The VMT-LIB Language and Tools

Alessandro Cimatti, Alberto Griggio and Stefano Tonetta

Fondazione Bruno Kessler, Trento, Italy

Abstract

We present VMT-LIB, a language for the representation of verification problems of invariant and linear-time temporal properties on infinite-state symbolic transition systems. VMT-LIB is developed with the goal of facilitating the interoperability and exchange of benchmark problems among different verification tools. The VMT-LIB language an extension of the standard SMT-LIB language for SMT solvers, from which it inherits the clean semantics and the many available resources. In this paper we describe the syntax and semantics of VMT-LIB, and present a set of open-source tools to work with the language.

Keywords

SMT-LIB, Model checking, Benchmarking

1. Introduction

We call Verification Modulo Theories the problem of verifying invariant and linear-time temporal properties on infinite-state symbolic transition systems described with first-order formulas. The VMT problem has received a lot of attention in recent years, also thanks to the enormous growth and success of Satisfiability Modulo Theories. In fact, the availability of strong SMT solvers paved the way to a number of approaches, e.g. Bounded Model Checking and k-induction for infinite-state systems [1, 2, 3], model checking using first-order interpolation [4, 5, 6], predicate abstraction and refinement [7, 8, 9], implicit predicate abstraction [10], various extensions of IC3 [11, 12, 13, 14, 15, 16], and approaches focused on Constrained Horn Clauses [17] or on parametrized systems [18]. In turn, these VMT approaches led the way to the solution of more extended problems, for example considering continuous time dynamic [19, 20, 21, 22] or temporal logic satisfiability modulo theory [23, 24, 25].

It is increasingly important to have a common language that allows the sharing of benchmarks and the comparison between verification tools. In this paper we describe VMT-LIB, a language for the representation of verification problems of invariant and linear-time temporal properties on infinite-state symbolic transition systems. VMT-LIB was designed with the main goals of having a clear semantics and being simple to use (i.e. to parse and generate) for verification tools, with the aim of facilitating the

SMT 2022: Satisfiability Modulo Theories, August 11–12, 2022, Haifa, Israel

cimatti@fbk.eu (A. Cimatti); griggio@fbk.eu (A. Griggio); tonettas@fbk.eu (S. Tonetta)

0000-0002-1315-6990 (A. Cimatti); 0000-0002-3311-0893 (A. Griggio); 0000-0001-9091-7899 (S. Tonetta)

CEUR Workshop Proceedings (CEUR-WS.org)

interoperability among different tools and the collection of verification benchmarks for infinite-state systems.

VMT-LIB was developed as an extension of the standard SMT-LIB [26] language for SMT solvers, by exploiting the capability of SMT-LIB of attaching metadata to terms and formulas via annotations. In particular, a valid VMT-LIB file is also a valid SMT-LIB file, and the elements of the problem (e.g. the initial condition, the transition relation) are represented as SMT-LIB formulae. By building on top of SMT-LIB, we inherit a clean and general formal framework, together with a number of theories of interest. Furthermore, this choice allows us to reuse all the libraries for manipulating SMT-LIB formulas that are available for various languages (e.g. [27, 28, 29]). Besides these generic libraries, we have also developed a set of tools to work with the language, including converters to and from other formats and formalisms (including Aiger, BTOR and Constrained Horn Clauses). All the tools are open source and available at the VMT-LIB webpage [30], together with a collection of benchmarks for various theories. VMT-LIB is supported by several tools, e.g. nuXmv [31], Euforia [16], AVR [32], and it has been used as a benchmark format in several publications over the last few years (e.g. [33, 34, 32, 35, 36]).

Related work. Compared to other similar approaches, VMT finds its main advantage in being built on top of SMT-LIB, offering to specify the formulas describing the verification problem directly in SMT.

VMT-LIB is similar in spirit to the Aiger [37] language for finite-state systems, and to the BTOR [38] language for word-level systems with arrays. Both were created with the same purpose, i.e. simple language, easy to parse and generate, a precise and simple semantics. However, BTOR is based on a generalization of Aiger, focused on word-level representation of finite-state transition systems. While BTOR appears to be less suitable to be extended to general support for first-order theories, VMT-LIB supports arbitrary background SMT theories, including e.g. linear and nonlinear arithmetic, uninterpreted functions, and quantifiers.

Compared to the SMV language[1] or the intermediate language for model checking of [39], currently under development, VMT-LIB is a lower-level language, focusing on simplicity and interoperability of tools, and hence sacrificing readability and the richer structure of the higher-level languages. For exammple, in contrast to SMV, which features reusable modules, VMT-LIB only considers flat transition systems. On the positive side, while in the SMV language the theories are "hardcoded", in VMT-LIB it is possible to directly reuse the theories available in SMT-LIB (and hence its future extensions).

Also the Numerical Transition Systems (NTS) format and the related NTS-LIB library [40, 41] are quite related to VMT-LIB. An NTS is an extended state machine representing a control-flow graph, where edges are annotated with arithmetic formulas. As for SMV, compared to VMT-LIB, the first-order theories are limited and hardcoded, while the NTS-LIB format provides a native support for the control-flow graph structure (which must be instead encoded in VMT-LIB).

[1] Or, more properly, the extension of the SMV language processed by the nuXmv model checker.

Constrained Horn Clauses have also been used to model verification modulo theory problems, with a direct encoding in quantified first-order logic modulo theories. In contrast, VMT-LIB allows to separately represent the model and possibly several properties.

Structure of the paper. The rest of the report is structured as follows. After providing the necessary theoretical background in §2, we describe the core VMT-LIB syntax in §3, and its semantics in §4. In §5 we describe an extension to deal with LTL properties. In §6 we describe a set of open-source tools that we have developed to work with the language. Finally, we conclude in §7.

2. Theoretical Background and Definitions

Our setting is many-sorted first order logic. We use the standard notions of theory, satisfiability, validity, and logical consequence. We refer to the SMT-LIB specifications [26] for more details. We denote generic theories as $\mathcal{T}$. We write $\varphi \models_{\mathcal{T}} \psi$ to denote that the formula ψ is a logical consequence of φ in the theory $\mathcal{T}$; when clear from context, we omit $\mathcal{T}$ and simply write $\varphi \models \psi$.

We refer to 0-arity predicates as Boolean variables, and to 0-arity uninterpreted functions as (theory) variables.

Given a set of variables X, a signature Σ, a domain M, an interpretation function $\mathcal{I}$ of the symbols in Σ on the domain M, an assignment σ to the variables in X on the domain M, and a Σ-formula $\phi(X)$ with free variables in X, the satisfaction relation $\langle M, \mathcal{I} \rangle \models \phi$ is defined in the usual way.

For each variable x, we assume that there exists a corresponding variable x', called the *primed version* of x. If X is a set of variables, X' is the set obtained by replacing each element x with its primed version ($X' = \{x' \mid x \in X\}$). φ' is the formula obtained by replacing each occurrence variable in φ with the corresponding primed.

In the following, the signature Σ and the theory $\mathcal{T}$ are implicitly given. A *transition system (TS)* S is a tuple $\langle X, I(X), T(Y, X, X') \rangle$ where X is a set of *state* variables, $I(X)$ is a formula representing the initial states, and $T(Y, X, X')$ is a formula representing the transitions, where Y is a set of *input* variables.

3. Syntax

VMT-LIB exploits the capability offered by the SMT-LIB language of attaching metadata to terms and formulas in order to specify the components of the transition system and the properties to verify. More specifically, we use the following SMT-LIB *annotations*:

`:next name` is used to represent state variables. For each variable x in the model, the VMT-LIB file contains a pair of variables, x^c and x^n, representing respectively the current and next version of x. The two variables are linked by annotating x^c with the attribute `:next` x^n. All the variables that are not in relation with another by

means of a `:next` attribute are considered inputs. Note that `:next` must define an injective function (i.e. it is an error if there are two variables with the same `:next` value), and that the names of the variables are not important.

`:init` is used to specify the formula for the initial states of the model. This formula should contain neither next-state variables nor input variables. Multiple formulas annotated with `:init` are implicitly conjoined. As a convenience, the annotation can also use a "dummy" value `true`.

`:trans` is used to specify the formula for the transition relation. As in the case for `:init`, multiple `:trans` formulas are conjoined together, and also in this case the annotation can be written as `:trans true`.

`:invar-property idx` is used to specify invariant properties, i.e. formulas of the form $\mathbf{G}p$, where p is the formula annotated with `:invar-property`. The non-negative integer *idx* is a unique identifier for the property.

`:live-property idx` is used to specify an LTL property of the form $\mathbf{FG}p$, where p is the formula annotated with `:live-property`. The non-negative integer *idx* is a unique identifier for the property.

In a VMT-LIB file, only annotated terms and their sub-terms are meaningful. Any other term is ignored. Moreover, only the following commands are allowed to occur in VMT-LIB files: `set-logic`, `set-option`, `declare-sort`, `define-sort`, `declare-fun`, `define-fun`.(For convenience, an additional `(assert true)` command is allowed to appear at the end of the file.)
The following example shows a simple model in the syntax of nuXmv [31] on the left, and its corresponding VMT-LIB translation on the right.

nuXmv	VMT
MODULE main -- declaring the state -- variable x VAR x : integer; IVAR b : boolean; INIT x = 1; TRANS next(x) = b ? x + 1 : x; INVARSPEC x > 0; LTLSPEC FG x > 10;	; declaring the state variable x (declare-const x Int) (declare-const x.next Int) (define-fun sv.x () Int (! x :next x.next)) (declare-const b Bool) (define-fun init () Bool (! (= x 1) :init)) (define-fun trans () Bool (! (= x.next (ite b (+ x 1) x)) :trans)) (define-fun p1 () Bool (! (> x 0) :invar-property 1)) (define-fun p2 () Bool (! (> x 10) :live-property 2))

Since the SMT-LIB format (and thus also the VMT-LIB one that inherits from SMT-LIB) does not allow to annotate the declaration of variables, it is a good practice to insert immediately after the declaration of the variables a set of defines to specify the

relations among variables. See for instance the define `sv.x` in the example above that introduces the relation between `x` and `x.next`.

4. Semantics

In this section we provide the semantics for the language.

States and Paths

Given a transition system $S \doteq \langle X, I(X), T(Y, X, X')\rangle$ over a background theory T with a signature Σ and an interpretation $\mathcal{I}$, a *state* s of S is an interpretation of the state variables X. A (finite) *path* of S is a finite sequence $\pi \doteq s_0, s_1, \ldots, s_k$ of states, with the same domain and interpretation of symbols in the signature Σ, such that $\mathcal{I}, s_0 \models I(X)$ and for all i, $0 \leq i < k$, $\mathcal{I}, s_i, s'_{i+1} \models \exists Y.T(Y, X, X')$. We say that a state s is reachable in S iff there exists a path of S ending in s. Note that, since the interpretation $\mathcal{I}$ is unique, uninterpreted function and predicate symbols are *rigid*, i.e. they are not allowed to change across states.

Invariant Properties

An *invariant property* p is a symbolic representation of a set of states that must be a superset of the reachable states of S. In other words, $S \models p$ iff $\forall s.s$ is reachable in S, $s \models p$. Consequently, a *counterexample* for p is a *finite path* $s_0, \ldots, s_k$ of S such that $s_k \models \neg p$.

Live Properties

A *live property* p represents a set of states that is *eventually invariant*. In LTL syntax, it would be denoted with $\mathbf{FG}p$. More formally, $S \models p$ iff for all paths $s_0, \ldots, s_i, \ldots$, $\exists i.\forall j > i.s_j \models p$. (Note that finite paths $s_0, \ldots, s_k$ vacuously satisfy a live property, because we can always take $i = k$ to satisfy the previous definition.) Consequently, a *counterexample* for p is an *infinite path* $s_0, \ldots, s_i, \ldots$ of S such that $\forall i.\exists j > i.s_j \models \neg p$.

5. LTL Properties, Invariant Constraints and Fairness Conditions

Since one of the main goals of VMT-LIB is that of simplicity, the language as presented above, which we call the *core VMT-LIB language*, does not provide any direct support for high-level constructs such as specifications written in full LTL, invariant constraints or fairness conditions. However, this is not a limitation in terms of expressiveness, as all such constructs can be easily encoded in VMT-LIB:

LTL properties can be compiled into invariant and/or live properties using standard algorithms from the literature (e.g. [42, 43, 44]);

invariant constraints can be straightforwardly embedded into init and trans formulas;

fairness conditions can be embedded into live properties using a symbolic version of standard degeneralization procedures for Büchi automata (e.g. [45]).

An alternative to the above encodings is that of specifying some extensions to the core language, by defining specific annotations to represent LTL properties, invariant constraints and fairness conditions. In particular, we define the following extensions to the core annotations:

`:invar` to specify explicit invariant constraints;

`:live-property idx` can be used to specify more general liveness properties with fairness conditions in the form $\mathbf{FG}\phi_1 \vee \ldots \vee \mathbf{FG}\phi_n$ (or, equivalently, $(\mathbf{GF}\neg\phi_1 \wedge \ldots \wedge \mathbf{GF}\neg\phi_{n-1}) \rightarrow \mathbf{FG}\phi_n$) by annotating $\phi_1, \ldots, \phi_n$ with the same index `idx`;

`:ltl-property idx` can be used to specify LTL properties. In order to avoid the introduction of special syntax for the temporal operators, we model them by predefining a set of functions from Booleans to Booleans of the form `ltl.OP`, where `OP` is one of the standard temporal operators (i.e. **F**, **G**, **U**, or **X**). For example, the formula

$$\mathbf{G}(\phi \rightarrow \mathbf{F}(\psi \mathbf{U} \xi))$$

can be written as

```
(! (ltl.G (=> φ (ltl.F (ltl.U ψ ξ)))) :ltl-property 1).
```

We refer to the augmented set of annotations as the *extended VMT-LIB language*. From the semantic point of view, these extensions are considered simply as "syntactic sugar". This keeps the definitions of §4 simple and concise. From the practical point of view, however, extensions might still be useful to provide higher-level information to solvers which might be able to exploit it (e.g. for solvers with built-in support for LTL specifications it might be more efficient to their native encoding of LTL rather than rewriting the property upfront). Such considerations are however beyond the scope of the present document.

6. VMT-LIB Tools

VMT-LIB support in verification tools. The VMT-LIB language is fully supported by nuXmv [31], a state-of-the-art symbolic model checker for finite- and infinite-state systems. Recently, the language has been adopted also by the AVR [32] model checker. VMT-LIB is also the native language of ic3ia [46], an efficient open-source model checker for invariant and LTL properties, as well as its recent extensions ProphIC3 [34] (for discovering universally quantified invariants over arrays) and Lambda [35] (for the verification of parametric systems).

Tools for working with VMT-LIB. We provide a set of tools (mainly written in the Python programming language) to work with the VMT-LIB language. They are all available from the VMT-LIB webpage [30]. Currently, the following tools are provided:

vmt.py: parsing and printing of transition systems in VMT-LIB.

vmt2btor.py: converter from VMT-LIB to the BTOR format.

btor2vmt.py: converter from BTOR to VMT-LIB.

vmt2horn.py: converter from VMT-LIB to Constrained Horn Clauses.

vmt2nuxmv.py: converter from VMT-LIB to the SMV dialect of nuXmv.

vmtext2core.py: a tool to convert an extended VMT-LIB file to a core VMT-LIB file, by performing the rewritings described at the end of §3.

Moreover, further converters to VMT-LIB are available through nuXmv and ic3ia. In particular, ic3ia provides a `horn2vmt` tool for converting Constrained Horn Clauses to VMT-LIB, whereas nuXmv can be used to convert from VMT-LIB to Aiger and vice versa.

7. Conclusions and Future Work

We have presented VMT-LIB, a language and a set of tools for the specification of verification problems over infinite-state transition systems. VMT-LIB is aimed at simplicity and interoperability, and is built on top of SMT-LIB, from which it inherits the formal foundations, the syntax and the semantics. It has been adopted by several verification engines, and comes with a library of benchmark and a set of scripts for conversion between several other formats.

Differently from the SMT-LIB language, VMT-LIB does not support instructions to the solvers. A python-based programmatic framework, called pyVMT [47], in the style of pySMT [27], is being developed to overcome this limitation. In the future, we plan to extend the format to support the representation of counterexample traces for violated properties (which is a non-trivial task in the case of infinite-state systems), and possibly also proof certificates for verified properties.

References

[1] G. Audemard, A. Cimatti, A. Kornilowicz, R. Sebastiani, Bounded model checking for timed systems, in: FORTE, volume 2529 of *Lecture Notes in Computer Science*, Springer, 2002, pp. 243–259.

[2] T. Kahsai, C. Tinelli, Pkind: A parallel k-induction based model checker, in: PDMC, volume 72 of *EPTCS*, 2011, pp. 55–62.

[3] A. Champion, A. Mebsout, C. Sticksel, C. Tinelli, The kind 2 model checker, in: CAV (2), volume 9780 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 510–517.
[4] K. L. McMillan, Applications of craig interpolation to model checking, in: ICATPN, volume 3536 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 15–16.
[5] K. L. McMillan, Lazy abstraction with interpolants, in: CAV, volume 4144 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 123–136.
[6] A. Griggio, Effective word-level interpolation for software verification, in: FMCAD, FMCAD Inc., 2011, pp. 28–36.
[7] S. K. Lahiri, R. Nieuwenhuis, A. Oliveras, SMT techniques for fast predicate abstraction, in: CAV, volume 4144 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 424–437.
[8] S. K. Lahiri, T. Ball, B. Cook, Predicate abstraction via symbolic decision procedures, Log. Methods Comput. Sci. 3 (2007).
[9] T. A. Henzinger, R. Jhala, R. Majumdar, K. L. McMillan, Abstractions from proofs, ACM SIGPLAN Notices 49 (2014) 79–91. URL: https://doi.org/10.1145/2641638.2641655. doi:10.1145/2641638.2641655.
[10] S. Tonetta, Abstract model checking without computing the abstraction, in: FM, volume 5850 of *LNCS*, Springer, 2009, pp. 89–105.
[11] A. Cimatti, A. Griggio, S. Mover, S. Tonetta, Infinite-state invariant checking with IC3 and predicate abstraction, Formal Methods Syst. Des. 49 (2016) 190–218.
[12] A. Komuravelli, A. Gurfinkel, S. Chaki, Smt-based model checking for recursive programs, Formal Methods Syst. Des. 48 (2016) 175–205. URL: https://doi.org/10.1007/s10703-016-0249-4. doi:10.1007/s10703-016-0249-4.
[13] K. Hoder, N. S. Bjørner, Generalized property directed reachability, in: SAT, volume 7317 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 157–171.
[14] J. Birgmeier, A. R. Bradley, G. Weissenbacher, Counterexample to induction-guided abstraction-refinement (CTIGAR), in: CAV, volume 8559 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 831–848.
[15] T. Lange, M. R. Neuhäußer, T. Noll, J. Katoen, IC3 software model checking, Int. J. Softw. Tools Technol. Transf. 22 (2020) 135–161. URL: https://doi.org/10.1007/s10009-019-00547-x. doi:10.1007/s10009-019-00547-x.
[16] D. Bueno, K. A. Sakallah, euforia: Complete software model checking with uninterpreted functions, in: VMCAI, volume 11388 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 363–385.
[17] N. S. Bjørner, A. Gurfinkel, K. L. McMillan, A. Rybalchenko, Horn clause solvers for program verification, in: Fields of Logic and Computation II, volume 9300 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 24–51.
[18] S. Ghilardi, E. Nicolini, S. Ranise, D. Zucchelli, Towards SMT model checking of array-based systems, in: IJCAR, volume 5195 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 67–82.
[19] A. Eggers, M. Fränzle, C. Herde, SAT modulo ODE: A direct SAT approach to hybrid systems, in: ATVA, volume 5311 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 171–185.

[20] A. Eggers, N. Ramdani, N. S. Nedialkov, M. Fränzle, Improving the SAT modulo ODE approach to hybrid systems analysis by combining different enclosure methods, Softw. Syst. Model. 14 (2015) 121–148.
[21] A. Cimatti, S. Mover, S. Tonetta, Smt-based scenario verification for hybrid systems, Formal Methods Syst. Des. 42 (2013) 46–66.
[22] S. Mover, A. Cimatti, A. Griggio, A. Irfan, S. Tonetta, Implicit semi-algebraic abstraction for polynomial dynamical systems, in: CAV (1), volume 12759 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 529–551.
[23] S. Ghilardi, E. Nicolini, S. Ranise, D. Zucchelli, Combination methods for satisfiability and model-checking of infinite-state systems, in: CADE, volume 4603 of *Lecture Notes in Computer Science*, Springer, 2007, pp. 362–378.
[24] S. Tonetta, Linear-time temporal logic with event freezing functions, in: GandALF, volume 256 of *EPTCS*, 2017, pp. 195–209.
[25] A. Cimatti, A. Griggio, E. Magnago, M. Roveri, S. Tonetta, Smt-based satisfiability of first-order LTL with event freezing functions and metric operators, Inf. Comput. 272 (2020) 104502.
[26] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB Standard: Version 2.6, Technical Report, 2021. https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf.
[27] M. Gario, A. Micheli, PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms, in: SMT Workshop 2015, 2015.
[28] M. Mann, A. Wilson, Y. Zohar, L. Stuntz, A. Irfan, K. Brown, C. Donovick, A. Guman, C. Tinelli, C. W. Barrett, Smt-switch: A solver-agnostic C++ API for SMT solving, in: SAT, volume 12831 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 377–386.
[29] D. Baier, D. Beyer, K. Friedberger, Javasmt 3: Interacting with SMT solvers in java, in: CAV (2), volume 12760 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 195–208.
[30] VMT-LIB, http://vmt-lib.org, 2022.
[31] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, S. Tonetta, The nuxmv symbolic model checker, in: CAV, volume 8559 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 334–342.
[32] A. Goel, K. A. Sakallah, AVR: abstractly verifying reachability, in: TACAS (1), volume 12078 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 413–422.
[33] A. Cimatti, A. Griggio, S. Mover, S. Tonetta, Infinite-state invariant checking with IC3 and predicate abstraction, Formal Methods Syst. Des. 49 (2016) 190–218. URL: https://doi.org/10.1007/s10703-016-0257-4. doi:10.1007/s10703-016-0257-4.
[34] M. Mann, A. Irfan, A. Griggio, O. Padon, C. W. Barrett, Counterexample-guided prophecy for model checking modulo the theory of arrays, in: TACAS (1), volume 12651 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 113–132.
[35] A. Cimatti, A. Griggio, G. Redondi, Universal invariant checking of parametric systems with quantifier-free SMT reasoning, in: CADE, volume 12699 of *Lecture Notes in Computer Science*, Springer, 2021, pp. 131–147.
[36] A. Cimatti, A. Griggio, A. Irfan, M. Roveri, R. Sebastiani, Incremental Linearization

for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions, ACM Trans. Comput. Log. 19 (2018) 19:1–19:52. URL: https://doi.org/10.1145/3230639. doi:10.1145/3230639.

[37] A. Biere, K. Heljanko, S. Wieringa, AIGER 1.9 And Beyond, Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[38] A. Niemetz, M. Preiner, C. Wolf, A. Biere, Btor2 , btormc and boolector 3.0, in: CAV (1), volume 10981 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 587–595.

[39] mcil, Developing an open-source, state-of-the-art symbolic model-checking framework for the model-checking research community, https://www.aere.iastate.edu/modelchecker/, 2022.

[40] H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, P. Rümmer, A verification toolkit for numerical transition systems - tool paper, in: FM, volume 7436 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 247–251.

[41] ntslib, Ntslib. http://nts.imag.fr/index.php/Main_Page, 2022.

[42] M. Y. Vardi, An automata-theoretic approach to linear temporal logic, in: Banff Higher Order Workshop, volume 1043 of *LNCS*, Springer, 1995, pp. 238–266.

[43] E. M. Clarke, O. Grumberg, K. Hamaguchi, Another look at LTL model checking, Formal Methods in System Design 10 (1997) 47–71.

[44] K. Claessen, N. Eén, B. Sterin, A circuit approach to LTL model checking, in: FMCAD, IEEE, 2013, pp. 53–60.

[45] A. Griggio, M. Roveri, S. Tonetta, Certifying proofs for SAT-based model checking, Form Methods Syst Des (2021). doi:10.1007/s10703-021-00369-1.

[46] ic3ia, Ic3ia. https://es-static.fbk.eu/people/griggio/ic3ia/, 2022.

[47] pyvmt, Pyvmt. https://github.com/pyvmt/pyvmt, 2022.