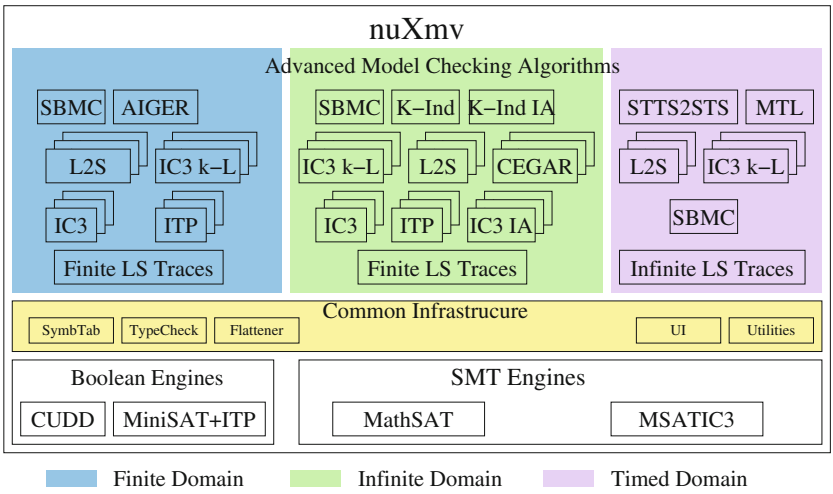

Fig. 1. The high level architecture of NUXMV.

synchronous timed transition systems with super-dense time semantics (where signals can have a sequence of values at any real time t). The support for temporal properties has been expanded to include $\text{MTL}_{0,\infty}$ formulas with parametric intervals [3,4]. Therefore, NUXMV now supports model checking of invariant, LTL and $\text{MTL}_{0,\infty}$ properties over (symbolic) timed transition systems, as well as validity/satisfiability checking of LTL and $\text{MTL}_{0,\infty}$ formulas. This is done via a correct and complete reduction to verification problems in the discrete-time case (thus allowing for the use of mature and efficient verification engines). In order to represent and find infinite traces where clocks may diverge, we extended the representation for lasso-shape traces (over discrete semantics) and we modified the bounded model checking algorithm to properly encode timed traces. We remark that, NUXMV is more expressive than timed automata, since the native management of time is added on top of an infinite state transition system. This makes it straightforward to encode stopwatches and comparison between clocks. We carried out an experimental evaluation comparing NUXMV with other state-of-the-art verification tools for timed automata, considering different benchmarks taken from competitor tools distributions.

2 Software Architecture

The high level architecture of NUXMV is depicted in Fig. 1. For symbolic transition systems NUXMV behaves like the previous version of the system [1], thus allowing for full backward compatibility (apart from some new reserved keywords). It provides the user with all the basic model checking algorithms for finite domains both using BDDs (using CUDD [5]) and SAT (e.g. MINISAT [6]). It supports various SMT-based model checking algorithms (implemented through a tight integration with the MATHSAT5 solver [2]) for the analysis of finite and infinite state systems (e.g. IC3 [7–9], k-liveness [10],

liveness to safety [11]). We refer the reader to [1] for a thorough discussion of these consolidated functionalities for the discrete-time setting.

```
1  @TIME_DOMAIN continuous  -- annotation to specify the time semantics, in this case dense time
2
3  MODULE main
4   FROZENVAR p: real; INIT p > 0  -- parameter
5   VAR i: real;  -- input of the sensor
6   VAR s: Sensor(i);
7   VAR m: Monitor(s.o,p);
8
9   LTLSPEC G ( s.fault -> F [0,p] m.alarm )  -- any fault is detected in p timed units
10
11 MODULE Sensor(i)
12  VAR o: real;
13  VAR fault: boolean;
14  TRANS !fault -> next(o) = i  -- if not faulty, the sensor provides in output directly the input
15  TRANS fault -> next(o) = o  -- if faulty, the sensor output is stuck at the last value
16  TRANS fault -> next(fault)  -- the fault is permanent
17
18 MODULE Monitor (i,p)
19  VAR previous_value: real;
20  VAR c: clock;
21  VAR alarm: boolean;
22  INIT c=0 & previous_value = i & !alarm
23  INVAR TRUE -> c <= p
24  TRANS time <= p | time >= p
25  TRANS (c = p & next(c) = 0 & next(previous_value) = i) |  -- the monitor reads the sensor every p time units
26        (c <= p & next(c) = c & next(previous_value) = previous_value)
27  TRANS next(alarm) <-> (alarm | i=previous_value)  -- alarm raised when the same value read twice consecutively
```

Fig. 2. A simple TIMED-NUXMV program.

To support the specification and model checking of invariant, LTL and $\mathrm{MTL}_{0,\infty}$ properties for timed transitions systems, and for the validity checking of properties over dense time semantics, NUXMV has been extended w.r.t. [1] as discussed here after.

- We extended the parser to allow the user to choose the time semantics to use for the read model. Depending on the time model some parse constructs and checks are enabled and/or disabled. For instance, variables of type clock and $\mathrm{MTL}_{0,\infty}$ properties are only allowed if the dense time semantics has been specified. By default the system uses the discrete time semantics of the original NUXMV. Notice also that, depending on the specified semantics, the commands available to the user change to allow only the analyses supported for the chosen semantics.
- We extended the parser to support the specification of symbolic timed automata (definition of clock variables, specification of urgent transitions and state invariants, etc.). Moreover, we extended the parser to allow for the specification of $\mathrm{MTL}_{0,\infty}$ properties, and we extended the LTL bounded operators not only to contain constants, but also complex expressions over clock variables. See Fig. 2 for a simple example showing some of the new language constructs.
- We extended the symbol table to support the specification of clock variables, and we extended the type checker to properly handle the new defined variables, expression types and language constructs.
- We added new modules for the encoding of the symbolic timed automata into equivalent transition systems to verify with the existing algorithms of NUXMV.
- We extended the traces for NUXMV to support timed traces (lasso-shaped traces where some clock variables may diverge).

– We modified the encoding for the loops in the bounded model checking algorithms to take into account that traces may contain diverging variables to allow for the verification and validation of LTL and $\text{MTL}_{0,\infty}$ properties.

For portability, NUXMV has been developed mainly in standard C with some new parts in standard C++. It compiles and executes on Linux, MS Windows, and MacOS.

3 Language Extensions

Timed Transition Systems. Discrete-time transition systems are described in NUXMV by a set V of variables, an initial condition $I(V)$, a transition condition $T(V, V')$ and an invariant condition $Z(V)$. Variables are introduced with the keyword `VAR` and can have type Boolean, scalar, integer, real or array. The initial and the invariant conditions are introduced with the keyword `INIT` and `INVAR` and are expressions over the variables in V. The transition condition is introduced with `TRANS` and is an expression over variables in V and V', where for each variable v in V, V' contains the "next" version denoted in the language by `next`(v). Expressions may use standard symbols in the theory associated to the variable types and user-defined rigid functions that are declared with the keyword `FUN`.

The input language of NUXMV has been extended to allow the specification of timed transition systems (TTS), which are enabled by the annotation `@TIME_DOMAIN continuous` at the beginning of a model description.

Besides the standard types, in the timed case, state variables can be declared of type `clock`. All variables of type different from `clock` are discrete variables.

The language provides a built-in `clock` variable, accessible through the reserved keyword `time`. It represents the amount of time elapsed from the initial state until now. `time` is initialized to 0 and its value does not change in discrete transitions. While all other `clock` variables can be used in any expression in the model definition, `time` can be used only in comparison with constants.

Initial, transition, and invariant conditions are specified in NUXMV with the keywords `INIT`, `TRANS`, and `INVAR`, as in the discrete case. In particular, `TRANS` allows to specify "arbitrary" clock resets. Like all other NUXMV state variables, if a clock is not constrained during a discrete transition, its next value is chosen non-deterministically.

Clock variables can be used in `INVAR` only in the form $\varphi \rightarrow \phi$, where φ is a formula built using only the discrete variables and ϕ is convex over the clock variables. This closely maps the concept of location invariant described for timed automata: all locations satisfying φ have invariant ϕ.

An additional constraint, not allowed in the discrete-time case, is introduced with the keyword `URGENT` followed by a predicate over the discrete variables, which allows to specify a set of locations in which time cannot elapse.

Comparison with Timed Automata. Timed automata can be represented by TTSs by simply introducing a variable representing the locations of the automaton. Note that, in TTS, it is possible to express any kind of constraint over clock variables in discrete transitions, while in timed automata it is only possible to reset them to 0 in transitions or compare them to constants in guards. Moreover, the discrete variables of a

timed automaton always have finite domain, while in TTSs, also the discrete variables might have an infinite domain. This additional expressiveness allows to describe more complex behaviors (e.g. it is straightforward to encode stopwatches and comparison between clocks) losing the decidability of the model checking problem.

Specifications. NUXMV's support for LTL has been extended to allow for the use of $\text{MTL}_{0,\infty}$ operators [12] and other operators such as event-freezing functions [13] and dense version of LTL `X` and `Y` operators. $\text{MTL}_{0,\infty}$ bounded operators extend the LTL ones of NUXMV to allow for bounds either of the form $[c,\infty)$, where c is a constant greater or equal to 0, e.g. `F[0,+oo)` φ, or generic expressions over parametric/frozen variables: e.g. `F [0, 3+v]` φ where `v` is a frozen variable.

In timed setting, next and previous operators come in two possible versions. The standard LTL operators `X` and `Y` require to hold, respectively after and before, a discrete transition. Dually, `X~` and `Y~` have been introduced to allow to predicate about the evolution over time of the system. They are always `FALSE` in discrete steps and hold in time elapses if the argument holds in the open interval immediately after/before (resp.) the current step. The disjunction $\texttt{X}(\varphi) \vee \texttt{X~}(\varphi)$ allows to check if the argument φ holds after the current state without distinction between time or discrete evolution.

The event-freezing operators *at next* and *at last*, written `@F~` and `@O~`, are binary operators allowed in LTL specifications. The left-hand side is a term, while the right-hand side is a temporal formula. They return the value of the term respectively at the next and at the last point in time in which the formula is true. If the formula will [has] never happen [happened] the operator evaluates to a default value.

`time_until` and `time_since` are two additional unary operators that can be used in LTL specifications of timed models. Their argument must be a Boolean predicate over current and next variables. $\texttt{time\_until}(\varphi)$ evaluates to the amount of time elapse required to reach the next state in which φ holds, while $\texttt{time\_since}(\varphi)$ evaluates to the amount of time elapsed from the last state in which φ held. As for the `@F~` and `@O~` operators if no such state exists they are assigned to a default value.

4 Extending Traces

Timed Traces. The semantics of NUXMV has been extended to take into account the timing aspects in case of super-dense time. While in the discrete time case, the execution trace is given by a sequence of states connected by discrete transitions (i.e., satisfying the transition condition), in the super-dense time case the execution trace is such that every pair of consecutive states is a discrete or a timed transition. As in the discrete case, discrete transitions are pair of states satisfying the transition condition. As in timed automata, in a timed transition time elapses for a certain amount (referred to as `delta_time`), clocks increase of the same amount, while discrete variables do not change.

Lasso-Shaped Traces with Diverging Variables. Traditionally, the only infinite paths supported by NUXMV have been those in lasso shape, i.e. those traces which can be

represented by a finite prefix $s_0, s_1, \ldots, s_l$ (called the stem) followed by a finite suffix $s_{l+1}, \ldots, s_k \equiv s_l$ (called the loop), which can be repeated infinitely many times. While this representation is sufficient for finite-state systems (because in a finite-state setting if a system does not satisfy an LTL property, then a lasso-shaped counter-example trace is guaranteed to exist), this is an important limitation in an infinite-state context, in which lasso-shaped counter-examples are not guaranteed to exist. (As a simple example, consider a system $M := \langle\{x\}, (x = 0), (x' = x + 1)\rangle$ in which $x \in \mathbb{Z}$. Then $M \not\models \mathbf{GF}(x = 0)$, but clearly M has no lasso-shaped trace). In fact, this is especially relevant for timed transition systems, which, by the presence of the always-diverging variable $time$, admit *no* lasso-shaped trace.

In order to overcome this limitation, we introduce new kinds of infinite traces, which we call *lasso-shape traces with diverging variables* (to allow also for representing traces with variables whose value might be diverging). We modified the bounded model checking algorithms to leverage on this new representation to then extend the capabilities to find witnesses for a given property. This representation significantly extends the capabilities of NUXMV to find witnesses for violated LTL and MTL properties on timed transition systems (see experimental evaluation).

Definition 1. *Let $\pi := s_0, s_1, \ldots, s_l, \ldots$ be an infinite trace of a system M over variables V. We say that π is a* lasso-shaped trace with diverging variables *iff there exist indexes $0 \leq l \leq k$, a partitioning of V into sets X and Y ($V = X \uplus Y$) and an expression $f_y(V)$ over V for every variable $y \in Y$ such that, for every $i > k$,*

$$s_i(v) := \begin{cases} s_{l+((i-l) \bmod (k-l))}(v) & \text{if } v \in X \text{ (like in lasso-shaped traces);} \\ f_v(s_{i-1}) & \text{if } v \in Y \text{ (as function of previous state).} \end{cases}$$

Intuitively, the idea of lasso-shaped traces with diverging variables is to provide a finite representation for infinite traces that is more general then simple lasso-shaped ones, and which allows to capture more interesting behaviors of timed transition systems.

Example 1. Consider the system $M := \langle\{y, b\}, \neg b \wedge y = 0, (b' = \neg b) \wedge (b \rightarrow y' = y{+}1) \wedge (\neg b \rightarrow y' = y)\rangle$. Then one lasso-shaped trace for M is given by: $\pi := s_0, s_1, s_2$, where $s_0 := \{b \mapsto \bot, y \mapsto 0\}$, $s_1 := \{b \mapsto \top, y \mapsto 0\}$, and $s_2 := \{b \mapsto \bot, y \mapsto 1\}$; the trace is lasso-shaped with diverging variables considering $Y := \{y\}$; the loop-back at index 0, and $f_y(b, y) := b \; ? \; y + 1 \; : \; y$.

Extended BMC for Traces with Divergent Clocks. The definition above requires the existence of the functions f_y for computing the updates of diverging variables. In case y is a clock variable, we can define a region $[\![\phi_y]\!]$ in which y can diverge (i.e., $f_y = y{+}\delta$, where δ is the delta time variable).

In order to capture lasso-shaped traces with diverging variables, we can modify the BMC encoding as follows. Let $\bigvee_{l=0}^{k}(\bigwedge_{v \in X \uplus Y}(v^l = v^k) \wedge {}_l[\![\varphi]\!]_k^0)$ be the formula

representing the BMC encoding of [14] at depth k with all possible loop-backs $0 \leq l \leq k$ for a given formula φ. The encoding is extended as follows:

$$\bigvee_{l=0}^{k} \left(\left(\bigwedge_{x \in X} (x^l = x^k) \wedge \bigwedge_{y \in Y} (y^l = y^k \vee \bigwedge_{i=l}^{k} [\![\phi_y]\!]_i) \right) \wedge {}_l[\![\varphi]\!]_k^0 \right).$$

The correctness of the encoding relies on a safe choice of the set Y, falling back to the incomplete lasso-shaped case when some syntactic restrictions on the expressions containing clocks are not met (see appendix for more details).

5 Related Work

There are many tools that allow for the specification and verification of infinite state symbolic synchronous transition systems. Given the focus of this paper, here we restrict our attention to tools supporting timed systems and/or MTL properties.

Uppaal [15], the reference tool for timed systems verification, supports only bounded variable types and therefore finite asynchronous TTS. Properties are limited to a subset of the branching-time logic TCTL [16,17]. LTSmin [18] and Divine [19] are two model checkers that support the Uppaal specification language and properties specified in LTL. RTD-Finder [20] handles only safety properties for real-time component-based systems specified in RT-BIP. The verification is based on a compositional computation of an invariant over-approximating the set of reachable states of the system and leverages on counterexample-based invariant refinement algorithm. The ZOT Bounded Model/Satisfiability Checker [21] supports different logic languages through a multi-layered approach based on LTL with past operators. Similarly to NUXMV, ZOT supports dense-time MTL. It leverages only on SMT-based Bounded Model Checking, and is therefore unable to prove that properties hold. Atmoc [22] implements an extension of IC3 [7] and K-induction [23] to deal with symbolic timed transition systems. It supports both invariant and $\text{MTL}_{0,\infty}$ properties, although for the latter it only supports bounded model checking. CTAV [24] reduces the model checking problem for an $\text{MTL}_{0,\infty}$ property φ to a symbolic language emptiness check of a timed Büchi automata for φ.

Differently from all the above tools NUXMV is able to prove $\text{MTL}_{0,\infty}$ properties on timed transition systems with infinite domain variables.

6 Experimental Evaluation

We compared NUXMV with Atmoc [22], CTAV [24], ZOT [21], Divine [19], LTSmin [18], and Uppaal [25].

For the evaluation we considered (i) scalable benchmarks taken from competitor tools distributions and from the literature; (ii) handcrafted benchmarks to stress various language features. In particular, we considered different versions of the Fisher mutual exclusion protocol (correct and buggy) with different properties, different versions of the emergency diesel generator problem (previously studied with Atmoc [22]). Finally we considered also the validity checks of some MTL properties also taken from [22].

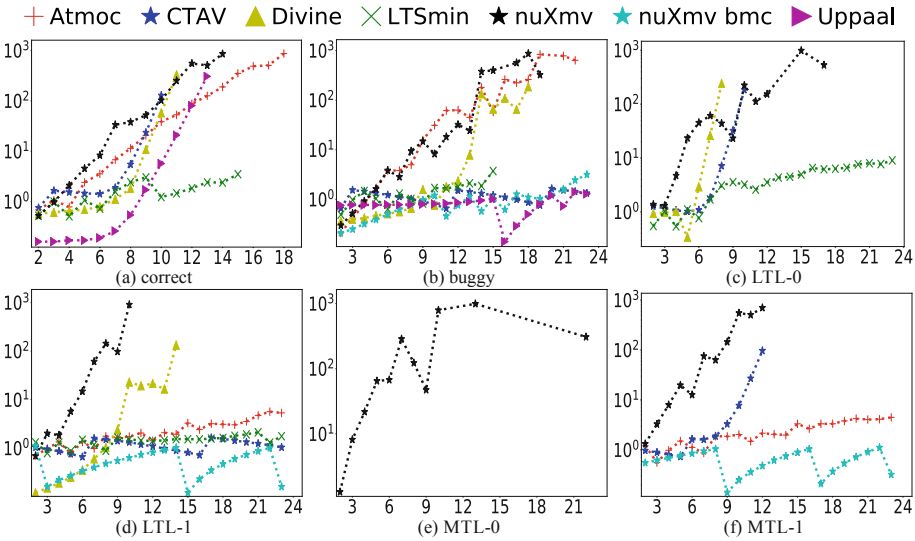

Fig. 3. Runtime for the Fisher mutual exclusion problem; x-axis: number of processes, y-axis: time (s). LTL-1 and MTL-1 properties are the bounded version of resp. LTL-0 and MTL-0.

We run all the experiments on a PC equipped with a 3.7 GHz Xeon quad core CPU and 16 Gb of RAM, using a time/memory limit of 1000 s/10 Gb for each test. We refer the reader to [26] to retrieve all the data to reproduce this experimental evaluation.

The results of the evaluation are reported in Fig. 3 for the Fisher family of experiments, and in Fig. 4 for the emergency diesel generator family of problems (CTAV does not appear in the plot of MTL-0 because it wrongly reports a counterexample although MTL-0 is the bounded version of LTL-0). While the results for the validity check of pure MTL properties are reported in Fig. 5. In the plots NUXMV refers to runtime for the IC3 with implicit abstraction in lockstep with BMC with the modified loop condition algorithm, and NUXMV-bmc refers to runtime for BMC alone with the modified loop condition algorithm. The results show that NUXMV is competitive with and in many cases performs better than other state-of-the-art tools, especially on validity problems for $\text{MTL}_{0,\infty}$.

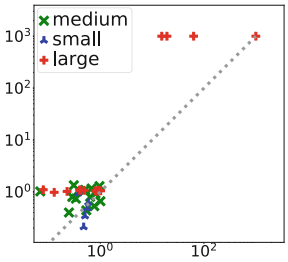

Fig. 4. Result for the runtime (s) for the emergency diesel generator family of problems: NUXMV (x) vs Atmoc (y).

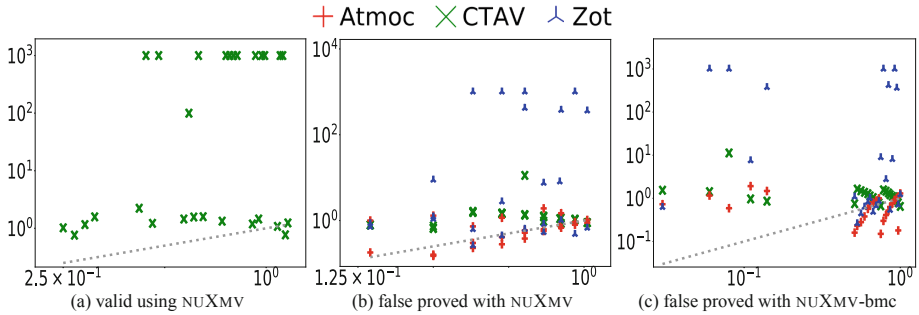

Fig. 5. Runtime (s) for the validity checks of MTL properties.

7 Conclusions

We presented the new version of NuXmv, a state-of-the art symbolic model checker for finite and infinite-state transition systems, that we extended to allow for the specification of synchronous timed transition systems and of $\text{MTL}_{0,\infty}$ properties. To support the new features, we extended the NuXmv language, we allowed for the specification $\text{MTL}_{0,\infty}$ formulas with parametric intervals, we adapted the model checking algorithms to find for lasso-shaped traces (over discrete semantics) where clocks may diverge. We evaluated the new features comparing NuXmv with other verification tools for timed automata, considering different benchmarks. The results show that NuXmv is competitive with and in many cases performs better than state-of-the-art tools, especially on validity problems for $\text{MTL}_{0,\infty}$.

References

1. Cavada, R., et al.: The NuXmv symbolic model checker. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 334–342. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_22
2. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
3. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. **2**(4), 255–299 (1990)
4. Ouaknine, J., Worrell, J.: On the decidability of metric temporal logic. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science. LICS 2005, pp. 188–197. IEEE (2005)
5. Somenzi, F.: CUDD: Colorado University Decision Diagram package – release 2.4.1
6. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
7. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7

8. Hassan, Z., Bradley, A.R., Somenzi, F.: Better generalization in IC3. In: FMCAD, pp. 157–164. IEEE (2013)
9. Vizel, Y., Grumberg, O., Shoham, S.: Lazy abstraction and sat-based reachability in hardware model checking. In: Cabodi, G., Singh, S. (eds.) FMCAD, pp. 173–181. IEEE (2012)
10. Claessen, K., Sörensson, N.: A liveness checking algorithm that counts. In: Cabodi, G., Singh, S. (eds.) FMCAD, pp. 52–59. IEEE (2012)
11. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006)
12. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. J. ACM **43**(1), 116–146 (1996)
13. Tonetta, S.: Linear-time Temporal Logic with Event Freezing Functions. In: GandALF, pp. 195–209 (2017)
14. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
15. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
16. Bouyer, P.: Model-checking timed temporal logics. In: Areces, C., Demri, S. (eds.) Proceedings of the 4th Workshop on Methods for Modalities (M4M–5). Electronic Notes in Theoretical Computer Science, vol. 1, pp. 323–341. Elsevier Science Publishers, Cachan, March 2009
17. Bouyer, P., Laroussinie, F., Markey, N., Ouaknine, J., Worrell, J.: Timed temporal logics. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 211–230. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_11
18. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 692–707. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_61
19. Baranová, Z., et al.: Model checking of C and C++ with DIVINE 4. In: D'Souza, D., Narayan Kumar, K. (eds.) ATVA 2017. LNCS, vol. 10482, pp. 201–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68167-2_14
20. Ben-Rayana, S., Bozga, M., Bensalem, S., Combaz, J.: RTD-finder: a tool for compositional verification of real-time component-based systems. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 394–406. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_23
21. Pradella, M.: A user's guide to zot. CoRR abs/0912.5014 (2009)
22. Kindermann, R., Junttila, T.A., Niemelä, I.: Smt-based induction methods for timed systems. CoRR abs/1204.5639 (2012)
23. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) FMCAD 2000. LNCS, vol. 1954, pp. 127–144. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-40922-X_8
24. Li, G.: Checking timed büchi automata emptiness using LU-abstractions. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 228–242. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_18
25. Larsen, K.G., Lorber, F., Nielsen, B.: 20 years of UPPAAL enabled industrial model-based validation and beyond. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11247, pp. 212–229. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03427-6_18
26. Cimatti, A., Griggio, A., Magnago, E., Roveri, M., Tonetta, S.: Extending nuXmv with timed transition systems and timed temporal properties (extended version) (2019). Extended version with data to reproduce experiments https://nuxmv.fbk.eu/papers/cav2019