Fast, Flexible, and Minimal CTL Synthesis via SMT

Tobias Klenze[12], Sam Bayless[1], and Alan J. Hu[1]

[1] University of British Columbia, Canada
{sbayless,ajh}@cs.ubc.ca
[2] Technische Universität München, Germany
tobias.klenze@mytum.de

Abstract. CTL synthesis [8] is a long-standing problem with applications to synthesising synchronization protocols and concurrent programs. We show how to formulate CTL model checking in terms of "monotonic theories", enabling us to use the *SAT Modulo Monotonic Theories* (SMMT) [5] framework to build an efficient SAT-modulo-CTL solver. This yields a powerful procedure for CTL synthesis, which is not only faster than previous techniques from the literature, but also scales to larger and more difficult formulas. Additionally, because it is a constraint-based approach, it can be easily extended with further constraints to guide the synthesis. Moreover, our approach is efficient at producing *minimal* Kripke structures on common CTL synthesis benchmarks.

1 Introduction

Computation Tree Logic (CTL) is widely used in the context of model checking, where a CTL formula specifying a temporal property, such as safety or liveness, is checked for validity in a program or algorithm (represented by a Kripke structure). Both the branching time logic CTL and its application to model checking were first proposed by Clarke and Emerson [8]. In that work, they also introduced a decision procedure for CTL satisfiability, which they applied to the synthesis of *synchronization skeletons*, abstractions of concurrent programs which are notoriously difficult to construct manually. Though CTL model checking has been a phenomenal success, there have been fewer advances in the field of CTL synthesis, due to its high complexity.

In CTL synthesis, a system is specified by a CTL formula, and the goal is to find a model of the formula — a Kripke structure in the form of a transition system in which states are annotated with sets of atomic propositions (so called *state properties*). The most common motivation for CTL synthesis remains the synthesis of synchronization for concurrent programs, such as mutual exclusion protocols. In this setting, the Kripke structure is interpreted as a global state machine in which each global state contains every process's internal local state. The CTL specification in this setting consists of both structural intra-process constraints on local structures, and inter-process behavioral constraints on the global structure (for instance, starvation freedom). If a Kripke structure is found which satisfies the CTL specification, then one can derive from it the guarded commands that make up the corresponding synchronization skeleton [8, 4].

In this paper, we introduce a novel method for CTL synthesis. We build on the recent introduction of SAT modulo Monotonic Theories (SMMT) [5], creating a CTL satisfiability procedure for the case where the number of states in the Kripke structure is bounded in advance. (Note, however, that the underlying CTL model checking theory is for the standard, unbounded semantics of CTL.) Due to the CTL small model property [12], in principle a bounded CTL-SAT procedure yields a complete decision procedure for unbounded CTL-SAT, but in practice, neither bounded approaches, nor classical tableau approaches, have been scalable enough for completeness to be a practical concern. Rather, our approach (like similar constraint-solver based techniques for CTL [14, 10] and LTL [17, 15]) is appropriate for the case where a formula is expected to be satisfiable by a Kripke structure with a modest number of states (~ 100). Nevertheless, we will show that our approach solves larger and more complex satisfiable CTL formulas, including ones with a larger numbers of states, much faster than existing bounded and unbounded synthesis techniques. This makes our approach particularly appropriate for CTL synthesis.

In addition to being more efficient than existing techniques, our approach is also capable of synthesizing minimal models. As we will discuss below, previous CTL synthesis approaches were either incapable of finding minimal models [8, 3], or could not do so with comparable scalability to our technique [14, 10].

The paper is structured as follows: We begin with a review of related work in Section 2. To make this paper self-contained, we go over the theory behind SAT Modulo Monotonic Theories in Section 3 and some challenges in applying it to CTL. In the same section, we show how to utilize this framework for bounded CTL synthesis. Section 4 explains the most important implementation details and optimizations. The experimental results of Section 5 demonstrate that our implementation, based on the open-source SMT solver MonoSAT[3] for Boolean monotonic theories, is able to outperform other approaches in two families of synthesis benchmarks: one derived from mutual exclusion protocols, and the other derived from readers-writers protocols.

2 Related Work

The original 1981 Clarke and Emerson paper introducing CTL synthesis [8] proposed a tableau-based synthesis algorithm, and used this algorithm to construct a 2-process mutex in which each process was guaranteed mutually exclusive access to the critical section, with starvation freedom.

Subsequently, although there has been steady progress on the general CTL synthesis problem, the most dramatic gains have been with techniques that are structurally-constrained, taking a CTL formula along with some additional 'structural' information about the desired Kripke structure, not specified in CTL, which is then leveraged to achieve greater scalability than generic CTL synthesis techniques. For example, in 1998, Attie and Emerson [3, 2] introduced a CTL

[3] http://www.cs.ubc.ca/labs/isd/Projects/monosat/

synthesis technique for the case where the Kripke structure is known to be composed of multiple similar communicating processes. They used this technique to synthesize a Kripke structure for a specially constructed 2-process version of the CTL formula (a 'pair-program') in such a way that the produced Kripke structure could be safely generalized into an N-process solution. This allowed them to produce a synchronization skeleton for a mutex with 1000 or more processes, far larger than other techniques. However, while this process scales very well, only certain CTL properties can be guaranteed to be preserved in the resulting Kripke structure, and in general the Kripke structure produced this way may be much larger than the minimal solution to the instance. In particular, `EX` and `AX` properties are not preserved in this process [2].

The similar-process synthesis techniques of Attie and Emerson rely on a generic CTL synthesis method to synthesize these pair-programs. As such, improvements to the scalability or expressiveness of generic CTL synthesis methods can be directly applied to improving this pair-program synthesis technique. Their use of the synthesis method from [8] yields an initially large Kripke structure that they minimize in an intermediate step. We note that our approach is particularly suited for synthesizing such pair-programs, not merely for performance reasons, but also because it is able to synthesize minimal models directly.

On the topic of finding minimal models, Bustan and Grumberg [7] introduced a technique for minimizing Kripke structures. However, the minimal models that our technique produces can in general be smaller than what can be achieved by starting with a large Kripke structure and subsequently minimizing it. This is because minimization techniques which are applied on an existing Kripke structure *after* its synthesis only yield a structure minimal with respect to equivalent structures (for some definition of equivalence, *e.g.,* strong or weak bisimulation). This does not necessarily result in a structure that is the overall minimal model of the original CTL formula. For this reason, techniques supporting the direct synthesis of minimal models, such as ours, have an advantage over post-synthesis minimization techniques.

In 2005, Heymans et al. [14] introduced a novel, constraint-based approach to the general CTL synthesis problem. They created an extension of answer set programming (ASP) that they called 'preferential ASP' and used it to generate a 2-process mutex with the added property of being 'maximally parallel', meaning that each state has a (locally) maximal number of outgoing transitions (without violating the CTL specification). They argued that this formalized a property that was implicit in the heuristics of the original 1981 CTL synthesis algorithm, and that it could result in Kripke structures that were easier to implement as efficient concurrent programs. As the formulation in their paper does not require additional structural constraints (though it can support them), it is a general CTL synthesis method. Furthermore, being a constraint-based method, one can flexibly add structural or other constraints to guide the synthesis. However, the scalability of their method was poor.

Subsequently, high performance ASP solvers [13] built on techniques from Boolean satisfiability solvers were introduced, allowing ASP solvers to solve much

larger and much more difficult ASP formulas. In 2012, De Angelis, Pettorossi, and Proietti [10] showed that (unextended) ASP solvers could also be used to perform efficient bounded CTL synthesis, allowing them to use the high performance ASP solver Clasp [13]. Similar to [3], they introduced a formulation for doing CTL synthesis via ASP in the case where the desired Kripke structure is composed of multiple similar processes. Using this approach, they synthesized 2-process and 3-process mutexes with properties at least as strong as the original CTL specification from [3]. The work we introduce in this paper is also a constraint-solver-based, bounded CTL-synthesis technique. However, we will show that our approach scales to larger and more complex specifications than previous work, while simultaneously avoiding the limitations that prevent those approaches from finding minimal models.

Our approach is based on *SAT Modulo Monotonic Theories* (SMMT), introduced by Bayless et al. in 2015 [5]. This is a technique for building lazy SMT solvers [18, 11] for a class of theories they defined as *Boolean monotonic theories*. The restriction to Boolean monotonic theories appears rather limiting, but in this paper, we will show how SMMT can be used to build an SMT solver for the theory of CTL model checking. We will then show that this 'SAT modulo CTL' solver can perform efficient and scalable CTL synthesis. We provide experimental comparisons to state-of-the-art techniques showing that this SMT-approach can find solutions to larger and more complex CTL formulas than comparable techniques, and does so without the limitations and extra expert knowledge that previous approaches require.

3 SAT Modulo Monotonic Theories for CTL

Bayless et al. [5] introduced techniques for building efficient SMT solvers for *Boolean monotonic theories* (SMMT), which are defined as follows:

Definition 1 (Boolean Monotonic Theory). *A theory T with signature Σ is Boolean monotonic if and only if:*

1. *The only sort in Σ is Boolean;*
2. *all predicates in Σ are monotonic; and*
3. *all functions in Σ are monotonic.*

A predicate P: $\{0,1\}^n \mapsto \{0,1\}$ is Boolean positive monotonic iff, for all i:

$$P(\dots, s_{i-1}, 0, s_{i+1}, \dots) \rightarrow P(\dots, s_{i-1}, 1, s_{i+1}, \dots)$$

A predicate P: $\{0,1\}^n \mapsto \{0,1\}$ is Boolean negative monotonic iff, for all i:

$$P(\dots, s_{i-1}, 1, s_{i+1}, \dots) \rightarrow P(\dots, s_{i-1}, 0, s_{i+1}, \dots)$$

The definition of monotonicity for a function F: $\{0,1\}^n \mapsto \mathcal{P}(S)$ (for some set S) is the same as above, but with "$\subseteq$" instead of "$\rightarrow$".

Theories operating over only Booleans are atypical in the SMT literature, and would appear at first glance to be highly restrictive. However, [5] showed that many common graph properties, such as reachability and maximum flow, can be expressed as Boolean monotonic theories, and that the resulting SMT solver (implemented in the lazy SMT solver MonoSAT) performs well in practice. Subsequently, MonoSAT has been extended to support theories of finite state machines, bit-vectors, and additional graph properties including acyclicity and connected component counts.

To see how [5] uses Boolean monotonic theories, consider the theory of graph reachability as an example. In that theory, a set of Boolean atoms determine which edges are included (*enabled*) in a finite graph. Reachability over such a graph is monotonic with respect to those edge atoms: given a graph in which node a reaches node b, a must still reach b after adding additional edges to the graph. One challenge of implementing lazy SMT solvers is that efficient solvers typically include theory propagation procedures that make deductions from partial assignments. However, because reachability is Boolean monotonic, two concrete graphs are sufficient to capture the space of possible graphs under a partial assignment: G_{under}, containing only edges that are enabled by the partial assignment, and G_{over}, in which additionally all unassigned edges are enabled. If a reachability predicate does not hold in G_{over}, then it can safely be deduced that it does not hold in any extension of the partial assignment. Similarly, if it holds in G_{under}, then it holds in all extensions of the partial assignment. These facts are used by MonoSAT to implement efficient theory propagation.

Below, we show that MonoSAT can be extended to support a theory of CTL model checking, allowing MonoSAT to express predicates of the form $Model_{\phi,K}(T, A)$, where ϕ is a CTL formula over atomic propositions P, and K is a Kripke structure with a fixed set S of states, T is a vector of $|S|^2$ Booleans controlling which transitions are in K, and A is a vector of $|S||P|$ Booleans controlling which atomic propositions hold in each state. $Model_{\phi,K}(T, A)$ is True if and only if the Kripke structure K is a model for ϕ under assignment to these transition and state property variables.

However, we face an immediate challenge: CTL model checking is neither monotonic with respect to the set T of transitions in the Kripke structure, nor with respect to the set A of property assignments in each state. Consider, for example, a two state Kripke structure with transitions between both states. $\phi = (\texttt{EF}\ a \wedge \neg(\texttt{AG}\ a))$ evaluates to False if atomic proposition a is in neither state, evaluates to True if a is in one, but not the other state, and evaluates to False if a is in both states (a similar argument can be made for the non-monotonicity of $Model_{\phi,K}(T, A)$ with respect to T).

Our solution begins with the observation that each individual CTL operator, considered on its own in a non-nested formula, is monotonic. We will use this observation to construct an alternative predicate, $ModelApprox_{\phi,K}(T_1, A_1, T_2, A_2)$, over two separate assignments of transitions and states to K. Unlike $Model_{\phi,K}$, $ModelApprox_{\phi,K}$ is Boolean monotonic, and we will show that it can be used either to safely over-approximate the semantics of CTL, or to safely under-

approximate them. By combining this new monotonic predicate with additional constraints on its arguments, we will then recover the semantics required to support our original CTL model checking predicate $Model_{\phi,K}(T, A)$.

3.1 A monotonic approximation of CTL

Below, we restrict our attention to the existentially quantified CTL operators `EX`, `EG` and `EU`, along with propositional operators ($\neg, \wedge, \vee$), as well as True and False, which are well known to form an adequate set. Any CTL formula can be efficiently converted into a logically equivalent existential normal form (ENF) in terms of these operators, linear in the size of the original formula [16].

First, we show that CTL formulas consisting of a single operator `EX` p, `EG` p or p `EU` q have each, individually, a Boolean positive monotonic satisfiability predicate (where p, q are atomic propositions). We let $solve_{s,\phi}(T, A)$ be the predicate that denotes whether or not the formula ϕ holds in the initial state s of the Kripke structure determined by the vector of Booleans T (transitions) and A (state properties).

Lemma 1. *$solve_{s,\phi}(T, A)$ is positive Boolean monotonic if ϕ is one of `EX` p, `EG` p or p `EU` q.*

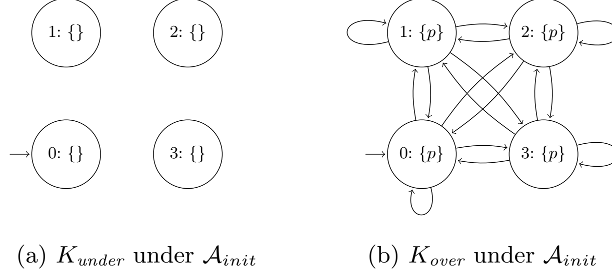
Proof. Take any T, A that determine a structure K for which the predicate holds. Let K' be a structure determined by some T', A' such that K' has the same states, state properties and transitions as K, except for one transition that is enabled in K' but not in K, or one state property which holds in K' but not in K. Formally, there is exactly one argument in either T' or A' that is 0 in T (or A respectively) and 1 in T' (or A' respectively). Then either (a) one of the states satisfies one of the atomic propositions in K', but not in K, or (b) there is a transition in K', but not in K.

We assume $solve_{s,\phi}(T, A)$ holds. Then, there must exist a witnessing infinite sequence starting from s in K. If (b), the exact same sequence must exist in K', since it has a superset of the transitions in K. Thus we can conclude $solve_{s,\phi}(T', A')$ holds. If (a), then the sequence will only differ in at most one state, where p holds instead of $\neg p$ (or q instead of $\neg q$). We note that for each of the three CTL operators, this sequence will be a witness for K', if the original sequence was a witness for K. Thus, $solve_{s,\phi}(T', A')$ holds as well.

It is easy to see that $\wedge$ and $\vee$ are positive monotonic in the same way, and $\neg$ is negative monotonic. Excluding negation, then, all the CTL operators needed to express formulas in ENF have positive Boolean monotonic *solve* predicates, while negation alone has a negative Boolean monotonic *solve* predicate.

Until now, we have considered the model checking algorithm to compute a predicate that returns True iff the initial state of the Kripke structure satisfies the formula. This can be extended to a function $solve(\phi, K)$ that evaluates the truth value of ϕ for each state in the Kripke structure, and returns a bit vector representing a set of states, that for each state is 1 iff that state satisfies ϕ. The

Example: Initially, the SAT solver's assignment $\mathcal{A}_{init} = \emptyset$ to transitions and state properties is empty, which determines K_{under} and K_{over} in the following way.

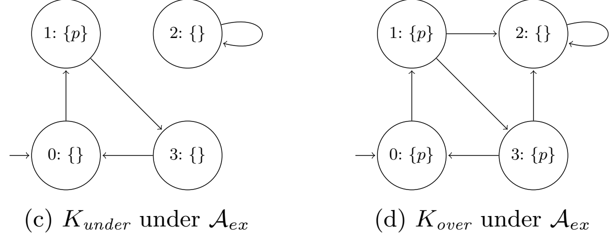


(a) K_{under} under $\mathcal{A}_{init}$ (b) K_{over} under $\mathcal{A}_{init}$

As the SAT solver makes assignments to theory atoms *trans*(...) and *sat*(...) (positive assignments affect K_{under}'s T_2, A_2, negative K_{over}'s T_1, A_1), both structures converge. Take for instance this partial assignment $\mathcal{A}_{ex}$:

$$\begin{aligned}\mathcal{A}_{ex} = & \mathit{sat}(1,p) \wedge \neg \mathit{sat}(2,p) \wedge \mathit{trans}(0,1) \wedge \mathit{trans}(1,3) \wedge \mathit{trans}(3,0) \wedge \mathit{trans}(2,2) \\ & \wedge \neg \mathit{trans}(0,0) \wedge \neg \mathit{trans}(0,2) \wedge \neg \mathit{trans}(0,3) \wedge \neg \mathit{trans}(1,0) \wedge \neg \mathit{trans}(1,1) \\ & \wedge \neg \mathit{trans}(2,0) \wedge \neg \mathit{trans}(2,1) \wedge \neg \mathit{trans}(2,3) \wedge \neg \mathit{trans}(3,1) \wedge \neg \mathit{trans}(3,3)\end{aligned}$$

The atoms $\mathit{sat}(0,p)$, $\mathit{sat}(3,p)$, $\mathit{trans}(1,2)$, and $\mathit{trans}(3,2)$ are unassigned by $\mathcal{A}_{ex}$. $\mathcal{A}_{ex}$ determines K_{under} and K_{over} in the following way:



(c) K_{under} under $\mathcal{A}_{ex}$ (d) K_{over} under $\mathcal{A}_{ex}$

$\mathit{solveApprox}(\phi, K_{over}, K_{under})$ returns an over-approximation (resp. under-approx with K_{over} and K_{under} exchanged) of the set of states in which ϕ may hold in extensions of the partial assignment. Assume K_{over} and K_{under} are obtained from $\mathcal{A}_{ex}$ and $\phi = \texttt{EX}\ \neg p$:

$$\begin{aligned}& \mathit{solveApprox}(\texttt{EX}\ \neg p, K_{over}, K_{under}) \\ & = \mathit{solve}_{EX}(\mathit{solveApprox}(\neg p, K_{over}, K_{under}), K_{over}) \\ & = \mathit{solve}_{EX}(\mathit{solve}_{\neg}(\mathit{solveApprox}(p, K_{under}, K_{over}), K_{under}), K_{over}) \\ & = \mathit{solve}_{EX}(\mathit{solve}_{\neg}(\{1\}, K_{under}), K_{over}) \\ & = \mathit{solve}_{EX}(\{0,2,3\}, K_{over}) \\ & = \{1,2,3\}\end{aligned}$$

The initial state is not in the over-approximation of states, where $\texttt{EX}\ \neg p$ holds ($0 \notin \mathit{solveApprox}(\texttt{EX}\ \neg p, K_{over}, K_{under})$), therefore $\texttt{EX}\ \neg p$ *does not* hold in *any* Kripke structure obtained from a full extension of the partial assignment $\mathcal{A}_{ex}$.

Fig. 1: Example of a partial assignment $\mathcal{A}_{ex}$ determining to K_{under} and K_{over}, and the evaluation *solveApprox* of a formula on K_{under} and K_{over}.

monotonicity properties above also hold for $solve(\phi, K)$, as every state in the bitset can be viewed as an initial state for which the operators are monotonic.

We introduce for each CTL operator *op* an evaluation function $solve_{op}(X, K)$ that evaluates the operator on a set of states X, instead of a subformula. This is a standard interpretation of CTL (and how CTL model checking is often implemented), and we refer to the literature for common ways to compute $solve_{op}$ for each operator. Our function $solve(\phi, K)$ takes the top-most operator *op* of ϕ: if it is an atomic proposition, it returns the set of states in which the atomic proposition holds, otherwise it solves its argument(s) recursively and then applies $solve_{op}$ on the returned set of states. One can think of the set X as defining the states in which a fresh atomic proposition holds, and of $solve_{op}(X, K)$ as computing the application of *op* on that atomic proposition.

Algorithm 1: solveApprox(ϕ, K_{over}, K_{under})

```
if φ is an atomic proposition then
    return set of states satisfying φ in K_over
else if φ is a unary operator op with argument ψ then
    if op is ¬ then                                  // negative monotonic
        X := solveApprox(ψ, K_under, K_over)
        return solve_op(X, K_under)
    else                                             // op ∈ {EX, EG}
        X := solveApprox(ψ, K_over, K_under)
        return solve_op(X, K_over)
else            // φ is binary op ∈ {EU, ∧, ∨} with arguments ψ1, ψ2
    X1 := solveApprox(ψ1, K_over, K_under)
    X2 := solveApprox(ψ2, K_over, K_under)
    return solve_op(X1, X2, K_over)
```

Algorithm 1, *solveApprox*(ϕ, K_{over}, K_{under}) takes a CTL formula ϕ and two Kripke structures, K_{over} and K_{under}. It returns a bit vector, representing a set of states.[4] We will show in Lemma 2 that for appropriate values of K_{over} and K_{under}, *solveApprox* computes a safe over-approximation of $solve(\phi, K)$ for a third Kripke structure, K: $solve(\phi, K) \subseteq solveApprox(\phi, K_{over}, K_{under})$. Further, as K_{under} and K_{over} converge, so do $solveApprox(\phi, K_{over}, K_{under})$ and $solve(\phi, K)$. If $K_{under} = K_{over} = K$, then $solveApprox(\phi, K_{over}, K_{under}) = solve(\phi, K)$. This follows directly from Lemma 2.

In order for the over-approximation property of *solveApprox* given in the following lemma to hold, K_{over} (determined by some T_1, A_1), K_{under} (determined by some T_2, A_2) and K (determined by some T, A) must be Kripke structures with the same number of states, and K_{over} must have a superset, and K_{under} a subset, of the transitions and state properties of K: $T_2 \subseteq T \subseteq T_1$ and $A_2 \subseteq A \subseteq A_1$. To illustrate how this will be used in the context of SMT, Fig. 1 shows

[4] A similar algorithm for evaluating CTL formulas on 'partial Kripke structures', in the context of model checking, can be found in [6].

an example of how the SAT solver's partial assignment determines K_{over} and K_{under}, and how *solveApprox* works on these structures.

Lemma 2. *$solve(\phi, K) \subseteq$ solveApprox(ϕ, K_{over}, K_{under}) and $solve(\phi, K) \supseteq$ solveApprox(ϕ, K_{under}, K_{over}).*

Proof. By induction over ϕ. If ϕ is an atomic proposition, then *solveApprox* returns the set of states satisfying ϕ in K_{over}. $solve(\phi, K)$ will return the set of states satisfying ϕ in K. The first claim holds, since $A \subseteq A_1$.

If $\phi = op\ \psi$ with *op* a unary positive monotonic operator, $solve(\phi, K)$ is $solve_{op}(X, K)$ for $X = solve(\psi, K) \subseteq_{IH}$ *solveApprox*$(\psi,\ K_{over},\ K_{under}) = X'$. *solveApprox*$(\phi,\ K_{over},\ K_{under})$ is $solve_{op}(X', K)$. The first claim holds, since $X \subseteq X'$ and $solve_{op}$ is positive monotonic. If *op* is unary negative monotonic, *i.e.* $\neg$, then $solve(\phi, K)$ is $solve_{op}(X, K)$ for $X = solve(\psi, K) \supseteq_{IH}$ *solveApprox*$(\psi,$ $K_{under},\ K_{over}) = X'$. *solveApprox*$(\phi,\ K_{over},\ K_{under})$ is $solve_{op}(X', K)$. The first claim holds, since $X \supseteq X'$ and $solve_{op}$ is negative monotonic.

The proof obligations for $solve(\phi, K) \supseteq$ *solveApprox*$(\phi,\ K_{under},\ K_{over})$ are left out here, as well as the proof obligations for positive monotonic binary operators. The proof for these proceeds similarly to the above cases.

3.2 CTL as a Boolean monotonic predicate

solveApprox$(\phi,\ K_{over},\ K_{under})$ computes an over-approximation (*resp.*, with K_{over} and K_{under} exchanged, an under-approximation) of the set of states in which a CTL formula ϕ holds in Kripke structure K, so long as K_{over} and K_{under} are, as defined above, structures that are over-, and respectively under-approximating K. We construct a corresponding Boolean monotonic predicate $ModelApprox_{\phi,K}(T_1, A_1, T_2, A_2)$ which holds iff the initial state $s_0 \in$ *solveApprox*$(\phi,\ K_{over},\ K_{under})$.[5] Its monotonicity follows from the following lemma:

Lemma 3. *solveApprox(ϕ, K_{over}, K_{under}) is a function positive monotonic in K_{over} and negative monotonic in K_{under}.*

Proof. By structural induction over ϕ. If ϕ is an atomic proposition, then *solveApprox* returns the set of states satisfying ϕ in K_{over}. If a state or transition is added to K_{over} (call the resulting structure ${K_{over}}'$), then *solveApprox*$(\phi,\ K_{over},\ K_{under})$ $\subseteq$ *solveApprox*$(\phi,\ {K_{over}}',\ K_{under})$. If a state or transition is removed from K_{under} (resulting in ${K_{under}}'$), then *solveApprox*$(\phi,\ K_{over},\ K_{under}) =$ *solveApprox*$(\phi,$ $K_{over},\ {K_{under}}')$.

Assume $\phi = op\ \psi$ with *op* a unary positive monotonic operator. Then *solveApprox*$(\phi,\ K_{over},\ K_{under})$ is the function composition of positive monotonic $solve_{op}$ and *solveApprox*$(\psi,\ K_{over},\ K_{under})$, which is positive monotonic in K_{over} and negative monotonic in K_{under} by the induction hypothesis. The composed function is then also positive monotonic in K_{over} and negative in K_{under}.

[5] If $T_2 \not\subseteq T_1$ or $A_2 \not\subseteq A_1$, *ModelApprox* can be defined to evaluate in any arbitrary way that maintains monotonicity. As discussed below, we exclude this case in our implementation, by enforcing $T_1 = T_2$ and $A_1 = A_2$.

Assume on the other hand that *op* is a unary negative monotonic operator, *i.e.* $\neg$. Then $solveApprox(\phi, K_{over}, K_{under})$ is the function composition of $solve_{\neg}$ and $solveApprox(\psi, K_{under}, K_{over})$, which is assumed by the induction hypothesis to be positive monotonic in K_{under}, and negative monotonic in K_{over}. Since $solve_{\neg}$ is negative monotonic in its first argument (and ignores its second argument), the composed function is positive monotonic in K_{over}, and negative in K_{under}.

The proof obligations for binary operators (all positive monotonic) are left out here. The proof for these proceeds similarly to the above cases.

Corollary 1. *$ModelApprox_{\phi,K}(T_1, A_1, T_2, A_2)$ is positive monotonic in T_1, A_1 and negative monotonic in T_2, A_2.*

Proof. By definition, $ModelApprox_{\phi,K}(T_1, A_1, T_2, A_2)$ holds if, and only if, $s_0 \in solveApprox(\phi, K_{over}, K_{under})$; therefore the monotonicity of $ModelApprox$ follows directly from the monotonicity of *solveApprox* (Lemma 3).

We complete our theory of CTL model checking by forcing $T_1 = T_2$ and $A_1 = A_2$. As we proved above, $ModelApprox_{\phi,K}(T, A, T, A) = Model_{\phi,K}(T, A)$, and so in this way we recover the expected definition of CTL model checking in our theory solver. The equalities $T_1 = T_2$ and $A_1 = A_2$ could be enforced by adding a linear number of additional Boolean constraints to the SAT solver; in our implementation we found it more efficient to enforce this equality internally in the theory solver.

4 Implementation and Optimizations

Above, we showed how CTL model checking can be posed as a Boolean monotonic theory. We then built a lazy SMT theory solver, following the theory propagation techniques for Boolean monotonic theories described in [5]. We have also implemented some additional optimizations which greatly improve the performance of our CTL theory solver. One basic optimization that we implement is pure literal filtering (see, *e.g.,* [18]): For the case where $Model_{\phi,K}(T, A)$ is assigned True (resp. False), we only need to check whether $Model_{\phi,K}(T, A)$ is falsified (resp., made true) during theory propagation. In all of the instances we will examine in this paper, $Model_{\phi,K}(T, A)$ is assigned True in the input formula, and so this optimization greatly simplifies theory propagation. We discuss several further improvements below:

In Section 4.1 we outline how our solver performs clause learning. In Section 4.2 we describe symmetry breaking constraints, which can greatly reduce the search space of the solver, and in Section 4.3 we show how several common types of CTL constraints can be cheaply converted into CNF, reducing the size of the formula the theory solver must handle. Finally, in Section 4.4, we discuss how in the common case of a CTL formula describing multiple communicating processes we can (optionally) add support for additional *structural constraints*, similarly to the approach described in [10]. These structural constraints allow our solver even greater scalability, at the cost of adding more states into the

smallest solution that can be synthesized. Thus, if structural constraints are used, iteratively decreasing the bound may no longer yield a minimal structure.

4.1 Clause Learning

Supporting efficient clause learning (also called "justification set" or "conflict set" learning in the SMT literature) is a critically important function of lazy SMT theory solvers. Theory solvers can always return a naive conflict set consisting of the entire conflicting (partial) assignment, however, efficient theory solvers typically implement clause learning procedures which attempt to find smaller, or sometimes even minimal, conflict sets.

Unlike our theory propagation implementation, which operates on formulas in existential normal form, to perform clause learning we convert the CTL formula into *negation normal form* (NNF), pushing any negation operators down to the innermost terms of the formula. To obtain an adequate set, the formula may now also include universally quantified CTL operators and Weak Until. Each of these operators is handled separately.

Our procedure $learn(\phi, s)$ operates recursively on the NNF of the formula and returns a conflict set of literals, the disjunction of which yields a CNF clause which is learned by the SAT solver. The same conflict set is populated on every level of the recursion, *i.e.* the learned literals have an additive effect on the conflict clause. For instance, if the formula $\texttt{EX}\ \phi$ is in conflict with the partial assignment, we first consider the operator `EX`. Our clause learning strategy for `EX` in the state s (in this case, the initial state) is to force the SAT solver to enable any disabled transitions from s, or to make ϕ true in any of the successor states. Literals for the latter are computed recursively via $learn(\phi, t)$ for every enabled transition (s, t). $learn(\phi, s)$ is defined as follows (for notation see Fig. 2):

- $learn(p, s)$ (resp. $learn(\neg p, s)$), where p is an atomic proposition. Add the literal $disableAPinState(p, s)$ (resp. $enableAPinState(p, s)$) to the conflict set.
- $learn(op\ \psi, s)$ (resp. $learn(\psi_1\ op\ \psi_2, s)$): Add the literals returned by the functions $learn_{op}(\psi, s)$ (resp. $learn_{op}(\psi_1, \psi_2, s)$) to the conflict set (see Fig. 2).

4.2 Symmetry Breaking

Due to the way we expose atomic propositions and transitions to the SAT solver with theory atoms, the SAT solver may end up exploring large numbers of isomorphic Kripke structures. We address this by enforcing extra symmetry-breaking constraints which prevent the solver from considering (some) redundant configurations of the Kripke structure. Symmetry reduction is especially helpful to prove instances UNSAT, which aids the search for suitable bounds.

Let $label(s_i)$ be the binary representation of the atomic propositions of state s_i, and let $out(s_i)$ be the set of outgoing edges of state s_i. Let s_0 be the initial state. The following constraint enforces an order on the allowable assignments of state properties and transitions in the Kripke structure.

$$\forall i, j : [i < j \wedge i \neq 0 \wedge j \neq 0] \rightarrow$$
$$[label(s_i) \leq label(s_j)\ \wedge\ (label(s_i) = label(s_j) \rightarrow |out(s_i)| \leq |out(s_j)|)]$$

Clause Learning functions $\boldsymbol{learn_{op}}$

$learn_{EX}(\phi, s)$: Let N be the neighbors of s that do not satisfy ϕ. Let D be the set of disabled transitions from s. Add $enableTransitionSet(D)$ to the conflict set, and $learn(\phi, n)$ for each $n \in N$.

$learn_{AX}(\phi, s)$: Let n be a neighbor of s that does not satisfy ϕ. Add $disableTransition(s, n)$ to the conflict set, and $learn(\phi, n)$.

$learn_{EF}(\phi, s)$: Let R be all states reachable from s. Let D be the set of disabled transitions from reachable states R to unreachable states. Add $enableTransitionSet(D)$ to the conflict set, and add $learn(\phi, r)$ for each $r \in R$.

$learn_{AF}(\phi, s)$: Let R be a set of states satisfying $\neg\phi$ that form lasso from s. Let D be the set of transitions in the lasso. Add $disableTransitionSet(D)$ to the conflict set, and add $learn(\phi, r)$ for each $r \in R$.

$learn_{EG}(\phi, s)$: Let R be the the states reachable from s via a path on which all states satisfy ϕ. Let N be the set of successor states of R which do not satisfy ϕ. Let D be the set of disabled transitions leaving R. Add $enableTransitionSet(D)$ to the conflict set, and add $learn(\phi, n)$ for each $n \in N$.

$learn_{AG}(\phi, s)$: Find a path of transitions D from s to a state r that doesn't satisfy ϕ. Add $disableTransitionSet(D)$ to the conflict set, and add $learn(\phi, r)$.

$learn_{\wedge}(\phi, \psi, s)$: If ϕ does not hold in the over-approximation (but ψ does), add $learn(\phi, s)$. If ψ does not hold in the over-approximation (but ϕ does), add $learn(\psi, s)$. If both do not hold, then construct a temporary conflict set for each, and add the smaller set to the conflict.

$learn_{\vee}(\phi, \psi, s)$: Add $learn(\phi, s)$ and $learn(\psi, s)$ to the conflict.

$learn_{EW}(\phi, \psi, s)$: Let R be the states satisfying ϕ and reachable via ϕ-satisfying states from s. Let D be the set of disabled transitions from states in R. Let P be the set of successors of R that are not in R. Add D to the conflict set, add $learn(\psi, r)$ for each $r \in R$, and add $learn((\phi \vee \psi), p)$ for each $p \in P$.

$learn_{AW}(\phi, \psi, s)$: Find a path starting from s such that all except the last state satisfy ϕ, and no state satisfies ψ. Let D be the set of transitions on that path; add $disableTransitionSet(D)$ to the conflict set. Let R be the set of states on that path, except for the last state of the path, n. Add $learn(\phi, n)$ and $learn(\psi, r)$ for each $r \in R$ to the conflict set.

$learn_{EU}(\phi, \psi, s)$: Same as $learn_{EW}$, but D is restricted to transitions to states outside of R.

$learn_{AU}(\phi, \psi, s)$: If there exists a finite path starting from s such that all except the last states on the path satisfy ϕ, add $learn_{AW}(\phi, \psi, s)$ to the conflict set. Else, add $learn_{AF}(\psi, s)$ to the conflict set.

Notation: $enableAPinState(p, s)$ returns the literal that assigns property p to state s. $enableTransition(s, t)$ returns the literal for transition $s \rightarrow t$. *enableTransitionSet(D)* returns $\{enableTransition(s,t) \mid (s, t) \in D\}$. *disableAPinState, disableTransition, disableTransitionSet* return corresponding negated literals. For existentially quantified CTL operators, transitions of K_{over} are used, for universally quantified CTL operators transitions of K_{under}. $solveApprox(\phi, K_{over}, K_{under})$ gives the set of states satisfying a subformula ϕ.
Notice that some functions (*e.g.,* $learn_{AG}$) are only correct if every state has an infinite path (`AG EX` TRUE), which is why we enforce this property in our solver.

Fig. 2: Clause learning functions returning sets of literals.

4.3 Preprocessing

Given a CTL specification ϕ, we identify certain common sub-expressions which can be cheaply converted directly into CNF, which is efficiently handled by the SAT solver at the core of MonoSAT. We do so if ϕ matches $\bigwedge_i \phi_i$, as is commonly the case when multiple properties are part of the specification. If ϕ_i is purely propositional, or of the form $\mathtt{AG}\ p$ with p purely propositional, we eliminate ϕ_i from the formula and convert ϕ_i into a logically equivalent CNF expression over the state property assignment atoms of the theory.[6] This requires a linear number of clauses in the number of states in K. We also convert formulas of the form $\mathtt{AG}\ \psi$, with ψ containing only propositional logic and at most a single Next-operator (`EX` or `AX`). Both of these are very common sub-expressions in the CTL formulas that we have examined.

4.4 Wildcard Encoding for Concurrent Programs

As will be further explained later, the synthesis problem for synchronization skeletons assumes a given number of processes, which each have a local transition system. The state transitions in the full Kripke structure then represent the possible interleavings of executing the local transition system of each process. This local transition system is normally encoded into the CTL specification.

Both [3] and [10] explored strategies to take advantage of the case where the local transition systems of these processes are made explicit. [10] were able to greatly improve the scalability of their answer-set-programming based CTL synthesis procedure by deriving additional 'structural' constraints for such concurrent processes. As our approach is also constraint-based, we can (optionally) support similar structural constraints. In experiments below, we show that even though our approach already scales better than existing approaches without these additional structural constraints, we also benefit from such constraints.

Firstly, we can exclude any global states with state properties that are an illegal encoding of multiple processes. If the local state of each process is identified by a unique atomic proposition, then we can enforce that each global state must make true exactly one of the atomic propositions for each process. For every remaining combination of state property assignments, excluding those determined to be illegal above, we add a single state into the Kripke structure, with a predetermined assignment of atomic propositions, such that only the transitions between these states are free for the SAT solver to assign. This is in contrast to the normal synthesis method, in which states are completely undetermined (but typically fewer are required).

Secondly, since we are interested in interleavings of concurrent programs, on each transition in the global Kripke structure we enforce that only a single process may change its local state, and it may change its local state only in a way that is consistent with the its local transition system.

---

[6] Since $\mathtt{AG}\ p$ only specifies reachable states, the clause is for each state s a disjunction of p being satisfied in s, or s having no enabled incoming transitions. This changes the semantics of CTL for unreachable states, but not for reachable states.

The above two constraints greatly reduce the space of transitions in the global Kripke structure that are left free for the SAT solver to assign (and completely eliminate the space of atomic propositions to assign in each state). However these constraints make our procedure incomplete, since in general more than a single state with the same atomic propositions (but different behavior) need to be distinguished. To allow multiple states with equivalent atomic propositions, we also add a small number of 'wildcard' states into the Kripke structure, whose state properties and transitions (incoming and outgoing) are not set in advance. In the examples we consider in this paper, we have found that a small number of such wildcard states (between 3 and 20) are sufficient to allow for a Kripke structure that satisfies the CTL formula, while still greatly restricting the total space of Kripke structures that must be explored by the SAT solver.

We disable symmetry breaking when using the wildcard encoding, as the wildcard encoding is incompatible with the constraint in Section 4.2.

5 Experimental Results

There are few CTL synthesis implementations available for comparison. Indeed, the original CTL synthesis/model-checking paper [8] presents an implementation of CTL model checking, but the synthesis examples were simulated by hand. The only publicly available, unbounded CTL synthesis tool we could find is Prezza's open-source CTLSAT tool[7], which is a modern implementation of the classic tableau-based CTL synthesis algorithm [8].

We also compare to De Angelis et al.'s encoding of bounded CTL synthesis into ASP [10]. De Angelis et al. provide encodings[8] specific to the n-process mutual exclusion example, which exploit structural assumptions about the synthesized model (for example, that it is the composition of n identical processes). We label this encoding "ASP-structural" in the tables below. For ASP-structural, we have only the instances originally considered in [10].

To handle the general version of CTL synthesis (without added structural information), we also created ASP encodings using the methods from De Angelis et al.'s paper, but without problem-specific structural assumptions and optimizations. We label those results "ASP-generic". For both encodings, we use the latest version (4.5.4) of Clingo [13], and for each instance we report the best performance over the included Clasp configurations.[9]

We compare these tools to two versions of MonoSAT: MonoSAT-structural, which uses the wildcard optimization presented in Section 4.4, and MonoSAT-generic, without the wildcard optimization.

With the exception of CTLSAT, the tools we consider are bounded synthesis tools, which take as input both a CTL formula and a maximum number of states. For ASP-structural, the state bounds follow [10]. For the remaining tools, we selected the state bound manually, by repeatedly testing each tool with different

[7] https://github.com/nicolaprezza/CTLSAT
[8] http://www.sci.unich.it/~deangelis/papers/mutex_FI.tar.gz
[9] These are: "auto", "crafty", "frumpy", "handy", "jumpy", "trendy", and "tweety".

bounds, and reporting for each tool the smallest bound for which it found a satisfying solution. In cases where a tool could not find any satisfying solution within our time or memory bounds, we report out-of-time or out-of-memory.

5.1 The Original Clarke-Emerson Mutex

The mutex problem assumes that there are n processes that run concurrently and on occasion access a single shared resource. Instead of synthesizing entire programs, the original Clarke-Emerson example [8] considers an abstraction of the programs called *synchronization skeletons*. In the instance of a mutex algorithm, it is assumed that each process is in one of three states: *non-critical section* (**NCS**), the *try section* (**TRY**) or the *critical section* (**CS**). A process starts in the non-critical section in which it remains until it requests to access the resource, and changes to the try section. When it finally enters the critical section it has access to the resource, and eventually loops back to the non-critical section. The synthesis problem is to find a global Kripke structure for the composition of the n processes, such that the specifications are met. Our first set of benchmarks are based on the Clarke and Emerson specification given in [8], that includes mutual exclusion and starvation freedom for all processes.

Approach	# of Processes				
	2	**3**	**4**	**5**	**6**
CTLSAT	TO	TO	TO	TO	TO
ASP-generic	3.6 (7*)	1263.7 (14)	TO	MEM	MEM
ASP-structural	0.0 (12)	1.2 (36)	-	-	-
MonoSAT-generic	0.0 (7*)	1.4 (13*)	438.6 (23*)	1744.9 (42)	TO
MonoSAT-struct	0.2 (7)	0.5 (13)	4.5 (23)	166.7 (41)	1190.5 (75)

Table 1: Results on the Original Clarke-Emerson Mutual Exclusion Example. Table entries are in the format *time(states)*, where *states* is the number of states in the synthesized model, and *time* is the run time in seconds. For ASP-structural, we only have the manually encoded instances provided by the authors. An asterisk indicates that the tool was able to prove minimality, by proving the instance is UNSAT at the next lower bound. TO denotes exceeding the 3hr timeout. MEM denotes exceeding 16GB of RAM. All experiments were run on a 2.67GHz Intel Xeon x5650 processor.

Results Table 1 presents our results on the mutex formulation from [8]. Both versions of MonoSAT scale to much larger instances than the other approaches, finding solutions for 5 and 6 processes, respectively. CTLSAT, implementing the classical tableau approach, times out on all instances.[10] Only the -generic

[10] Notably, CTLSAT times-out even when synthesizing the original 2-process mutex from [8], which Clarke and Emerson originally synthesized by hand. This may be because in that work, the local transition system was specified implicitly in the algorithm, instead of in the CTL specification as it is here.

versions can guarantee minimal solutions, and MonoSAT-generic is able to prove minimal models for several cases.

As expected, structural constraints greatly improve efficiency for both ASP-structural and MonoSAT-structural relative to their generic counterparts.

5.2 Mutex with Additional Properties

Approach	# of Processes 2	3	4	5	6	7
Property: ORIG ∧ BO						
ASP-generic	3.4 (7*)	1442.0 (14)	TO/MEM	MEM	MEM	MEM
ASP-structural	0.0 (12)	2.3 (36)	-	-	-	-
MonoSAT-gen	0.0 (7*)	11.1 (13*)	438.3 (23*)	1286.6 (42)	TO	TO
MonoSAT-str	0.1 (7)	0.6 (13)	5.3 (23)	59.5 (41)	375.3 (75)	10739.5 (141)
Property: ORIG ∧ BO ∧ MR						
ASP-generic	10.1 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.8 (10)	950.9 (27)	-	-	-	-
MonoSAT-gen	0.0 (9*)	6.0 (25*)	TO	TO	TO	TO
MonoSAT-str	0.1 (10)	8.7 (26)	TO	TO	TO	TO
Property: ORIG ∧ NB ∧ NJ						
ASP-generic	34.8 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.1 (10)	7326.1 (27)	-	-	-	-
MonoSAT-gen	0.0 (9*)	1275.7 (22*)	TO	TO	TO	TO
MonoSAT-str	0.2 (10)	1.6 (26)	5314.7 (51)	TO	TO	TO
Property: ORIG ∧ NB ∧ NJ ∧ BO						
ASP-generic	15.4 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.1 (10)	TO	-	-	-	-
MonoSAT-gen	0.0 (9*)	127.7 (22*)	TO	TO	TO	TO
MonoSAT-str	0.1 (10)	1.3 (24)	TO	TO	TO	TO
Property: ORIG ∧ NB ∧ NJ ∧ BO ∧ MR						
ASP-generic	10.7 (9*)	TO	MEM	MEM	MEM	MEM
ASP-structural	0.1 (10)	1917.6 (27)	-	-	-	-
MonoSAT-gen	0.0 (9*)	4.4 (25*)	TO	TO	TO	TO
MonoSAT-str	0.1 (10)	2.7 (26)	TO	TO	TO	TO

Table 2: Results on the Mutual Exclusion Example with Additional Properties (described in Section 5.2). As with Table 1, entries are in the format *time(states)*. ORIG denotes the original mutual exclusion properties from Section 5.1. As before, although problem-specific structural constraints improve efficiency, MonoSAT-generic is comparably fast to ASP-structural on small instances, and scales to larger numbers of processes. MonoSAT-structural performs even better.

As noted in [14], the original Clarke-Emerson specification permits Kripke structures that are not *maximally parallel*, or even practically reasonable. For instance, our methods synthesize a structure in which one process being in NCS

will block another process in TRY from getting the resource — the only transition such a global state has is to a state in which both processes are in the TRY section. In addition to the original formula, we present results for an augmented version in which we eliminate that solution[11] by introducing the "Non-Blocking" property, which states that a process may always remain in the NCS:

$$\texttt{AG}\ (\mathrm{NCS}_i \rightarrow\ \texttt{EX}\ \mathrm{NCS}_i) \qquad \text{(NB)}$$

In addition, in the original paper there are structural properties implicit in the given local transition system, preventing jumping from NCS to CS, or from CS to TRY. We encode these properties into CTL as "No Jump" properties.

$$\texttt{AG}\ (\mathrm{NCS}_i \rightarrow\ \texttt{AX}\ \neg\mathrm{CS}_i) \quad \wedge \quad \texttt{AG}\ (\mathrm{CS}_i \rightarrow\ \texttt{AX}\ \neg\mathrm{TRY}_i) \qquad \text{(NJ)}$$

We also consider two properties from [10]: Bounded Overtaking (BO), which guarantees that when a process is waiting for the critical section, each other process can only access the critical section at most once before the first process enters the critical section, and Maximal Reactivity (MR), which guarantees that if exactly one process is waiting for the critical section, then that process can enter the critical section in the next step.

Results We repeat our experimental procedure from Section 5.1, except with various combinations of additional properties. This provides a richer set of benchmarks, most of which are harder than the original.

Table 2 presents our results. As before, the -structural constraints greatly improve efficiency, but nevertheless, MonoSAT-generic outperforms ASP-structural. MonoSAT-generic is able to prove minimality on several benchmarks, and on one benchmark, MonoSAT-structural scales to 7 processes.

5.3 Readers-Writers

To provide even more benchmarks, we present instances of the related Readers-Writers problem [9]. Whereas the Mutex problem assumes that all processes require exclusive access to a resource, the Readers-Writers problem permits some simultaneous access. Two types of processes are distinguished: writers, which require exclusive access, and readers, which can share their access with other readers. This is a typical scenario for concurrent access to shared memory, in which write permissions and reading permissions are to be distinguished. The local states of each process are as in the Mutex instances.

We use Attie's [2] CTL specification. We note however that this specifition allows for models which are not maximally parallel, and in particular disallows

[11] While the properties that we introduce in this paper mitigate some of the effects of underspecification, we have observed that the formulas of many instances in our benchmarks are not strong enough to guarantee a sensible solution. We are mainly interested in establishing benchmarks for synthesis performance, which is orthogonal to the task of finding suitable CTL specifications, which resolve these problems.

concurrent access by two readers. In addition to this original formula, we also consider one augmented with the Multiple Readers Eventually Critical (MREC) property. This ensures that there is a way for all readers, if they are in TRY, to simultaneously enter the critical section, if no writer requests the resource.

$$\texttt{AG}\ (\bigwedge_{w_i} \mathrm{NCS}_{w_i} \rightarrow (\bigwedge_{r_i} \mathrm{TRY}_{r_i} \rightarrow\ \texttt{EF}\ \bigwedge_{r_i} CS_{r_i})) \qquad \text{(RW-MREC)}$$

This property turns out not to be strong enough to enforce that concurrent access for readers must always be possible. We introduce the following property, which we call Multiple Readers Critical. It states that if a reader is in TRY, and all other readers are in CS, it is possible to enter the CS in a next state – as long as all writers are in NCS, since they have priority access over readers.

$$\texttt{AG}\ (\bigwedge_{w_i} \mathrm{NCS}_{w_i} \rightarrow (\mathrm{TRY}_{r_i} \bigwedge_{r_j \neq r_i} \mathrm{CS}_{r_j} \rightarrow\ \texttt{EX}\ \bigwedge_{r_i} CS_{r_i})) \qquad \text{(RW-MRC)}$$

Using this property, we are able to synthesize a structure for two readers and a single writer, in which both readers can enter the critical section concurrently, independently of who enters it first, without blocking each other.

	# of Processes (# of readers, # of writers)					
Approach	**2 (1, 1)**	**3 (2, 1)**	**4 (2, 2)**	**5 (3, 2)**	**6 (3, 3)**	**7 (4, 3)**
Property: RW						
CTLSAT	TO	TO	TO	TO	TO	TO
ASP-generic	0.6 (5*)	9.5 (9*)	TO	MEM	MEM	MEM
MonoSAT-gen	0.0 (5*)	0.0 (9*)	2.8 (19*)	30.0 (35*)	5312.7 (74)	TO
MonoSAT-str	0.1 (5)	0.5 (9)	0.7 (19)	2.9 (35)	98.8 (74)	384.4 (142)
Property: RW ∧ NB ∧ NJ						
ASP-generic	6.8 (8*)	2865.5 (16)	MEM	MEM	MEM	MEM
MonoSAT-gen	0.0 (8*)	1.4 (16*)	110.4 (27*)	843.8 (46*)	TO	TO
MonoSAT-str	0.1 (9)	0.2 (16)	3.4 (27)	35.9 (54)		TO
Property: RW ∧ NB ∧ NJ ∧ RW-MREC						
ASP-generic	2.4 (8*)	120.6 (22)	MEM	MEM	MEM	MEM
MonoSAT-gen	0.0 (8*)	238.4 (22*)	TO	TO	TO	TO
MonoSAT-str	0.1 (9)	0.25 (23)	5.3 (52)	159.1 (127)	TO	TO
Property: RW ∧ NB ∧ NJ ∧ RW-MRC						
ASP-generic	2.4 (8*)	TO	MEM	MEM	MEM	MEM
MonoSAT-gen	0.0 (8*)	1114.1 (22)	18.1 (27*)	251.6 (46*)	TO	TO
MonoSAT-str	0.1 (9)	0.2 (23)	2.5 (28)	28.0 (47)	TO	TO

Table 3: Results on the Readers-Writers Instances. Property (RW) is Attie's specification [2]. Data is presented as in Table 1, in the format *time(states)*.

Results We run benchmarks on problem instances of various numbers of readers and writers, and various combinations of the CTL properties. ASP-structural has identical process constraints, which make it unsuitable to solve an asymmetric

problem such as Readers-Writers (we exclude it from these experiments). As with the Mutex problem, as CTLSAT is unable to solve even the simplest problem instances, we do not include benchmarks for the more complex instances.

Our experiments on each variation of the Readers-Writer problem are presented in Table 3. We observe that in general, Readers-Writers instances are easier to solve than Mutex instances with the same number of processes. At the same time, the additional properties introduced by us restrict the problem further, and make the instances harder to solve than the original Readers-Writers formulation. Taken together with the results from Tables 1 and 2, this comparison further strengthens our argument that MonoSAT-generic scales better than ASP-generic. The results also confirm that the structural MonoSAT solver making use of the wildcard encoding performs much better than MonoSAT-generic.

6 Conclusion and Future Work

We have demonstrated a novel approach to CTL synthesis that greatly outperforms existing tools, with the ability to flexibly add additional constraints (e.g., about the structure of the desired solution), and without sacrificing generality (by e.g., assuming identical processes). In many cases, we are also able to compute a provably minimal satisfying Kripke structure.

Our approach is based on formulating CTL model checking in terms of monotonic theories, enabling use of the SAT Modulo Monotonic Theories (SMMT) approach to build an efficient, lazy SAT Modulo CTL solver. This success reinforces the claim that monotonic theories, and more generally the lazy SMT approach, are a performant and versatile basis for SMT solvers.

There are many directions for future work. Although we have not tested this yet, MonoSAT has support for optimization constraints, which might allow one to synthesize *maximally parallel solutions*, as described in [14]. At the implementation level, we have many ideas for improving performance and scalability. We have expended little effort to optimize the CTL model checker at the heart of the theory solver. With improved performance, more applications may be feasible. For example, we believe our solver is suitable for the *repair problem* [1], because we can easily specify constraints of the existing system, repair possibilities, and the specification of correctness. Another promising approach to scalability is to leverage techniques like Attie and Emerson's [3], which rely on synthesizing small 2-process Kripke structures and generalizing them to vast networks of similar processes; using our techniques in conjunction with theirs should allow much more realistic complexity in the pairwise synthesized programs. In a more theoretical direction, we have implemented preliminary support for fairness constraints. If this proves robust and scalable, it may open the door toward synthesis of more expressive temporal logics.

7 Acknowledgments

This work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada. We also thank Javier Esparza for his encouragement and helpful advice.

References

1. P. Attie, A. Cherri, K. Dak Al Bab, M. Sakr, and J. Saklawi. Model and program repair via SAT solving. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 148–157. ACM/IEEE, 2015.
2. P. C. Attie. Synthesis of large concurrent programs via pairwise composition. In *CONCUR99 Concurrency Theory*, pages 130–145. Springer, 1999.
3. P. C. Attie and E. A. Emerson. Synthesis of concurrent systems with many similar processes. *ACM Trans. Prog. Lang. Sys. (TOPLAS)*, 20(1):51–115, 1998.
4. P. C. Attie and E. A. Emerson. Synthesis of concurrent programs for an atomic read/write model of computation. *ACM Trans. Prog. Lang. Sys. (TOPLAS)*, 23(2):187–242, 2001.
5. S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.
6. G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Computer Aided Verification*. Springer, 1999.
7. D. Bustan and O. Grumberg. Simulation-based minimization. *ACM Trans. Comput. Logic*, 4(2):181–206, Apr. 2003.
8. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs*, 1982.
9. P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with readers and writers. *Commun. ACM*, 14(10):667–668, Oct. 1971.
10. E. De Angelis, A. Pettorossi, and M. Proietti. Synthesizing concurrent programs using answer set programming. *Fundamenta Informaticae*, 120(3-4):205–229, 2012.
11. L. de Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*. Springer, 2009.
12. E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Symposium on Theory of Computing*, STOC '82, pages 169–180. ACM, 1982.
13. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2007.
14. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Synthesis from temporal specifications using preferred answer set programming. In *Theoretical Computer Science*, pages 280–294. Springer, 2005.
15. S. Jacobs and R. Bloem. Parameterized synthesis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 362–376. Springer, 2012.
16. A. Martin. Adequate sets of temporal connectives in CTL. *Electronic Notes in Theoretical Computer Science*, 52(1):21 – 31, 2002. EXPRESS'01, 8th Intl Workshop on Expressiveness in Concurrency (Satellite Event of CONCUR 2001).
17. S. Schewe and B. Finkbeiner. Bounded synthesis. In *Automated Technology for Verification and Analysis*, pages 474–488. Springer, 2007.
18. R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 3:141–224, 2007.