# An automata-based implementation of a symbolic CTL* model checker

Francesco Galla'

**Abstract**

This dissertation presents the implementation of a symbolic CTL* model checking algorithm based on multi-valued decision diagrams (MDDs). Given a Petri Net model and a CTL* proposition, the algorithm is capable of identifying LTL sub-formulae, translate them to Büchi automata and compute the synchronized product of each LTL formula with the model. MDDs are used to encode the Petri Net and such composition, provably lowering both system memory and time required to manipulate its graph of reachable states with respect to explicit model checking tecniques. By combining the sets of satisfying states for each LTL sub-formula according to the temporal quantifiers preceding them, the algorithm is capable of producing the boolean satisfaction result of the CTL* formula and a counterexample or witness run describing such outcome. Timed test runs executed againts a large set of models and specifications of LTL, CTL and CTL* temporal logics show competitive performance results with respect to other tools which process CTL* by translating each formula to $\mu$-calculus. This algorithm has been implemented as one of the free and open source programs composing the GreatSPN framework for formal verification of systems.

## 1 Introduction

### 1.1 Model Checking

Model checking is a formal verification technique intended to analyze properties of system designs. Given a formal model and a specification, the objective of model checking is to decide whether the behavior of the model satisfies the specification or not. The model is usually represented as a Kripke structure or by a high-level formalism that can be transformed into a Kripke structure. Examples of these formalisms are Petri Nets and Process algebra.

The specification is provided using a temporal logic expression, that is, a formula that expresses a temporal and logical statement. Temporal logics are modal logics geared towards the description of the temporal ordering of events. It is important to clarify that these logics do not consider precise timing requirements of activities or events, but reason about their *abstract temporal order*. For this reason, they are particularly useful when applied to concurrent systems in which all components proceed in a lock-step fashion over a discrete time domain. The system behavior is assumed to be observable at integral time points and each time point identifies a snapshot of all variables of the system, called *state*.

Two brands of temporal logics have been proposed over the years for specifying the properties of reactive systems. Linear Temporal Logic [22] is based on *linear time*, therefore considering every moment in time as having a unique possible future. Computational Tree Logic [9] is defined upon *branching time*: it pictures the structure of time as a tree, allowing each moment in time to split into different possible futures. From now on we will refer to these two logics using their well-known acronyms LTL and CTL respectively.

The difference between LTL and CTL is rooted in their satisfaction relations, which are conceptually different. LTL is said to be path-based, since a system $S$ satisfies a LTL formula $\phi$ if for all initial paths of $S$, paths starting in an intial state $s_0$ satisfy $\phi$. Conversely, CTL is said to be state-based, since a system $S$ satisfies a CTL formula $\phi$ if and only if $\phi$ holds in all initial states of $S$.

Furthermore, the expressiveness of LTL and CTL temporal logics is incomparable [19]. Conversely, CTL is particularly useful to express the *possibility* of existence of a specific path of execution of a model, that is, the occurrence of an event happening on one branch but not necessarily all of them. This concept cannot be expressed using a formalism based on linear time such as LTL, which describes executions of a system, not the way those executions are organized in a branching tree. On the contrary, CTL cannot express situations in which the same behavior may occur on distinct branches at distinct times, while the ability of LTL to describe individual paths is more convenient in this case. In practice, the LTL formula $\mathbf{FG}\phi$ is not expressible in CTL, while the formula $\mathbf{AFAG}\phi$ is not expressible in LTL. Given the shortcomings of both these temporal logics, a superset of LTL and CTL called CTL* [12] has been introduced.

## 1.2 A brief history of LTL model checking

Model checking LTL properties comes down to checking language emptyness of the syncronized product between the Kripke structure representing the model and a formalism which can represent a LTL formula while being translatable to a Kripke structure, namely a Büchi automaton. This *automata-theoretic approach* [25] treats the synchronized product as a transition system [18] whose state graph can be analyzed using tecniques classified in two main categories:

- *Explicit* methods process the state graph of the synchronized product using graph traversal algorithms.
- *Symbolic* methods represent the state graph using *decision diagrams* and usually apply fixed point algorithms to the set of states to find the strongly connected components (SCCs) of the transition system.

Explicit methods are based on graph traversal algorithms but are often limited by the complexity of the LTL model checking problem, which is constrained by the size of the state space of the model, the size of the underlying automaton used to represent the formula and the combined size of the two Kripke structures in a transition system. More specifically, the model checking problem for LTL is known to be PSPACE-complete [23]. Historically, the first LTL model checkers were explicit: a notable example is SPIN [14], which takes advantage of an optimized version of Tarjan DFS algorithm for finding strongly connected components in a graph, called *Nested Depth First Search* [15].

In practice, model checkers applied to complex, real-world sistems have to face the *state space explosion* problem: the exponential growth in the number of variables of the state graph dimension. Given that, in general, a system with $n$ variables over a domain of $k$ possible values requires at least $n^k$ states in the reachability set, it is understandable how even a simple model might necessitate a large reachability graph. Furthermore, dealing with real-valued variables, which have infinite possible assignments, results in a reachability graph with infinitely many states.

This problem has encouraged the development of various tecniques which have proven to be successful in mitigating the state explosion. Explicit model checkers have introduced *on-the-fly* state-space construction to avoid storing the whole state graph. The most basic *on-the-fly* algorithm [13] stores states which have already been visited in memory and is therefore able to check for cycles in the reachability graph while generating it through DFS. SPIN uses *on-the-fly* state graph construction combined with *partial order reduction* [21], a tecnique which reduces the size of the reachability graph by exploiting commutativity of concurrently executed transitions which result in the same state.

## 1.3 The origin of symbolic model checking

Efforts towards state space minimization have been particularly successful in developing clever representation of the state graph. *Symbolic* model checking represent the system model and the LTL formula using set of states and set of transitions. These sets can be represented as solutions to logical equations, using decision diagrams to represent this state space implicitly. Since syntactically small equations can represent large set of states, this tecnique ultimately avoids building the state graph explicitly, thus saving space in memory. Above all, Ordered Binary Decision Diagrams (OBDDs) provide a canonical form for boolean formulae which can be substantially more compact than conjunctive or disjuntive normal form and efficient algorithms have been implemented for manipulating them. McMillan was the first to introduce the use of OBDDs to represent the state space of a model, developing a CTL model checking tool called SMV [20]. Symbolic model checking is considered to be one of the biggest breakthrough in the history of model checking for its impact on the state explosion problem [8].

Symbolic model checking was initially applied to CTL because of the significantly lower complexity of the model checking problem with respect to LTL: CTL model checking is known to be P-Complete and its time complexity is bilinear in the size of the model and of the formula [10]. After the introduction of SMV, further research work increased the capabilities of decision diagrams. By introducing Multi-valued decision diagrams (MDDs) , tools were able to represent integral and real-valued functions, thus enhancing the applications of symbolic strategies to formal verification. In recent years, the LTSmin tool [17] developed by the University of Twente employs the SYLVAN multi-core MDDs library [24] to speed up symbolic analysis algorithms for CTL model checking. A different solution was adopted by an extended version of SMV, NuSMV [7], which mantained the specification language of McMillan's tool while improving it by introducing LTL model checking and Sat-based Bounded model checking [4], which exploits propositional satisfiability without using BDDs to represent the state graph. This approach was chosen because working with decision diagrams does not always guarantee an improvement over explicit model checking tecniques due to the often time-consuming procedure of selecting a variable ordering for all variables in the system, which is a known NP-complete problem [5].

Until the present day, our GreatSPN framework used multi-valued, multi-terminal decision diagrams (MTMDDs) provided by the Meddly library [3] to perform CTL model checking on Petri Nets [2]. Meddly is possibly the only open-source library to implement MTMDDs, Edge-valued MDDs (EVMDDs) while providing state-of-the-art algorithms to manipulate them. To determine the optimal variable ordering for the MDDs used to represet the state graph, GreatSPN uses a set of algorithms based on different heuristics which are run during the state space generation procedure [1]. As we are going to discuss later in this dissertation, GreatSPN is now capable of CTL* model checking by reducing symbolic LTL model checking to CTL model checking with fairness constraints, as demonstrated in a notorious article [11] by Clarke et al.

## 2 Background

### 2.1 Linear Temporal Logic

LTL is a propositional temporal logic with linear time model, meaning that it considers a single realized future behavior of a system, that is, a single path in a Kripke structure.

**Definition 1.** Syntax of LTL. *The formal syntax of LTL is given by the following grammar in Backus-Naur form (BNF), where* a ∈ AP *is an atomic proposition.*

$$\phi ::= \text{true} \mid \text{a} \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi\ \mathbf{U}\ \phi$$

Using boolean connectors such as $\neg$ and $\wedge$ allows LTL to be treated as a propositional logic. Other boolean connectives such as disjunction $\vee$, implication $\rightarrow$, equivalence $\leftrightarrow$ and the exclusive or (xor) operator $\oplus$ can be derived as for any other propositional logic. This LTL grammar is defined using two basic temporal modalities: **X** (next) and **U** (until). Using those, we can derive two essential temporal modalities **F** (eventually) and **G** (always), as follows:

$$\mathbf{F}\phi = \mathit{true}\mathbf{U}\phi$$
$$\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$$

We now present a list of the temporal modalities which will be used at a later time in this dissertation.

- $\mathbf{G}\phi$: "always" (now and forever in the future $\phi$ is true)
- $\mathbf{F}\phi$: "eventually" (eventually in the future $\phi$ is true)
- $\mathbf{X}\phi$: "next" (in the next time step $\phi$ is true)
- $\psi\mathbf{U}\phi$: "until" ($\psi$ is true until $\phi$ is true)

By combining the aforementioned temporal operators we obtain other, more complex modalities. A typical example is the specification which requires a property to be true *infinitely often*, $\mathbf{GF}\phi$.

**LTL Semantics** LTL semantics is defined for *infinite words* $\sigma$ over the alphabet $2^{\mathrm{AP}}$. The satisfiability rules are shown below:

- $\sigma \models true$
- $\sigma \models a$ iff $a \in A_0$
- $\sigma \models \phi_1 \wedge \phi_2$ iff $\sigma \models \phi_1$ and $\sigma \models \phi_2$
- $\sigma \models \neg\phi$ iff $\neg(\sigma \models \phi)$
- $\sigma \models \mathbf{X}\phi$ iff $\sigma[1...] \models \phi$
- $\sigma \models \phi_1\mathbf{U}\phi_2$ iff $\exists j \geq 0, \sigma[j...] \models \phi_2$ and $\sigma[i...] \models \phi_1$ for all $0 \leq i < j$

A LTL formula $\phi$ is said to be *valid* with regard to a Kripke structure $M$ if it holds for all paths of $M$. It is *satisfiable* if it holds for some path in $M$.

## 2.2 CTL*

We briefly introduced CTL* while dealing with the shortcomings of LTL and CTL. CTL* is a branching temporal logic which extends CTL following a proposal by Emerson and Halpern. Being based on the concept of branching time, CTL* is able to represent the possibility of existence of a determinate behavior in a tree of execution, using CTL path quantifiers **E** (for some path) and **A** (for all paths) to specify whether the required behavior must be verified for some execution of our system or all possible ones.

CTL* allows path quantifiers to be arbitrarily nested with linear temporal operators **G**, **F**, **X** and **U**. In contrast, CTL only supports linear temporal operators if they are immediately preceded by a path quantifier. In a similar fashion as CTL, the syntax of CTL* distinguishes between *state* and *path* formulae. CTL* path formulae are defined as LTL formulae, whith the only difference that here state formulae can be used as atoms.

**Definition 2.** Syntax of CTL*. *The formal syntax of* CTL* *is made of state formulae* $\Phi$ *and path formulae* $\phi$. *The syntax of* CTL* *state formulae* $\Phi$ *is defined over the set* AP *of atomic propositions:*

$$\Phi ::= \text{true} \mid \text{a} \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{E}\phi$$

*The syntax of CTL* path formulae* $\phi$ *is given by the following grammar, where* $\Phi$ *is a state formula:*

$$\phi ::= \Phi \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \phi \, \mathbf{U} \, \phi$$

As previously seen for LTL, the syntax of CTL* can be treated as any propositional logic, therefore all boolean connectives can be derived from $\neg$ and $\wedge$. The missing temporal modalities **F** and **G** descend from **X** and **U** as it was the case for LTL, while the path quantifier **A** can be obtained from the following equivalence:

$$\mathbf{A}\phi = \neg\mathbf{E}\neg\phi$$

**CTL* Semantics** Let a $\in AP$ be an atomic proposition, $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, state $s \in S$, $\Phi$ and $\Psi$ be CTL* state formulae and $\phi$, $\phi_1$ and $\phi_2$ be CTL* path formulae.

**State formulae: semantics**

- $s \models a$ iff $a \in$ *L(s)*
- $s \models \Phi \wedge \Psi$ iff $s \models \Phi$ and $s \models \Psi$
- $s \models \neg\Phi$ iff $\neg(s \models \Phi)$
- $s \models \mathbf{E}\phi$ iff $\sigma \models \phi$ for some $\sigma \in$ *Paths(s)*

**Path formulae: semantics**

- $\sigma \models \Phi$ iff $s_0 \models \Phi$
- $\sigma \models \phi_1 \wedge \phi_2$ iff $\sigma \models \phi_1$ and $\sigma \models \phi_2$
- $\sigma \models \neg\phi$ iff $\neg(\sigma \models \phi)$
- $\sigma \models \mathbf{X}\phi$ iff $\sigma[1...] \models \phi$
- $\sigma \models \phi_1 \mathbf{U} \phi_2$ iff $\exists j \geq 0, \sigma[j...] \models \phi_2$ and $\sigma[i...] \models \phi_1$ for all $0 \leq i < j$

## 2.3 Decision Diagrams

Decision diagrams are directed, acyclic graphs used to represent functions over variables with finitely many possible assignments. They were originally studied by Bryant [6] as a representation of boolean functions in the form of Binary Decision Diagrams (BDDs). We focus on a generalization of BDDs called Multi-value Decision Diagrams (MDDs) [16], whose variable domain can be arbitrarely large and which can be used to represent functions of integral and real-valued variables.

**Definition 3.** Multi-valued Decision Diagram. *A multi-valued desicion diagram, or MDD, is an acyclic graph in which each node represents a function over variables with finitely many possible assignments, of the form:*

$$f : S_K \times ... \times S_1 \rightarrow \{0, \dots, m-1\}$$

*Each one of the sets $S_k$ is considered to be finite on an arbitrarely large domain. We can write:*

$$S_k = \{0, 1, \dots, n_k - 1\}$$

An MDD is composed of two types of nodes: *terminal* nodes and *non-terminal* nodes.

- Terminal nodes are labeled with values from the set $\{0, 1, \dots, m-1\}$, representing the constant function:

$$g(x_K, \dots, x_1) = a$$

- Non-terminal nodes are labeled with one of the function variables $x_k$ and contain $n_k$ arcs to other nodes.

A non terminal node labeled with $x_{\mathrm{k}}$ has an outgoing arc corresponding to value $v$ which goes to a node representing the function:

$$f_{x_k=v}(x_K, \dots, x_1) \equiv f(x_K, \dots, x_{k+1}, v, x_{k-1}, \dots, x_1)$$

**Reduced ordered MDDs** In the contest of model checking, MDDs are useful to encode the state space of a model as a boolean, integral or real-valued function. To fit this purpose, MDDs must be in *canonical* form, meaning that for any given function and variable ordering, there must be exactly *one* representation of that function as MDD. For this to be true, we have to impose two constraints on MDDs: that they are *ordered* and *reduced* (ROMDDs).

**Definition 4.** Ordered MDDs. *An MDD is ordered if all paths through the MDD visit non-terminal nodes accortding to the same variable ordering.*

**Definition 5.** Reduced Ordered MDDs. *A reduced, ordered MDD is an ordered MDD that contains no duplicate nodes and no redundant nodes.*

- *Two nodes are* ***redundant*** *if all of its outgoing arcs point to the same node.*
- *Two terminal nodes are* ***duplicates*** *if they have the same label, while two non-terminal nodes are duplicates if they have the same variable label and the same outgoing arcs for each value.*

### 2.4 Automata over infinite words

We've described how Linear Temporal Logic provides a language to describe the temporal order of a series of events. When applied to formal verification, these sequences of events can be interpreted as computations of the program. A computation is a potentially infinite sequence of program states: each state is described by a finite set of atomic propositions, which will be referred to as a nonempty *alphabet*. Therefore a computation can be treated as an infinite word over the alphabet of truth assignments to the atomic propositions of a given alphabet.

This reasoning suggests that a LTL specification can be thought of as a description of a *language* over some alphabet. This language is made of infinite *words*, which represent program computations. We can exploit this equivalence of computations and words to connect linear temporal logic to automata theory applied on infinite words. As we are going to show, automata over infinite words depict a suitable formalism to represent LTL specifications. More precisely, given any propositional temporal formula, one can construct a *finite automaton* over infinite words that accepts precisely the computations satisfied by the formula [26]. This chapter will present an introduction to automata theory and describe how LTL model checking employs a particular class of automata over infinite words, called *Büchi automata.*

## References

[1] Amparore, E. G., Donatelli, S., Beccuti, M., Garbi, G., and Miner, A. S. Decision diagrams for petri nets: Which variable ordering? In *Proceedings of the International Workshop on Petri Nets and*

*Software Engineering (PNSE'17), co-located with the38th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2017 and the 17th International Conference on Application of Concurrency to System Design ACSD 2017, Zaragoza, Spain, June 25-30, 2017* (2017), D. Moldt, L. Cabac, and H. Rölke, Eds., vol. 1846 of *CEUR Workshop Proceedings*, CEUR-WS.org, pp. 31–50.

[2] Babar, J., Beccuti, M., Donatelli, S., and Miner, A. S. Greatspn enhanced with decision diagram data structures. In *Applications and Theory of Petri Nets, 31st International Conference, PETRI NETS 2010, Braga, Portugal, June 21-25, 2010. Proceedings* (2010), J. Lilius and W. Penczek, Eds., vol. 6128 of *Lecture Notes in Computer Science*, Springer, pp. 308–317.

[3] Babar, J., and Miner, A. S. Meddly: Multi-terminal and edge-valued decision diagram library. In *QEST 2010, Seventh International Conference on the Quantitative Evaluation of Systems, Williamsburg, Virginia, USA, 15-18 September 2010* (2010), IEEE Computer Society, pp. 195–196.

[4] Biere, A., Cimatti, A., Clarke, E. M., and Zhu, Y. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings* (1999), R. Cleaveland, Ed., vol. 1579 of *Lecture Notes in Computer Science*, Springer, pp. 193–207.

[5] Bollig, B., and Wegener, I. Improving the variable ordering of obdds is np-complete. *IEEE Trans. Computers 45*, 9 (1996), 993–1002.

[6] Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers 35*, 8 (1986), 677–691.

[7] Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings* (2002), E. Brinksma and K. G. Larsen, Eds., vol. 2404 of *Lecture Notes in Computer Science*, Springer, pp. 359–364.

[8] Clarke, E. M. The birth of model checking. In *25 Years of Model Checking - History, Achievements, Perspectives* (2008), O. Grumberg and H. Veith, Eds., vol. 5000 of *Lecture Notes in Computer Science*, Springer, pp. 1–26.

[9] Clarke, E. M., and Emerson, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981* (1981), D. Kozen, Ed., vol. 131 of *Lecture Notes in Computer Science*, Springer, pp. 52–71.

[10] Clarke, E. M., Emerson, E. A., and Sistla, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (1986), 244–263.

[11] Clarke, E. M., Grumberg, O., and Hamaguchi, K. Another look at LTL model checking. *Formal Methods in System Design 10*, 1 (1997), 47–71.

[12] Emerson, E. A., and Halpern, J. Y. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM 33*, 1 (1986), 151–178.

[13] Fernandez, J., Mounier, L., Jard, C., and Jéron, T. On-the-fly verification of finite transition systems. *Formal Methods in System Design 1*, 2/3 (1992), 251–273.

[14] Holzmann, G. J. The model checker SPIN. *IEEE Trans. Software Eng. 23*, 5 (1997), 279–295.

[15] Holzmann, G. J., Peled, D. A., and Yannakakis, M. On nested depth first search. In *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996* (1996), J. Grégoire, G. J. Holzmann, and D. A. Peled, Eds., vol. 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, DIMACS/AMS, pp. 23–31.

[16] Kam, T., Villa, T., and Brayton, R. Multi-valued decision diagrams: Theory and applications," multiple-valued logic. *Multiple-Valued Logic 4* (01 1998).

[17] Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., and van Dijk, T. Ltsmin: High-performance language-independent

model checking. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings* (2015), C. Baier and C. Tinelli, Eds., vol. 9035 of *Lecture Notes in Computer Science*, Springer, pp. 692–707.

[18] Keller, R. M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (1976), 371–384.

[19] Lamport, L. "sometime" is sometimes "not never" - on the temporal logic of programs. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nevada, USA, January 1980* (1980), P. W. Abrahams, R. J. Lipton, and S. R. Bourne, Eds., ACM Press, pp. 174–185.

[20] McMillan, K. L. *Symbolic model checking.* Kluwer, 1993.

[21] Peled, D. A. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design 8*, 1 (1996), 39–64.

[22] Pnueli, A. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* (1977), IEEE Computer Society, pp. 46–57.

[23] Sistla, A. P., and Clarke, E. M. The complexity of propositional linear temporal logics. *J. ACM 32*, 3 (1985), 733–749.

[24] van Dijk, T., and van de Pol, J. Sylvan: multi-core framework for decision diagrams. *STTT 19*, 6 (2017), 675–696.

[25] Vardi, M. Y. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings)* (1995), F. Moller and G. M. Birtwistle, Eds., vol. 1043 of *Lecture Notes in Computer Science*, Springer, pp. 238–266.

[26] Vardi, M. Y., and Wolper, P. Reasoning about infinite computations. *Inf. Comput. 115*, 1 (1994), 1–37.