Model Checking

William Schultz

January 24, 2023

At a high level, *model checking* is a computer assisted method for the analysis of dynamical systems that can be modeled as discrete state transition systems [CHV18]. In the basic setting, we can view the model checking problem as taking in a transition system (e.g. a Kripke structre) K and some system specification φ (typically specified in some temporal logic) and verifying whether

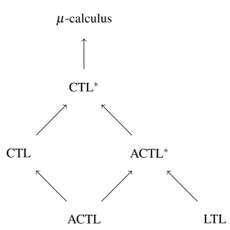
$$K \vDash \varphi$$

and, if $K \nvDash \varphi$, returning a counterexample. In the basic scenario, we can assume K is finite state.

1 Temporal Logic Model Checking

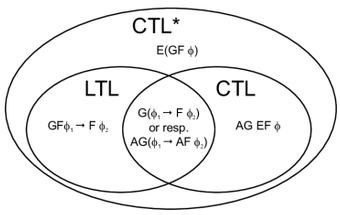
There are a variety of temporal logics that have been used to reason about properties of programs/systems. A high level overview is shown in the relationship diagram below, where CTL* is one of the most expressive logics (encompassing both CTL and LTL).

Fig. 5 Basic temporal logics and their relationships. All inclusions are strict. ACTL is the intersection of ACTL* and CTL



In practice, I don't think it's that important to worry much about the finer distinctions between the various logics, since typically I am more concerned with how to express a desired correctness property (e.g. for which LTL may often be sufficient). Nevertheless, it is good to have a general view of the classification hierarchy to understand the landscape (and since various papers may choose to express/formalize things in different choices of logics).

The below diagram also shows the relationship between LTL, CTL, and CTL*. Namely, that LTL and CTL are not directly comparable, and CTL* subsumes them both.



The following diagram [CHV18] also provides a good overview of common varieties of temporal logic properties and their counterexample characterizations.

Table 2 Examples of CTL* formulas and respective counterexamples

	Sub-logic	Formula	Intuition	Counterexample
1	CTL, LTL	$\mathbf{AG}p$	p is an invariant	finite path leading to $\neg p$
2	CTL, LTL	$\mathbf{AF}p$	p must eventually hold	infinite path (lasso-shaped) without p
3	CTL, (negated) LTL	$\mathbf{EF}\neg p$	$\neg p$ is reachable	substructure with all reachable states, all containing p
4	CTL, LTL	$\mathbf{AG}(p \vee \mathbf{X}p) = \mathbf{AG}(p \vee \mathbf{AX}p)$	p holds at least every other state	finite path leading to $\neg p$ twice in a row
5	CTL, LTL	$\mathbf{AGF}p = \mathbf{AGAF}p$	p holds infinitely often	infinite path (lasso) on which p occurs only finitely often
6	CTL, LTL	$\mathbf{AG}(p \rightarrow \mathbf{F}q) = \mathbf{AG}(p \rightarrow \mathbf{AF}q)$	every p is eventually followed by q	finite path leading to p, but no q now nor on the infinite path (lasso) afterwards
7	CTL, (boolean combination of) LTL	$(\mathbf{AGF}p) \wedge \mathbf{EF}\neg p$	both 3 and 5 hold	either counterexample for $\mathbf{AGF}p$ or for $\mathbf{EF}\neg p$
8	CTL only	$\mathbf{AGEX}p$	reachability of p in one step is an invariant	finite path leading to a state whose successors all have $\neg p$
9	CTL only	$\mathbf{AG}(p \vee \mathbf{AXAG}q \vee \mathbf{AXAG}\neg q)$	once p does not hold, either q or $\neg q$ become invariant in one step	finite path leading to $\neg p$ from which two finite extensions reach q and $\neg q$
10	LTL only	$\mathbf{AFG}p$	p must eventually become an invariant	infinite path (lasso) on which $\neg p$ occurs infinitely often
11	LTL only	$\mathbf{A}(\mathbf{GF}p \rightarrow \mathbf{GF}q)$	if p holds infinitely often, so does q	infinite path (lasso) on which p occurs infinitely often, but q does not

CTL Model Checking

For *computation tree logic* (CTL), there is a model-checking algorithm whose running time depends linearly on the size of the Kripke structure and on the length of the CTL formula [CES86].

LTL Model Checking

For *linear temporal logic* (LTL) it is the case that any counterexample to a property ψ is w.l.o.g. restricted to have a "lasso" shape $v \cdot w^{\omega}$ i.e., an initial path (prefix) followed by an infinitely repeated finite path (cycle) [WVS83]. Certain LTL properties have even simpler counterexamples e.g. safety properties always have finite paths as counterexamples. Note that the "lasso"-ness of LTL counterexamples is exploited in certain model checking approaches e.g. some liveness to safety reductions [BAS02].

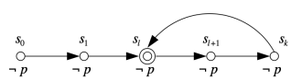


Fig. 1. A generic lasso-shaped counter example trace for $\mathbf{AF}p$.

There is an LTL model checking algorithm whose running time depends linearly on the size of the Kripke structure and exponentially on the length of the LTL formula [LP85]. This is done by translating an LTL specification ψ into a Büchi automata B_{ψ} over the alphabet 2^A (where A is the set of atomic propositions) such that for all Kripke structures K and infinite paths π, the infinite word $L(\pi)$ is accepted by B_{ψ} iff π is a counterexample of ψ in K. The size of the automaton B_{ψ}, though, can be exponential in the length of the formula ψ.

More precisely, we translate the negation of our property $\neg\psi$ into a Buchi automaton, and then consider the product of this automaton with our original system $K \times B_{\neg\psi}$. The problem then reduces to checking whether there are any accepting runs in $K \times B_{\neg\psi}$. Recall that an accepting run in a (deterministic) Buchi automaton is any run that visits an accepting state infinitely often. A standard algorithm for checking existence of an accepting run consists of

1. Consider the automaton as a directed graph and decompose it into its strongly connected components (SCCs).
2. Run a search to find which SCCs are reachable from the initial state
3. Check whether there is a non-trivial SCC (i.e. consists of ≥ 1 vertex) that is reachable and contains an accepting state.

Technically, the LTL model checking problem is PSPACE-complete [SC85], but it's worth keeping in mind that in practice the limiting complexity factor is usually the size of a system's state space, rather than the size of the temporal specification. (TODO: why PSPACE?)

2 Abstraction for Model Checking

Abstraction, in the context of model checking, is generally aimed at reducing the size of the state space in an attempt to remove details that are irrelevant to the property being verified [DG18]. That is, broadly, abstraction is a fundamental tool in tackling the "state explosion" problem.

Abstraction of Kripke Structures

In general, an abstraction framework defines a set of concrete objects and abstract objects and a definition of how to map between them. For model checking, we typically use Kripke structures as our concrete objects. Recall that a *Kripe structure* $M = (AP, S, I, R, L)$ is defined as

- a set AP of atomic propositions
- a set of states S
- a set of initial states $I \subseteq S$
- a transition relation $R \subseteq S \times S$
- a labeling function $L : S \to 2^{AP}$

Simulation

To define a notion of abstraction for Kripke structures, we define a few standard relations between two structures M_1 and M_2. *Simulation* is a preorder (a reflexive and transitive partial order) in which the larger structure may have more behaviors, but possibly fewer states and transitions.

Let $M_1 = (AP_1, S_1, I_1, R_1, L_1)$ and $M_2 = (AP_2, S_2, I_2, R_2, L_2)$ be Kripke structures such that $AP_2 \subseteq AP_1$. A relation H is a *simulation relation from M_1 to M_2* if for every $s_1 \in S_1$ and $s_2 \in S_2$ such that $H(s_1, s_2)$, both of the following conditions hold:

- $\forall p \in AP_2 : p \in L_1(s_1) \iff p \in L_2(s_2)$
- $\forall t_1 \in S_1 : R_1(s_1, t_1) \Rightarrow \exists t_2 : (R_2(s_2, t_2) \wedge H(t_1, t_2))$

The second condition states that for every transition of the "smaller" (i.e. more concrete) system M_1, there must exist a corresponding transition in the larger system M_2. We say that M_1 *is simulated by* M_2 (or M_2 *simulates* M_1) (denoted $M_1 \leq M_2$) if there exists a simulation relation H from M_1 to M_2 such that

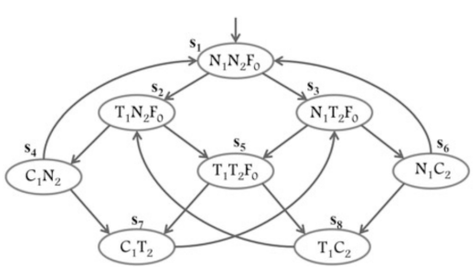
$$\forall s_1 \in I_1 : (\exists s_2 \in I_2 : H(s_1, s_2))$$

For example, consider the concrete Kripke structure M, modeling a mutual exclusion program where its atomic propositions $AP = \{N_1, T_1, C_1, N_2, T_2, C_2, F_0\}$:

Fig. 1 Process P_i

$$\begin{array}{lcl} v_i = \mathit{Neutral} & \to & v_i := \mathit{Trying} \\ v_i = \mathit{Trying} \ \wedge \ \mathit{Flag} = \mathrm{tt} & \to & v_i := \mathit{Critical};\ \mathit{Flag} := \mathrm{ff} \\ v_i = \mathit{Critical} & \to & v_i := \mathit{Neutral};\ \mathit{Flag} := \mathrm{tt} \end{array}$$

Fig. 2 Kripke structure M for the mutual exclusion program



and then an abstraction of this structure, M_1, with atomic propositions $AP_1 = \{C_1, C_2, F_0\}$:

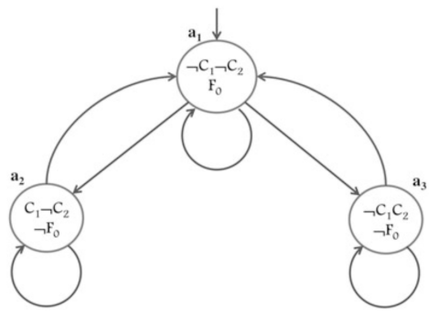


Fig. 3 Abstract Kripke structure M_1 for the mutual exclusion program

This abstraction basically only tracks whether a particular process is in the critical section or not, but ignores all other information. Note that $AP_1 \subseteq AP$. A simulation relation $H \subseteq S \times S_1$ from M to M_1 can then be defined as

$$H = \{(s_1, a_1), (s_2, a_1), (s_3, a_1), (s_4, a_2), (s_5, a_1), (s_6, a_3), (s_7, a_2), (s_8, a_3)\}$$

Bisimulation

One state is related to another by the bisimulation relation If they agree on their common atomic propositions and, in addition, for every successor of one state there is a corresponding successor of the other state, and *vice versa*.

Existential Abstraction

One way to define an abstract model (Kripke structure) from a concrete one is via a concretization function γ. We can define abstract Kripke structures by means of *existential abstraction* [CGL94]. Given a set $\widehat{S}$ of abstract states, the *concretization function* $\gamma : \widehat{S} \to 2^S$ indicates, for each abstract state $\widehat{s}$, what set of concrete states are represented by $\widehat{s}$. Similarly, there is a transition from abstract state $\widehat{s}$ to another abstract state $\widehat{s}'$ if there is a transition from a state represented by $\widehat{s}$ to a state represented by $\widehat{s}'$. Essentially, we just take every transition between concrete states and add it into our abstract transition system, based on the abstract states that represent those concrete state transitions.

Counterexample-Guided Abstraction Refinement (CEGAR)

Regardless of how we choose our abstraction, our abstract model $\widehat{M}$ generally contains less information than the concrete model M, and so model checking $\widehat{M}$ may produce incorrect results. If a universal property is true in $\widehat{M}$ then it is also true in M, but if the abstract model produces an error, the concrete model may still be correct.

For example consider the following "traffic light" model and a simple abstraction of it

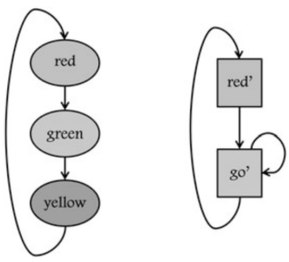


Fig. 6 Abstraction of a traffic light

If we wanted to check the universal CTL property $\mathbf{AGAF}(IsRed)$ (i.e. along all paths, $IsRed$ holds infinitely often), this clearly holds in the concrete traffic light model, but fails in the abstract model. When an abstract counterexample does not correspond to any concrete counterexample, we call it *spurious*.

Consider another example of a spurious counterexample, shown as follows:

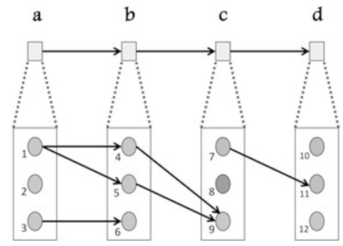


Fig. 7 Spurious counterexample. The abstract state c is a failure state

in this case, reachability of state 11 is not preserved by the abstraction. That is, it is not reachable in the lower level system, but it is reachable in the abstract system (consisting of abstract states $\{a, b, c, d\}$).

The framework of *counterexample-guided abstraction refinement (CEGAR)* deals with this issue. The main steps of CEGAR are as follows:

1. Given a concrete model M and some universal temporal formula to check, ψ, generate an initial abstract model $\widehat{M}$.
2. Model check $\widehat{M}$ with respect to ψ. If $\widehat{M}$ satisfies ψ, then conclude that the concrete model satisfies ψ and terminate. If a counterexample $\widehat{T}$ is found, check whether it is also a counterexample in the concrete model.
   - If it is, conclude that the concrete model does not satisfy the formula and stop.
   - Otherwise, the counterexample is spurious, and proceed to step 3.
3. Refine the abstract model, $\widehat{M}$, so that $\widehat{T}$ will not be included in the new, refined abstract model. Go back to step 2.

Note that refinement is typically done by *partitioning* an abstract state. That is, the set of concrete states represented by the abstract state is partitioned

Identifying Spurious Counterexamples

If we discover an abstract counterexample $\widehat{T}$, we need some way to check if this is a real counterexample in the concrete model. Assume that $\widehat{T}$ is a path $\widehat{s_1}, \ldots, \widehat{s_n}$ starting at the initial abstract state $\widehat{s_1}$. We can extend the concretization function γ to sequences of abstract states as follows: $\gamma(\widehat{T})$ is the set of concrete paths defined as:

$$\gamma(\widehat{T}) = \left\{ \langle s_1, \ldots, s_n \rangle \mid \bigwedge_{i=1}^{n} s_i \in \gamma(\widehat{s_i}) \wedge I(s_1) \wedge \bigwedge_{i=1}^{n} R(s_i, s_{i+1}) \right\}$$

Then, we need an algorithm to compute a sequence of sets of states that correspond to $\gamma(\widehat{T})$. We let $S_1 = \gamma(\widehat{s_1}) \cap I$, and then define

$$S_i := Image(S_{i-1}, R) \cap \gamma(\widehat{s_i})$$

where $Image(S_{i-1}, R)$ is the set of successors, in M, of the states in S_{i-1}. Basically, we just want to symbolically execute concrete model, starting from the concretized version of the initial abstract counterexample state, and, at each step, check whether there is some concrete state in this image set that corresponds to the set of states from the abstract counterexample. We can formalize this into the following lemma. Specifically, the following are equivalent:

1. The path $\widehat{T}$ corresponds to a concrete counterexample.
2. The set $\gamma(\widehat{T})$ of concrete paths is non-empty.
3. For all $1 \leq i \leq n, S_i \neq \emptyset$.

Note that checking whether a counterexample is spurious involves computations on the concrete model.

SAT-based Abstraction

An alternative to the CEGAR based approach (introduced shortly after the initial CEGAR publication) is to rely more directly on a SAT solver for performing abstraction. A main idea is to do bounded model checking and then use proofs of unsatisfiability in this case to provide an *explanation* of correctness, and to help us generate an abstraction for proving a property in the unbounded case. Using such a proof to generate an abstraction is called *proof-based abstraction*[MA03].

In the initial work of [MA03] (appeared in *TACAS 2003*, April), verification of a system M is done by performing bounded model checking of M for a fixed bound k. If an error is found for this bound, then we're done, since a counterexample has been found. Otherwise, the SAT solver can return a (resolution) proof of unsatisfiability. This proof is used to generate an abstraction, M', of M by seeing what clauses of the encoding of M are actually used in the proof (see [MA03] for more detail on this). Then, *unbounded* model checking is performed on the abstract model M', which should in theory now be much cheaper since M' has been abstracted from M.If model checking determines that there are no error traces in M', then we are done. Otherwise, if model checking determines that M' does have an error run, then we know its length k' is greater than k. Thus, we then restart the procedure with k' (or, generally, any value larger than k')

Interpolation

Slghtly subsequent work by McMillan [McM03] (appeared in *CAV 2003*, July) presented an extension of this approach that makes use of *interpolation* for doing abstraction with a SAT solver. Essentially this uses a similar bounded model checking approach and then extracts an *interpolant* from a bounded unsatisfiability proof to compute an abstraction.

The standard bounded model checking unrolling is a constraint of the following basic form:

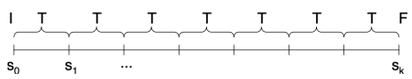


Fig. 1. Bounded model checking.

The idea of the interpolation based approach is to partition the path constraint into two sets A and B:

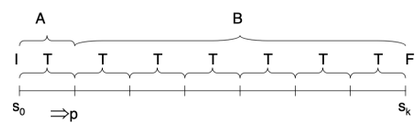


Fig. 2. Computing image by interpolation.

Then, if we produce a proof of unsatisfiability of the whole unrolling constraint, we derive an interpolant P of the pair (A, B), where the common variables of A and B are exactly those variables representing state s_1, in the sample diagram above. Note that an interpolant P for two formulas A, B is one such that

- $A \Rightarrow P$
- P refers only to the common variables of A and B
- $P \wedge B$ is unsatisfiable

By this, we can know that P is implied by the initial condition and by the first transition constraint, so it represents some approximation of all 1-step reachable states (i.e. it is an over-approximation of the forward image of I). Moreover, P and B are unsatisfiable, meaning that no state satisfying P can reach a final state in $k-1$ steps. This over-approximate image operation can then be iterated to compute an over-approximation of the reachable states.

References

[BAS02] Armin Biere, Cyrille Artho, and Viktor Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002. FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop).

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, apr 1986.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, sep 1994.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. *Introduction to Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[DG18] Dennis Dams and Orna Grumberg. *Abstraction and Abstraction Refinement*, pages 385–419. Springer International Publishing, Cham, 2018.

[LP85] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, page 97–107, New York, NY, USA, 1985. Association for Computing Machinery.

[MA03] Kenneth L. McMillan and Nina Amla. Automatic abstraction without counterexamples. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'03, page 2–17, Berlin, Heidelberg, 2003. Springer-Verlag.

[McM03] K. L. McMillan. Interpolation and sat-based model checking. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 1–13, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, jul 1985.

[WVS83] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*, SFCS '83, page 185–194, USA, 1983. IEEE Computer Society.