Learning to Verify Branching Time Properties

Abhay Vardhan and Mahesh Viswanathan *

Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, USA

Abstract. We present a new model checking algorithm for verifying computation tree logic (CTL) properties. Our technique is based on using language inference to learn the fixpoints necessary for checking a CTL formula instead of computing them iteratively as is done in traditional model checking. This allows us to analyze infinite or large state-space systems where the traditional iterations may not converge or may take too long to converge. We allow fairness constraints to be specified for verification of various liveness properties. The main challenge in developing a learning based model checking algorithm for CTL is that CTL properties express nested fixpoints. We overcome this challenge by developing a new characterization of CTL properties in terms of functions that have unique fixpoints. We instantiate our technique to systems in which states are encoded as strings and use a regular inference algorithm to learn the CTL fixpoints. We prove that if the fixpoints have a regular representation, our procedure will always terminate with the correct answer. We have extended our LEVER tool to use the technique presented in this paper and demonstrate its effectiveness by verifying a number of parametric and integer systems.

1 Introduction

Infinite state models, such as pushdown systems, communicating finite state machines with unbounded FIFO buffers, and automata with integer variables, often arise as natural abstractions of software systems at the design and modeling stage. Developing automated verification techniques for such systems is, therefore, an important problem in formal verification.

Computation Tree Logic (CTL) [17] is a temporal logic in which one can express properties about the branching, non-deterministic behavior of the system. Properties about information flow in the system, which cannot be expressed in a specification language that reasons only about individual computations, can be written in CTL. For these reasons, CTL is very often used to describe the correctness requirements of a system.

Algorithmic verification typically involves computing sets of states that are fixpoints of certain functions. For example, in order to check if a certain invariant holds in a system, one needs to compute the set of *reachable states* of the system and then check that none of the reachable states violate the invariant. The computation of the reachable states is a fixpoint computation: the set of reachable states is the smallest set that is closed under the one-step transition relation associated with the system. When one is analyzing an infinite state system, computing such fixpoint sets may not terminate; in fact, the problem of verifying an infinite state system is, in general, undecidable [1]. For example, in an infinite state system, one may need to apply the transition relation an unbounded number of

* Supported in part by DARPA/AFOSR MURI Award F49620-02-1-0325 and NSF 04-29639.

[1] For certain classes of infinite state systems, some problems become decidable; see [21] for some general results.

times to the set of initial states before all the reachable states are encountered. Hence, the focus of the research community has been to develop semi-decidable algorithms for these problems. There have been two broad approaches to algorithmically verify infinite state systems. The first approach uses ideas like *widening* [13, 9, 27] and *acceleration* [10, 8, 7] to conservatively approximate (in the case of invariants) the set of reachable states of the system. The second complementary approach that has been proposed recently, is to use *machine learning* [30, 29, 31, 22]. The idea here is to view the verification problem as a problem where the (unknown) fixpoint set needs to be learnt from examples. The approach has been applied to verify safety properties and ω-regular properties of infinite-state systems.

The learning based approach to verification enjoys several benefits. First, the running time of the algorithm does not depend on the time it takes to converge to the fixpoint, but rather on the size of the symbolic representation of the fixpoint set. If the fixpoint set being computed has a succinct representation, then the learning algorithm will converge to it very quickly. Second, because intermediate approximations to the fixpoint are never computed, it avoids the space overhead of storing fixpoint approximations that may have a large symbolic representation. Finally, the learning based algorithms are typically accompanied by a precise statement of when the method is guaranteed to work: if the set being computed for verification can be represented using the symbolic representation used by the learner, the method is guaranteed to either prove the system to be correct, or produce a counter-example demonstrating its violation. The learning based approach has thus far been used in the context of *regular model checking*, where states of the system are encoded as strings, and sets of states are represented using *deterministic finite automata* (DFA). In this context, the completeness theorem takes the form that if the set of reachable states (in the case of invariant properties) is regular, then the learning based model checking algorithm is guaranteed to work.

In this paper, we present a learning based model checking algorithm for infinite state systems with respect to CTL properties. The algorithm presented here is the first CTL model checker (based on machine learning or otherwise) for infinite state systems with fairness constraints; the CTL model checker for infinite state systems reported in [15] did not account for fairness constraints. It is also the first extension of the learning based method to branching-time properties, as all previous applications of learning were confined to linear-time properties. Finally, there is precise characterization of the class of systems for which this model checking algorithm is complete; for every subformula, if the set of states satisfying it form a regular language, then the algorithm presented here is guaranteed to terminate with the right answer.

In order to apply the learning based verification method to CTL, we need to overcome three fundamental challenges. To better understand these problems, let us consider the case of verifying invariants. As outlined before, the learning based method to verify invariants calls an algorithm that attempts to learn the set of reachable states, and then checks whether the invariant is violated in any of the states in the learnt set. To guarantee soundness, the model checker has to check if the set returned by the learning algorithm is indeed the set of reachable states, without actually computing the set of reachable states again. This turns out to be a difficult problem because while it is easy to check if a set is closed with respect to the transition relation (and hence contains all reachable states), there is no easy way to check if it is the *smallest* such set. Instead, the learning based method checks whether the learnt set contains all reachable states and does not violate the invariant, or whether a specific unsafe state in the learnt set is reachable. These

"approximate" tests turn out to be feasible, and sufficient for the purposes of verifying the invariant. Thus, in the case of invariant verification, the learning based model checker does not ever know whether it has actually computed the set of reachable states, but only knows whether it has discovered a proof of correctness or a proof of violation.

To extend the learning approach to verify CTL, the following approach suggests itself immediately. Similar to the classical model checking algorithm for CTL [17], progressively compute the set of states satisfying each of the subformulas, starting from simple atomic propositions; the only difference being that we learn the sets instead of computing them iteratively. However, this approach runs into the problem that unlike safety properties and ω-regular properties, CTL properties are nested fixpoints, where the set of states satisfying inner subformulas is used in the computation of the set of states satisfying outer subformulas. Hence, we will need a test that checks whether the learning algorithm has learnt the exact set of states satisfying a particular subformula (and not some over or under-approximation). Once again, while it is easy to check if a set is a fixpoint, it is unclear how to check if it is the *least* fixpoint or the *greatest* fixpoint. We overcome this central problem by presenting a new characterization of CTL operators in terms of functions with unique fixpoints. The output of a learning algorithm trying to discover these unique fixpoints can then be easily checked, and this allows us to get a learning based model checking algorithm for CTL.

The next challenge is to adequately take into account the fairness constraints that might be associated with the system being verified. In the case of finite state systems, this is handled using the observation that it is sufficient to only consider fair computations that are *ultimately periodic* and *looping*, i.e., computations that repeatedly execute a sequence of steps that loop to a state. However, this observation does not extend to infinite state systems. In order to soundly verify an infinite state system with respect to fairness constraints, we need to also consider fair computations that are truly infinite, and are not looping. We generalize ideas that we developed for the verification of ω-regular properties [31] to account for fairness.

Finally, we have implemented this algorithm in our tool LEVER, and extensively experimented with the tool on many examples. Since we represent sets of states using DFAs, we use algorithms for learning regular languages that have been investigated in the literature. In order to make them scalable to practical software systems, we make some improvements to these algorithms. First we combine Rivest and Schapire's idea [26] for processing counterexamples, with the Kearns and Vazirani algorithm [23] to yield a space and time efficient learning algorithm for regular languages. Second, the main challenge to the scalability of this algorithm is that the running time and space of the learning algorithm depends on the size of the alphabet used in encoding the state. Since the alphabet size grows exponentially in the number of variables in the system being verified, this presents a significant hurdle when analyzing practical software systems. We address this problem by representing the transition relation of the DFA using ordered BDDs; the idea of representing the transition relation of a DFA using BDDs as been explored before in [24, 3], though in a slightly different context.

The rest of the paper is organized as follows. In Section 2 we discuss the representation we use for programs, the classical model checking algorithm for CTL and give some background about learning algorithms. Section 3 gives details of the learning based verification for CTL with and without fairness constraints. In Section 4, this procedure is instantiated for systems with states represented as strings and it is shown how various operations described in Section 3 can be achieved with regular sets. Section 5 discusses

the implementation of these ideas in our LEVER tool and presents our experimental results. Finally, Section 6 concludes the paper and discusses some future work.

1.1 Related Work

Some of the popular approaches used for verification of infinite state systems are: *widening* [13, 9, 27, 19], where the computation of a fixpoint is extrapolated to get an over-approximation in finite time; *meta-transition* or *acceleration* [10, 8, 7], where the effect of infinite iteration of certain loops in the system is calculated; collapsing of automata states based on simulation relations as in the framework of *Regular Model Checking* [12, 1]; and iterating transducers as described in Boigelot *et. al.* [11]. As mentioned earlier, the main advantage of using the alternative *learning*-based verification is that as long as the set being computed for verification is representable in the class of concepts being learned, the method is guaranteed to provide a sound answer to the verification problem.

We introduced the learning to verify approach in [30, 29] for verifying safety properties of FIFO automata. In [31], we used learning techniques for verifying ω-regular properties (which can express safety as well as liveness properties) and applied it in the context of regular model checking. Concurrently and independently of our work, Habermehl *et al.* [22] have also proposed a learning based approach for verification of systems. However, the important class of branching times was not covered earlier in the learning paradigm and this is what we address in the present paper. The other major work that has addressed CTL properties (using techniques different from learning) is that of Bultan *et. al.* [15] where a combination of widening and finite iteration are used for *conservative approximations*. The notable differences between their work and our work are: first, we also are able to handle fairness constraints which are needed for liveness properties and second, we never return a "do not know" answer which a conservative analysis can return.

Machine learning has also been applied to other contexts in verification such as learning assumptions about the environment for compositional verification [18, 3, 16]); synthesis of interface specifications [2] and mining specifications [5].

2 Preliminaries

2.1 Program Representation

We use an event-action based language as in [14] to represent programs. In this representation, a program is a tuple (V, I, E) where V is a finite set of control or data variables, I gives the initial values of the variables in V and E is a finite set of events. Each event $e \in E$ has an enabling condition $enabled(e)$ which constraints the states in which the transition can be taken and an $action(e)$ specifying how the variables are transformed by the transition. Event-action based languages have been used in the literature to express concurrent programs including abstractions of multi threaded Java programs, parameterized systems, petri nets, communication protocols, counter systems, broadcast protocols and cache coherence protocols [4, 20].

Example 1. A simplified example program written in this language is shown in Figure 1. In this example, a producer adds items to a buffer b while two consumers consume these items from b. The variable o records the total number of items produced while i_1 and i_2 record the number of items consumed by each of the two consumers respectively. In the action clause, the new value of a variable is given by a primed version of that variable

(for brevity, only the variables that are changed are shown). Note that this program has an infinite number of reachable states.

Data Variables: b, o, i_1, i_2: integer
Control Variables: $pc : \{q_0\}$
Initial Conditions: $b = o = i_1 = i_2 = 0, pc = q_0$
Events:

1. **enabled:** $true$,
   action: $b' = b + 1, o' = o + 1$
2. **enabled:** $b > 0$,
   action: $b' = b - 1, i_1' = i_1 + 1$
3. **enabled:** $b > 0$,
   action: $b' = b - 1, i_2' = i_2 + 1$

Fig. 1. A event-action program for a simple buffer problem

2.2 Kripke Structures and CTL

While event-action based languages are rich enough to model and define the semantics of programs, they need to be enhanced in order to perform verification. In particular, states of the system need to be annotated with logical propositions that describe properties that are relevant for verification. For example, a state in which two processes are incorrectly accessing a critical resource simultaneously may be labeled *bad* while other states may be labeled *good*. Such a model of the software system where states are annotated with atomic propositions is formally called a *Kripke structure*, and is defined as follows. It is a quintuple $(S, AP, R, S_0, \mathcal{L})$ where S is a set of (possibly infinite) states, AP is a finite set of atomic propositions, $R \subseteq S \times S$ is a (total) transition relation, $S_0 \subseteq S$ is a set of initial states and $\mathcal{L} : S \to 2^{AP}$ is function that assigns to each state the set of propositions that are true in that state. We restrict ourselves to Kripke structures that are finitely branching, *i.e.*, for any state s, the set $\{s' \mid R(s, s')\}$ is finite. We will sometimes denote $(s_1, s_2) \in R$ by $s_1 \to s_2$. A *computation* starting from state s is a sequence of states $s_0, s_1, \ldots$ such that $s_0 = s$, and $s_i \to s_{i+1}$ for each i. Constructing the Kripke structure corresponding to an event-action program is straightforward, provided we are also given the labeling function $\mathcal{L}$: the states S are functions that assign to each variable of the program a value in the appropriate domain, and each event $e \in E$ defines a binary relation $R_e \subseteq S \times S$ to be $\{(s_1, s_2) \mid s_1 \in \mathit{enabled}(e) \text{ and } (s_1, s_2) \in \mathit{action}(e)\}$. Then, the transition relation is simply given by $R = \bigcup_{e \in E} R_e$.

Kripke structures are sometimes augmented with fairness constraints that hold in the system. Formally, a *Fair Kripke structure* is $(S, AP, R, S_0, \mathcal{L}, \mathcal{F})$, where $(S, AP, R, S_0, \mathcal{L})$ is a Kripke structure, and $\mathcal{F} \subseteq S$ is the set of *fair* states. A *fair computation* starting

from s is then a computation $s_0, s_1, \ldots$ starting from s that visits the fair states infinitely often, i.e., $s_j \in \mathcal{F}$ for infinitely many j [2].

Computation Tree Logic (CTL) [17] is a popular temporal logic used to specify correctness requirements. Formulas in the logic are built up from atomic propositions, using the temporal operators X("next time"), G ("globally"), and U ("until"). In addition, the logic has the path quantifier E ("for some computation path"). The formal syntax of formulas in CTL is as follows (p is an atomic proposition) [3]:

$$f ::= p \mid \neg f \mid f_1 \vee f_2 \mid EXf \mid EGf \mid E[f_1 U f_2]$$

Informally, a state s satisfies a formula EXf if there is a computation starting from s where f holds in the second state; s satisfies EGf if there is a computation starting from s where every state satisfies f; s satisfies $E[f_1 U f_2]$ if there is a computation starting from state s such that f_1 holds in all states until a state satisfying f_2 is encountered. The connectives $\neg$ and $\vee$ are boolean negation and disjunction and are interpreted as in propositional logic. In the presence of fairness constraints, the path quantifiers are interpreted only over fair paths. So for example, a state s satisfies EGf in the presence of fairness, if there is a *fair* computation starting from state s such that f holds in all states. For a formal presentation of the semantics for all the operators, the reader is referred to [17].

2.3 Model Checking CTL and Fixpoints

The *verification* or *model checking* problem for CTL is as follows. Given a system description in terms of a Kripke structure $(S, AP, R, S_0, \mathcal{L})$ (or alternatively, a fair Kripke structure $(S, AP, R, S_0, \mathcal{L}, \mathcal{F})$) and a CTL specification f, we have to check if all states $s \in S_0$ satisfy the formula f. The standard algorithm [17] for CTL model proceeds by progressively computing the set of states that satisfy the various subformulas of f (including f itself). Initially, we compute the set of states that satisfy each of the atomic propositions in f. We know that s satisfies p iff $p \in \mathcal{L}(s)$. The algorithm then proceeds in stages. In the ith stage, the set of states corresponding to subformulas with $i-1$ nested CTL operators are computed using the results of the computation in the previous stages. Once the states satisfying f are found, the system is deemed to be correct if and only if this set wholly contains S_0, the initial set of states.

The algorithm to compute the set of states satisfying a subformula g in some stage, say i, depends on the outermost logical operator of g. For the operators EU and EG, computing the set of states satisfying them, involves computing the least and greatest fixpoint, respectively, of certain functions. We conclude this section, by introducing terminology about fixpoints and operators that we will find useful.

Consider a function $T : 2^S \mapsto 2^S$, from sets of states to sets of states. A *fixpoint* for T is a set $Z \subseteq S$ such that $T(Z) = Z$; it is the *least fixpoint* if it is the least, with respect to $\subseteq$-ordering, among all the fixpoints of T, and is denoted as $\mu(T)$. *Greatest fixpoint* of T is defined analogously and is denoted by $\nu(T)$. The function T is said to be *monotonic* if

[2] Usually fairness in CTL is described by a *set* of constraints using the so called *generalized* Büchi condition. However, it is known that a *generalized* Büchi condition can be converted into the kind of fairness constraint described here.

[3] Typical presentations of CTL have additional temporal operators like F ("eventually") and R ("release"), and also the universal path quantifier. However it can be shown [17] that the logic presented here as just as expressive

$Z_1 \subseteq Z_2 \Rightarrow T(Z_1) \subseteq T(Z_2)$. It is well-known that if T is monotonic, then both $\mu(T)$ and $\nu(T)$ exist. T is $\cup$-*continuous* if $Z_1 \subseteq Z_2 \subseteq \ldots \Rightarrow T(\cup_i Z_i) = \cup_i T(Z_i)$. It is $\cap$-*continuous* if $Z_1 \supseteq Z_2 \supseteq \ldots \Rightarrow T(\cap_i Z_i) = \cap_i T(Z_i)$. Another well-known result about fixpoints is that if T is $\cup$-*continuous* then $\mu(T)$ can be calculated by starting from the empty set and repeatedly applying the function until the result does not change. Analogously, if T is $\cap$-*continuous* then the greatest fixpoint can be computed by starting from the entire set S and repeatedly applying the function until convergence. Finally, T^i will denote the i-fold composition of a function T.

2.4 Learning Regular Languages

We use the framework of *active learning* [6], in which the learning algorithm is given access to a knowledgeable teacher who can be queried. The teacher can be thought of as a pair of oracles: a *membership oracle* and an *equivalence oracle*. The membership oracle provides answers to queries about whether an example belongs to the concept being learnt or not. The equivalence oracle is a more powerful oracle which answers question about whether a hypothesis proposed by the learning algorithm is indeed equivalent to the concept being learnt. If at some point the learning algorithm's hypothesis is deemed correct by the equivalence oracle then the learning process stops. If on the other hand, the learner submits a hypothesis which is not equivalent to the target concept, the equivalence oracle not only says *no*, but also provides a counter-example to demonstrate why the hypothesis is wrong. The counter-example is either an example belonging to the hypothesis but not to the target concept, or it is an example belonging to the target concept but not to the submitted hypothesis.

The specific algorithm that we use to learn regular languages is due to Kearns and Vazirani [23], which is a variation of the classical algorithm by Angluin [6]. The algorithm maintains a candidate DFA for the language being learnt. This DFA has two characteristics. First it is a minimal DFA, i.e., every pair of states can be *distinguished* by string that leads one to an accept state and the other to a non-accept state. Second, the DFA is consistent with the sample strings the learning algorithm has seen so far, i.e., all strings observed that belong to the language are accepted by the DFA, and all observed strings that do not belong to the language are rejected. Progressively, the algorithm queries the equivalence oracle and the membership oracle, and stops when the equivalence oracle deems the candidate DFA to be correct. In this process, the learner may discover inconsistencies with the current DFA, which it resolves by splitting certain states. The details of the algorithm determine how the DFA is represented, which queries are asked, and how inconsistencies when discovered are resolved by splitting states. In what follows, we present only a brief overview of the algorithm, giving only the details that are relevant to highlight our improvements. For a comprehensive presentation the reader is referred to [23].

Each state of the candidate DFA is represented as a string, namely, the input string that takes the DFA from the initial state to the given state. These strings are called *access strings* and the learner maintains a set AS of such access strings. In addition, the learner also maintains a set DS of *distinguishing strings*, that demonstrate why each string in AS will correspond to a different state in the desired minimal DFA. So for any two access strings $s_1$and s_2 there is a string $d \in DS$ such that exactly one out of $s_1.d$ and $s_2.d$ belongs to the regular language being learnt. The algorithm maintains the sets AS and DS in a binary *classification tree* where each internal node is labeled by a string in DS and each leaf by a string in AS. Any one string d that distinguishes between

two access strings can be placed at the root of the tree. All access strings s such that sd belongs to the concept being learnt are placed in the right subtree and the rest are placed in the left subtree. This process is repeated for each subtree until each access string is at its own leaf. The classification tree implicitly represents the current candidate automaton in the following manner. The states correspond to the access strings AS. The transition on a symbol a from a state s can be obtained by "sifting" through the classification tree as follows. Start from the root (say labeled d), and check whether the string $s.a.d$ belongs to the learnt concept or not, by asking the membership oracle. If $s.a.d$ belongs to the concept then move to the right child else move to the left child. Repeat this process recursively, until a leaf is reached. This is the target state of the transition.

Observe, that the process of "sifting" can be used to determine whether two strings s_1 and s_2 must go to different states in the smallest DFA for the concept: if sifting s_1 and s_2 result in reaching different leaves then s_1 and s_2 must go to different states. This observation plays a crucial role determining how the candidate DFA must be refined when an inconsistency is encountered. Let w be the counter-example returned by the equivalence oracle, and let u be the first prefix of w where the state s reached on reading u in the DFA is different than the state s' obtained when u is sifted through the classification tree. Then the state s must be split in order to resolve the inconsistency. In our variant of the Kearns and Vazirani algorithm, we use the Rivest and Schapire's [26] binary search to discover this prefix u, to get a faster algorithm. The resulting algorithm is guaranteed to learn the minimal DFA and runs in polynomial time.

Theorem 1. *([26]) The regular inference algorithm described above always terminates after making $O(kn^2 + n \log m)$ membership queries and $O(n)$ equivalence queries. Here, k is the size of the alphabet, n is the number of states in the minimal automaton representing the target regular language, and m is the longest counterexample returned by the teacher.*

3 Learning to Verify CTL Properties

Recall (from Section 2.3) that the classical model checking algorithm for CTL proceeds by inductively determining the set of all states that satisfy each of the subformulas. For each subformula, the algorithm to determine the set of states satisfying it is determined based on the outermost logic operator. Given a suitable representation for sets of states, $\neg f$ and $f_1 \vee f_2$ correspond to performing the boolean operations of complementation and union on the sets of states satisfying f, f_1 and f_2. In the case of EX it involves computing predecessors: $EX(Z) = \{s \mid \exists s \rightarrow s' \text{ and } s' \in Z\}$.

The most interesting cases are those of EU and EG, which are handled by computing fixpoints. To illustrate the challenges in developing a model checking algorithm for infinite-state systems, let us consider a formula $E[true\ U\ f]$ which is also sometimes written as EFf. The set of states satisfying EFf consists of the states which can reach a state labeled by f. Thus, EFf can be found by starting with a set Z_0 consisting of states satisfying f and in the ith iteration adding the states that can reach a state satisfying f in i-steps. Clearly this method of computing the set of states satisfying EFf may not terminate for a system with infinitely many states. As mentioned before, therefore, our idea is to learn this set instead of performing this iterative computation. In order to do this, we have to answer membership and equivalence queries for the set of states satisfying EFf. We do have a weak test for equivalence; if the set hypothesized for EFf

changes under backward reachability then it is certainly not the right set. However, even if it does not change under backward reachability, it may be just an overapproximation of EFf (in the case of EGf this can be an underapproximation). For membership queries the situation is even more difficult; it is unclear how we can answer whether a state s satisfies EFf without solving the original verification problem. The solution is to learn a set with more information from which EFf can be computed and which allows answering membership and equivalence queries. In the case of EFf, one simple way to achieve this is to learn a set X of (s, i) pairs where $(s, i) \in X$ means that the state s can reach some state labeled f in i steps. Now, a membership check $(s, i) \in X$ involves checking if in i steps a computation from s can reach X, which is an easier problem. Moreover, it can be shown that there is a unique set that does not change under the function $\Gamma(Z) = \{(s, j) \mid \exists s \rightarrow s' \text{ and } (s', j-1) \in Z\} \cup \{(s, 1) \mid s \models f\}$. This allows us to answer the equivalence query for EFf exactly.

Using the ideas informally presented above, we can develop a learning based algorithm for CTL, in the absence of fairness constraints. This is formally presented next. After this, we consider the case of model checking in the presence of fairness constraints.

3.1 CTL formulas without Fairness

First let us consider the problem of model checking a Kripke structure that does not have any fairness constraints. As we saw before, $\neg$, $\vee$, EX can be handled in a fairly straightforward manner. From [17], we know that $E[f_1 U f_2]$ is the least fixpoint of the function $T_{E[f_1 U f_2]} : 2^S \mapsto 2^S$ given by $T_{E[f_1 U f_2]}(Z) = [\![f_2]\!] \cup ([\![f_1]\!] \cap EX\ Z)$ or equivalently:

$$T_{E[f_1 U f_2]}(Z) = \{s \mid s \in [\![f_1]\!] \text{ and } \exists s'.s \rightarrow s' \text{ and } s' \in Z\} \cup [\![f_2]\!]$$

Here, $[\![f]\!]$ denotes the set of states satisfying the subformula f. Further, EGf is the greatest fixpoint of the function $T_{EGf}(Z) : 2^S \mapsto 2^S$ given by $T_{EGf}(Z) = [\![f]\!] \cap EX\ Z$ or equivalently:

$$T_{EGf}(Z) = \{s \mid s \in [\![f]\!] \text{ and } \exists s'.s \rightarrow s' \text{ and } s' \in Z\} \cap [\![f]\!]$$

It can also be shown that $T_{E[f_1 U f_2]}$ and T_{EGf} are monotonic, $T_{E[f_1 U f_2]}$ is $\cup$-continuous and T_{EGf} is $\cap$-continuous. As discussed before, we want to derive new functions which will allow us to use learning techniques.

Let $\mathbb{N}$ be the set of natural numbers, and $\Gamma_{E[f_1 U f_2]} : 2^{S \times \mathbb{N}} \mapsto 2^{S \times \mathbb{N}}$ and $\Gamma_{EGf} : 2^{S \times \mathbb{N}} \mapsto 2^{S \times \mathbb{N}}$ be two functions defined as follows.

Definition 1.

$$\begin{aligned}\Gamma_{E[f_1 U f_2]}(Z) = \{(s, i+1) \mid\ & s \in [\![f_1]\!] \textit{ and} \\ & \exists s'.s \rightarrow s' \textit{ and } (s', i) \in Z\} \cup\ [\![f_2]\!] \times \{1\}\end{aligned}$$

Definition 2.

$$\begin{aligned}\Gamma_{EGf}(Z) = \{(s, i+1) \mid\ & s \in [\![f]\!] \textit{ and} \\ & \exists s'.s \rightarrow s' \textit{ and } (s', i) \in Z\} \cup\ [\![f]\!] \times \{1\}\end{aligned}$$

Intuitively, $\Gamma_{E[f_1 U f_2]}$ (resp. Γ_{EGf}) keeps an additional counter i with each state, with the counter value of 1 associated with all states in $[\![f_2]\!]$ (resp. $[\![f]\!]$). Both functions do a similar backward transition step as $T_{E[f_1 U f_2]}$ and T_{EGf} except that at each step the

value of the counter is incremented by one. It can be shown that $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} are monotonic and $\cup$-continuous.

Instead of trying to learn the least (greatest) fixpoint of $T_{E[f_1 U f_2]}$, we can learn $\Gamma_{E[f_1 U f_2]}$ (Γ_{EGf}) and then retrieve the desired fixpoint of T. The first challenge is to answer equivalence queries. For this, we have the following proposition.

Proposition 1. *Each of $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} has a unique fixpoint.*

Proof (Sketch). Since $\Gamma_{E[f_1 U f_2]}$ is monotonic, it has fixpoints. Let Z and Z' be two fixpoints for $\Gamma_{E[f_1 U f_2]}$. Let $Z_{\leq i}$ be the set $\{(s,j) \mid (s,j) \in Z \text{ and } j \leq i\}$. We will prove by induction on i that for all i, $Z_{\leq i}$ and $Z'_{\leq i}$ are the same. The base case for $i = 1$ is trivial since for all fixpoints of $\Gamma_{E[f_1 U f_2]}$, the set of states with counter value of i as 1 has to be $[\![f_2]\!]$. Assume that the inductive hypothesis holds up to some $j > 1$. We need to show that $(s, j{+}1) \in Z_{\leq j+1} \Leftrightarrow (s, j{+}1) \in Z'_{\leq j+1}$. If $(s, j{+}1) \in Z$ then $(s, j{+}1) \in \Gamma_{E[f_1 U f_2]}(Z)$. This implies $s \in [\![f_1]\!]$ and there is some $(s', j) \in Z$ or $(s', j) \in Z_{\leq j}$ such that $s \to s'$. By the inductive hypothesis, $(s', j) \in Z'_{\leq j}$ or $(s', j) \in Z'$. But then, $(s, j+1) \in \Gamma_{E[f_1 U f_2]}(Z')$ or $(s, j{+}1) \in Z'$ or $(s, j{+}1) \in Z'_{\leq j+1}$. Similarly, if $(s, j{+}1) \in Z'_{\leq j+1}$ then $(s, j{+}1) \in Z_{\leq j+1}$. This establishes $Z_{\leq j+1} = Z'_{\leq j+1}$.

The proof for Γ_{EGf} goes through in the same manner.

The above proposition helps us answer equivalence queries as follows. The query asks whether a proposed hypothetical set Z' is the same as the fixpoint Z of Γ (Γ could be $\Gamma_{E[f_1 U f_2]}$ or Γ_{EGf}). Since the fixpoint of Γ is unique, this can be correctly answered by checking if Z' itself is a fixpoint, *i.e.* comparing Z' with the image $\Gamma(Z')$. It is important to note that in general the fixpoints encountered in CTL verification are not unique; it is due to our construction of Γ that the fixpoint is unique.

The next challenge is to answer a membership query asking if (s, i) is in the fixpoint of $\Gamma_{E[f_1 U f_2]}$ or Γ_{EGf}. The following proposition shows that in order to check if some pair (s, i) is in the fixpoint of $\Gamma_{E[f_1 U f_2]}$ (Γ_{EGf}) we only need to check the i-fold composition $\Gamma^i_{E[f_1 U f_2]}(\emptyset)$ ($\Gamma^i_{EGf}(\emptyset)$).

Proposition 2.

1. *Let Z be the fixpoint of $\Gamma_{E[f_1 U f_2]}$. Then, for all $i > 0$, $(s, i) \in \Gamma^i_{E[f_1 U f_2]}(\emptyset)$ if and only if $(s, i) \in Z$*
2. *Let Z be the fixpoint of Γ_{EGf}. Then, for all $i > 0$, $(s, i) \in \Gamma^i_{EGf}(\emptyset)$ if and only if $(s, i) \in Z$*

Proof (Sketch).

1. To show that $(s, i) \in \Gamma^i_{E[f_1 U f_2]}(\emptyset)$ implies $(s, i) \in Z$ we observe that $\Gamma_{E[f_1 U f_2]}$ is $\cup$-continuous. This gives us $Z = \cup_i \Gamma^i_{E[f_1 U f_2]}(\emptyset)$.
   We prove by induction on i that $(s, i) \in Z$ implies $(s, i) \in \Gamma^i_{E[f_1 U f_2]}(\emptyset)$. Base case for $i = 1$ is trivial. Assume the inductive hypothesis up to j. If $(s, j + 1) \in Z$ then since Z is a fixpoint, $(s, j + 1) \in \Gamma_{E[f_1 U f_2]}(Z)$. But then $s \in [\![f_1]\!]$ and there must be some $(s', j) \in Z$ such that $s \to s'$. By the inductive hypothesis, $(s', j) \in \Gamma^j_{E[f_1 U f_2]}(\emptyset)$ which easily leads us to $(s, j + 1) \in \Gamma^{j+1}_{E[f_1 U f_2]}(\emptyset)$.
2. The proof follows the same steps as the case for $\Gamma_{E[f_1 U f_2]}$.

In case $Z' \neq Z$, the learner typically also needs an element in the symmetric difference $Z' \oplus Z$ to make progress. We can obtain such an element using the method in our earlier work [31] which is summarized for $\Gamma_{E[f_1 U f_2]}(Z')$ ($\Gamma_{EGf}(Z')$ can be done in a similar manner) as follows for the different cases possible.

- $\Gamma_{E[f_1 U f_2]}(Z') \backslash Z \neq \emptyset$. Let $l = (s, i)$ be some element in this set. If $l = (s, 1)$ then $l \in Z$, because the only way we can have any $(s, 1)$ in $\Gamma_{E[f_1 U f_2]}(Z')$ is if $(s, 1) \in \Gamma_{E[f_1 U f_2]}(\emptyset)$. In this case, l is in Z and hence in $Z' \oplus Z$. If $l = (s, i)$ for some $i > 1$, we can check if $l \in Z$ using the membership query. If yes, then l is also in $Z' \oplus Z$ and we are done. Otherwise, $l \in \Gamma_{E[f_1 U f_2]}(Z')$ because of the existence of some pair $(s', i-1) \in Z'$. Clearly $(s', i-1)$ cannot be in Z otherwise (s, i) would have to be in Z. Hence $(s', i-1) \in Z' \oplus Z$.
- $\Gamma_{E[f_1 U f_2]}(Z') \subsetneq Z'$. From standard fixpoint theory, since Z happens to also be the least fixpoint of $\Gamma_{E[f_1 U f_2]}$, it must be the intersection of all prefixpoints of $\Gamma_{E[f_1 U f_2]}$ (a set Y is a prefixpoint if it *shrinks* under the function T, *i.e.* $T(Y) \subseteq Y$). Now, Z' is clearly a prefixpoint. Applying $\Gamma_{E[f_1 U f_2]}$ to both sides of the equation $\Gamma_{E[f_1 U f_2]}(Z') \subsetneq Z'$ and using monotonicity of $\Gamma_{E[f_1 U f_2]}$, we get

  $$\Gamma_{E[f_1 U f_2]}(\Gamma_{E[f_1 U f_2]}(Z')) \subsetneq \Gamma_{E[f_1 U f_2]}(Z')$$

  Thus, $\Gamma_{E[f_1 U f_2]}(Z')$ is also a prefixpoint. Let l be some string in the set $Z' \backslash \Gamma_{E[f_1 U f_2]}(Z')$. Since l is outside the intersection of two prefixpoints, it is not in the least fixpoint Z. Hence, l is in $Z' \oplus Z$.

Finally, once the learning procedure is done, we need to retrieve the set of states satisfying $E[f_1\ U\ f_2]$ (resp. EGf) from fixpoint of $\Gamma_{E[f_1 U f_2]}$ (resp. Γ_{EGf}). The following proposition addresses this.

Proposition 3.

1. *Suppose Z be the fixpoint of $\Gamma_{E[f_1 U f_2]}$. Then,*

   $$[\![E[f_1\ U\ f_2]]\!] = \{s \mid \exists i \text{ s.t. } (s, i) \in Z\}$$

2. *Suppose Z is the fixpoint of Γ_{EGf}. Then, $[\![EGf]\!] = \{s \mid \forall i > 0 (s, i) \in Z\}$.*

Proof (Sketch).

1. First, we show that if $(s, i) \in Z$ then $s \in [\![E[f_1\ U\ f_2]]\!]$. We use induction on i. Base case for $i = 1$ is trivial. Next, if $(s, j+1) \in Z$ then $(s, j+1) \in \Gamma_{E[f_1 U f_2]}(Z)$. But then $s \in [\![f_1]\!]$ and there must exist some $(s', j) \in Z$ such that $s \rightarrow s'$. The inductive hypothesis for j gives $s' \in [\![E[f_1\ U\ f_2]]\!]$. This means that there is a path from s' to a state satisfying f_2 such that all states in the path satisfy f_1. Since, $s \rightarrow s'$ and s satisfies f_2, this path can be extended to start from s. Hence, $s \in [\![E[f_1\ U\ f_2]]\!]$. For the other direction, we show that if $s \in [\![E[f_1\ U\ f_2]]\!]$ then for some $i \in \mathbb{N}$, $(s, i) \in Z$. If a state s satisfies $E[f_1\ U\ f_2]$ then there exists a path from s to a state in f_2 such that all the states in this path satisfy f_1. If the length of the path is i, it can be seen that (s, i) will get included in $\Gamma^i_{E[f_1 U f_2]}(\emptyset)$. Hence, $(s, i) \in Z$.
2. First, we show that if for all $i > 0$, $(s, i) \in Z$ then $s \in [\![EGf]\!]$. Construct a tree with root s, containing edges appearing in all shortest paths such that all states in this path satisfy f. A few observations about this tree are in order. First, the tree is finite

branching; an immediate consequence of the Kripke structure being finite branching. Second, all nodes of the tree satisfy f. Finally, this tree has infinitely many vertices. By König's Lemma, there must be an infinite path in the tree. Clearly, this infinite path witnesses EGf.

For the other direction, we need to show that if $s \in Z$ then for all $i > 0$, $(s, i) \in Z$. If a state s satisfies EGf then there exists an infinite path from s such that all the states in this path satisfy f. If s_i is the ith state on this path (counting s as s_1), it can be seen that (s, i) will get included in $\Gamma^i_{EGf}(\emptyset)$. Hence, for all $i > 0$, $(s, i) \in Z$.

Example 2. We illustrate the verification procedure using the system described in Figure 1. Suppose we want to verify the CTL property $AG(i_1 + i_2 \leq o)$ which says that in all states reachable from the initial states the number of items consumed is always less than the number of items generated. This can be written as $\neg EF(\neg(i_1 + i_2 \leq o))$ or $\neg E[true\ U\ (i_1 + i_2 > o)]$. Since there is only a single control state q_0, we can represent the global state of the system by a four-tuple giving the values of the data variables, $x = (b, o, i_1, i_2)$. Then, we have to calculate the least fixpoint of the function

$$\begin{aligned} T(Z) = &\{(b, o, i_1, i_2) \mid i_1 + i_2 > o\} \\ &\cup \{x \mid x \rightarrow x' \text{ and } x' \in Z\} \end{aligned}$$

This is transformed into another function

$$\begin{aligned} \Gamma(Z) = &\{(b, o, i_1, i_2, 1) \mid i_1 + i_2 > o\} \\ &\cup \{(x, j + 1) \mid x \rightarrow x' \text{ and } (x', j) \in Z\} \end{aligned}$$

Once the fixpoint for Γ is learnt, we can project away the fifth component of the states of the fixpoint to get the states satisfying $EF(i_1 + i_2 > o)$. We complement this set and then check if all the initial states are included in the complement. If the answer is yes, then the system verifies the property otherwise it does not.

3.2 CTL with Fairness Constraints

We are now ready consider the problem of model checking a Kripke structure that has fairness constraints $\mathcal{F}$. Evaluating CTL formulas with fairness constraints is known to be harder than the case where there are no fairness constraints. As shown in [17], the problem can be reduced to the following. Let *fair* denote the set of all states s such that there is a fair computation starting from s. It can be shown that $EX(f)$ under fairness conditions is equivalent to $EX(f \wedge fair)$ without fairness conditions. Similarly, $E[f\ U\ g])$ under fairness conditions is equivalent to $E[f\ U\ g \wedge fair])$ without fairness conditions. The set *fair* can be shown to be the states satisfying $EGtrue$ under the fairness constraint. Therefore, if we can evaluate a formula EGf under fairness constraints, we can compute all other CTL formulas using the method in Section 3.1.

Let us now look at the learning problem for EGf under a fairness constraint $\mathcal{F}$. As described in Section 2.3, EGf means that there exists a path beginning with the current state on which f holds globally and states in $\mathcal{F}$ are encountered infinitely often on this path. The set of such states Z is the largest set with the following properties: a) all of the states in Z satisfy f, b) for all states $s \in Z$, there is a sequence of states of length one or greater to a state in Z which is also in $\mathcal{F}$ such that all states on the path satisfy f. This set can be written as the greatest fixpoint of a function $T(Z) = f \wedge EX\ E[f\ U\ (Z \wedge \mathcal{F})]$ but we cannot directly use the procedure outlined in the Section 3.1 because each application of

the function requires evaluating an EU formula which itself needs a fixpoint computation. However, we can adapt a fixpoint characterization we developed in an earlier work ([31]) to EGf.

Definition 3. *Let $\Gamma^{fair}_{EGf} : 2^{S\times\mathbb{N}\times\mathbb{N}}$ be a function defined by $\Gamma^{fair}_{EGf}(Z) = ((\Gamma_1(Z)\cup\Gamma_2(Z)\cup \Gamma_3(Z)) \cap [\![f]\!]) \times \mathbb{N} \times \mathbb{N}$ as follows.*

$$\begin{array}{lll} \Gamma_1(Z) = \{(s,0,j) & | & s \in \mathcal{F} \text{ and } j \in \mathbb{N}\} \\ \Gamma_2(Z) = \{(s,i,j) & | & s \notin \mathcal{F} \text{ and } \exists s'.\, s \rightarrow s' \\ & & \exists j' < j.\, (s',i,j') \in Z\} \\ \Gamma_3(Z) = \{(s,i,j) & | & s \in \mathcal{F} \text{ and } \exists s'.\, s \rightarrow s' \\ & & \exists j' < j.\, (s',i-1,j') \in Z\} \end{array}$$

Intuitively, we associate two counters with each state. Let Z_{EGf} be the fixpoint of Γ^{fair}_{EGf}. A triple (s,i,j) in Z_{EG} means that there exists a path of length j starting from s which encounters at least $i+1$ states labeled with $\mathcal{F}$ and all states in this path satisfy f. Since this can be checked in finite time, we have a method of answering membership queries.

Proposition 4. *Γ^{fair}_{EGf} has a unique fixpoint (Z_{EGf}). Further, the set of states satisfying EGf is given by*

$$\{s \mid \forall i.\exists j.(s,i,j) \in Z_{EGf}\}$$

Proof. The proof follows the same steps as in [31].

The uniqueness of the fixpoint allows equivalence queries to be answered as before. The proposition also gives us a way to compute EGf from Z_{EG}.

To recapitulate, we have developed fixpoint characterizations for all CTL operators with or without fairness such that each such fixpoint can be computed using a learning procedure. This allows us to model check any CTL formula by starting from the innermost sub-expressions and finally checking to see if all initial states are in the states for the outermost expression.

The overall verification procedure is depicted in Figure 2.

4 Representing States with Regular Sets

In the previous section, we presented a general set of conditions under which we can use a learning based approach to verify CTL properties of systems. In this section, we give details of how this can be achieved within the context of using regular languages to represent sets of states. Regular sets are a popular symbolic representation for sets of states of for infinite state systems and have been successfully used ([12]) for modeling *parameterized systems, FIFO automata, systems with integer variables* and *push down stacks.*

We assume that the states of the system can be encoded as strings over some finite alphabet ρ. We further assume that the *enabled* and *action* pairs of the events the program can be used to create a transducer representing the transition relation of the Kripke structure. The transducer is a finite state machine which accepts a pair of strings (s_1, s_2) if the state corresponding to s_1 can transition to the state corresponding to s_2. We assume that the set of initial states; the set of states with labeled with a atomic proposition; and the fairness constraint $\mathcal{F}$ are all given as regular sets.

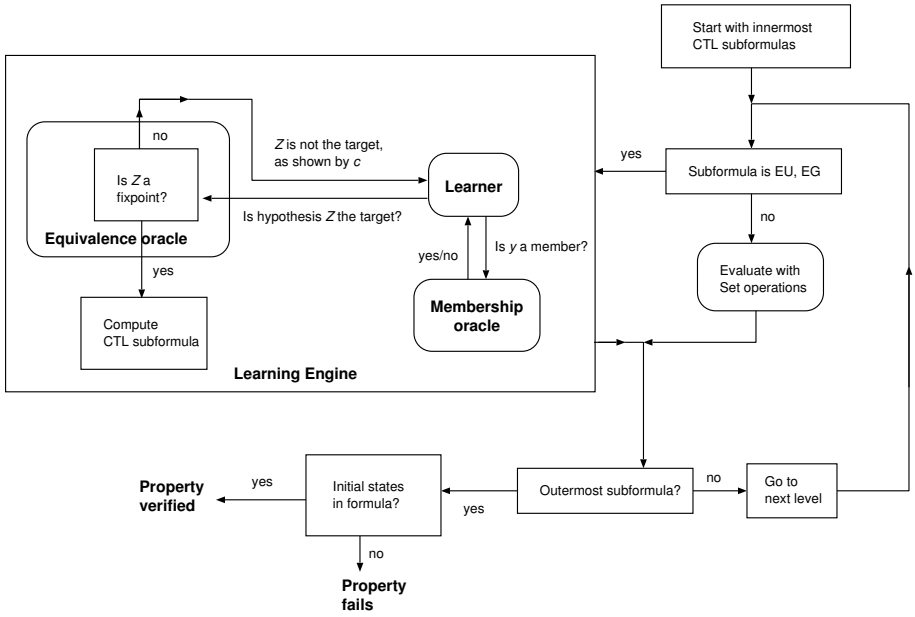

Fig. 2. Verification procedure

4.1 Representation of states with counter

In general, a set Z of pairs (s, i) is a subset of $\rho^* \times \mathbb{N}$. To encode Z as a regular set we use the alphabet Σ given by $(\rho \cup \{\bot\}) \times \{0, 1\}$. This is the alphabet that will be used by the learning algorithm. Here $\bot$ is a new "filler" symbol. An element (s, i) is encoded as string over Σ such that projecting the symbols on the first component gives us s (the $\bot$ symbols are ignored); and projecting on the second component gives i in binary notation. A similar encoding can be used to represent sets of triples (s, i, j).

4.2 Symbolic computation of operators

The various operations required for verification can be done efficiently using regular sets. Standard procedures are available for complementation and union of regular sets represented by finite automata. Given a DFA M_Z representing a set of states Z, the set of states $EX(Z)$ can be found as follows. Let τ_{inv} which represents the inverse of the Kripke relation. We define a construction $\tau(M)$ for a transducer τ and DFA M as a DFA (say D) whose states are the cross product of the states of M and the states of τ. Then, D accepts a string s if and only if M accepts some string s' and τ accepts (s', s). It is easy to see that $EX(Z)$ is simply $\tau_{inv}(M_Z)$. Next, we discuss how to compute the image of a regular set of states under the function $\Gamma_{E[f_1 U f_2]}$ (the case for Γ_{EGf} and Γ^{fair}_{EGf} can be handled similarly).

Definition 4. *Given Z a set of strings in the alphabet of Σ, define*

$$Inc(Z) = \{(s, i) \mid (s, i-1) \in Z\}$$
$$Dec(Z) = \{(s, i) \mid (s, i+1) \in Z\}$$

It is known ([10]) that if natural numbers are represented by strings in binary, a transducer exists which simply increments its input string.[4] This can be used to create a transducer τ_{Inc} for the operation *Inc*. Essentially, τ_{Inc} only changes the "component" for the counter i in any input string and copies the "component" for s. A similar construction can be defined for *Dec*. Further, τ_{inv} can be used to create another transducer τ'_{inv} which ignores the counter component and only changes the state component. Using this construction, for a DFA M_Z representing a set of states Z, $\Gamma_{E[f_1 U f_2]}(Z)$ is found by computing $\tau_{Inc}(\tau'_{inv}(M_Z))$, intersecting the resulting regular set with the regular set $[\![f_1]\!] \times \mathbb{N}$ and finally applying union with the regular set $[\![f_2]\!] \times \{1\}$.

Let $Proj_1$ denote the projection to the first component, $Proj_2$ to the second component and $Proj_{1,2}$ to the first and second component (if there are more than two components). These projections can be done on regular sets using homomorphisms. For example, $Proj_1$ can be done by a homomorphism $h : \Sigma \mapsto \rho^*$ which takes each letter in Σ and maps it to a letter in ρ which corresponds to the state component.

Finally, given a regular set Z we need a way to calculate the following.

- $\{s \mid \exists i \text{ s.t. } (s,i) \in Z\}$ (needed for EU subformulas): This can be calculated as $Proj_1(Z)$.
- $\{s \mid \forall i > 0 (s,i) \in Z\}$ (needed for EG subformulas without fairness): This is the complement of the set $\{s \mid \exists i.(s,i) \notin Dec(Z)\}$. Thus, the desired set is $\overline{Proj_1(\overline{Dec(Z)})}$.
- $\{s \mid \forall i.\exists j.(s,i,j) \in Z\}$ (needed for EG subformulas with fairness): This can be written as
$$\{s \mid \neg(\exists i \neg(\exists j.(s,i,j) \in Z))\}$$
or equivalently, $\overline{Proj_1(\overline{Proj_{1,2}(Z)})}$.

Figure 3 outlines the procedure that can be used to compute an EU CTL subformula.

4.3 Soundness and Completeness

The learning based verification procedure is always sound since it computes all CTL subformulas exactly. Further, we have the following completeness result.

Theorem 2. *Given a CTL property to verify, assume that the following conditions hold:*

1. *Given a Kripke structure K, for every subformula of the form $E[f_1 U f_2]$ or EGf the fixpoints of $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} have a regular representation*
2. *Given a fair Kripke structure K, for every subformula of the form $E[f_1 U f_2]$ or EGf the fixpoints of $\Gamma_{E[f_1 U f_2]}$ and Γ^{fair}_{EGf} have a regular representation*

Then the learning-based verification procedure will always terminate and correctly infer whether the system satisfies the given CTL property.

Remark 1. Note that the set of states for the subformulas for other CTL operators $\neg$, $\vee$ and EX always have regular representation since they can be obtained using standard automata operations.

[4] For pedagogical reasons, we are visualizing a transducer as taking an input string and producing possibly different output strings.

algorithm *learner*
begin
Regular inference algorithm
end

algorithm *isMember*
Input: (s, i)
Output: is (s, i) in fixpoint?
begin
  Compute $Z_i = \Gamma^i_{E[f_1 U f_2]}(\emptyset)$ if not done already
  Is $(s, i) \in Z_i$?
  If yes *return* true
  else *return* false
end

algorithm *Equivalence Check*
Input: Hypothesis Z'
Output: For fixpoint Z, is $Z' = Z$?
If not, then some string in $Z' \oplus Z$
begin
  If $\Gamma_{E[f_1 U f_2]}(Z') \setminus Z' \neq \emptyset$ {fixpoint check}
    let $(s, i) \in \Gamma_{E[f_1 U f_2]}(Z') \setminus Z'$
    Find (s', i') which causes (s, i) to be in $\Gamma_{E[f_1 U f_2]}(Z')$
    if isMember((s, i))
      return (no, (s, i))
    else
      return (no, (s', i'))
  else if $\Gamma_{E[f_1 U f_2]}(Z') \subsetneq Z'$
    return (no, $l \in (Z' \setminus \Gamma_{E[f_1 U f_2]}(Z'))$)
  else {found fixpoint}
    CTL subformula $= Proj_1(Z')$
end

Fig. 3. Computing $E[p_1\ U\ p_2]$ with regular sets

4.4 Complexity Analysis

The main cost of the verification procedure is learning the fixpoints. Let f be the number of CTL subformulas requiring fixpoint computations. We now analyze the cost of the learning procedure.

Let m be the length of the longest string returned by the teacher in a negative answer to an equivalence query, n be the number of states of the minimal automaton representing a fixpoint, k be the size of the alphabet of the learned language and t be the number of states of the automaton representing the transducer for the function Γ whose fixpoint is being learned. By Theorem 1, the language inference algorithm makes $O(kn^2 + n \log m)$ membership queries and $O(n)$ equivalence queries. The worst case for the equivalence query for a hypothesis Y occurs when we look for a string in the difference of Y and $\Gamma(Y)$. The size of DFA representing Y is bounded by n. Looking at $\Gamma(Y)$, it can be seen that the DFA representing the difference of Y and $\Gamma(Y)$ would be $O(nt)$. Thus the length of the longest string returned by an equivalence query is $m = O(nt)$.

The cost of answering membership queries dominates the total runtime cost of the procedure. Using $m = O(nt)$, the number of membership queries is $O(kn^2 + n \log nt)$. The cost of the membership queries is equal to the number of membership queries and the cost of building the DFA D_j representing $\Gamma^j(\emptyset)$. The cost for D_j is $(O(t))^j$ which leads to the total cost of membership queries of $O(t^{O(nt)} + kn^2 + n \log nt)$ (using maximum value of j to be $m = O(nt)$).

Hence, the overall running time is $O(f(t^{O(nt)} + kn^2 + n \log nt))$

4.5 Scalability Issues

In the previous section, we observed that the model checking algorithm presented here has running time that depends on an exponential factor $t^{O(nt)}$. This cost arises when constructing the DFA D_j that represents $\Gamma^j(\emptyset)$, where Γ could be either $\Gamma_{E[f_1 U f_2]}$ or Γ_{EGf}. Although, the size of D_j can grow exponentially, we have found that in practice, the size of the minimal DFA equivalent to D_j is manageable.

Another significant run-time cost is due to the size of the alphabet of the language. Although the run-time cost is just linear in the alphabet size, the problem is that the alphabet size itself grows exponentially in the number of variables in the system. We now discuss some of the techniques we use to manage this cost.

As mentioned in Section 2.4, when the learner introduces a new state in its hypothesis, it has to make membership queries to find the transition function from the new state. Let the access string for the new state being formed be s. For each symbol b, to find the destination for b-transition out of s, we have to sift sb down the classification tree. This requires membership queries for strings in $s.b.DS$. We first compute a DFA $\mathcal{A}^k$ representing $\Gamma(\emptyset) \cup \Gamma^2(\emptyset) \cup \ldots \cup \Gamma^k(\emptyset)$ (where Γ is either $\Gamma_{E[f_1 U f_2]}$ or Γ_{EGf}) up to a sufficient depth k such that all the membership queries for $s.\Sigma.DS$ can be answered by looking up acceptance in $\mathcal{A}^k$. A naive implementation would now need $O(|\Sigma|)$ membership queries; since $|\Sigma|$ is exponential in the number of variables, this is an unacceptable cost. To mitigate this, we can use a symbolic learning algorithm that symbolically sifts $s.\Sigma$ for the entire alphabet at one go, rather than individually sift sb for each $b \in \Sigma$. We present its details next.

Let the automaton $\mathcal{A}^k$ have states Q_A and transition function δ_A, and let $q_s \in Q_A$ be the state of $\mathcal{A}^k$ reached on reading s. Consider b_1 and b_2 such that $\delta_A(q_s, b_1) = \delta_A(q_s, b_2)$. Hence we know that for every $d \in DS$, $\delta_A(q_s, b_1.d) = \delta_A(q_s, b_2.d)$. Since $\mathcal{A}^k$ was chosen

to be such that all membership queries for strings in $s.\Sigma.DS$ can be faithfully answered by $\mathcal{A}^k$, it follows that for every $d \in DS$, $s.b_1.d$ belongs to the concept iff $s.b_2.d$ belongs to the concept. Thus, the strings $s.b_1$ and $s.b_2$ will sift in the same way in the classification. One important consequence of this observation is that one need not sift all strings in $s.\Sigma$; we only need to sift strings that go to different states in $\mathcal{A}^k$, which can be significantly less than that number of symbols in Σ. Furthermore, if the transition relation of all our DFAs are represented using ordered BDDs, then all this computation can be done implicitly and efficiently.

5 Implementation and Examples

We have extended our learning based verification tool LEVER [25] with the techniques presented in this paper. The tool takes as input a representation of the system in terms of its transitions and the initial states. Currently, systems with unbounded integers and parameterized systems are supported. The transition actions, transition guards and the description of the initial states and states corresponding to atomic propositions can be expressed using formulas in Presburger arithmetic. As in FAST [20], the global state of a system with n integer variables is represented by a string over the alphabet $\{0,1\}^n$.

Internally, we use the MONA tool [24] to store the transition relation as well as sets of states. MONA represents edges between states of an automata symbolically using BDDs. This allows us to use the symbolic method for answering membership queries for a new state in the hypothesis automata as discussed in Section 4.5. It is commonly seen that the BDD approach leads to a compact representation of the DFA even if the number of variables is large.,

For efficiency, we first calculate the set of reachable states and restrict all intermediate states calculated during fixpoint computations to be reachable.

5.1 Examples

We use the LEVER tool to analyze various examples taken from the literature. A number of these examples are taken from the FAST [20] web site and include cache coherence protocols such as *Dragon, Firefly, Illinois, MESI, MOESI, Berkeley, Futurebus* and *Synapse*; mutual exclusion protocols such as *peterson, lamport, ticket* and *bakery*; broadcast protocols such as *consistency*, and *producer-consumer*; petri nets such as *lastinfirstserved protocol, Esparza-Finkel-Mayr Counter Machine, RTP* and *manufacturing*; and counter machines such as *lift* and *barber*. We also analyze an example called *noaccel* for which the reachability set is regular but on which the *acceleration* methods employed in tools such as FAST cannot be applied because the transition relation does not satisfy the conditions needed for *acceleration*.

We analyze three different CTL formulas AGp, $AG(EFp)$ and $AG(\text{req} \rightarrow AF\text{resp})$. In this, AGp is a simple reachability property which says that the property p holds invariantly. $AG(EFp)$ implies that it is always true in all states that it possible to get to a state in future which satisfies p. This is often used to assert properties such as *proper termination*. Note that this is a branching time property that cannot be expressed in linear time logics. Finally, $AG(\text{req} \rightarrow AF\text{resp})$ illustrates the application to a liveness property which also requires fairness constraints. The fairness constraints were manually added to the examples taken from the literature but due to lack of time, this could be

done only for some of the more well known examples. Hence, $AG(\text{req} \rightarrow AF\text{resp})$ was not analyzed for all the examples.

In order to evaluate the performance of our tool with other available tools, we compare our LEVER tool with FAST [20] for the case of AGp. FAST was chosen because it has already assembled a large number of examples from various sources in the literature. While the comparison does not constitute an exhaustive analysis by any means, as shown in Table 1, LEVER outperforms FAST on a number of examples. Further, it is interesting to note that since LEVER only relies on regularity of the set being learnt, it is able to analyze the example *noaccel* trivially, while FAST is unable to make any headway (denoted in the table by the symbol $\uparrow$) since the transition relation does not lend itself to *acceleration* based methods. Note that the comparison with FAST is restricted to AGp properties since FAST can only handle reachability properties.

We plan to add comparisons of LEVER with other infinite state model checkers such as ALV [4] and TREX [28] in future.

Tables 1 and 2 report the running times on an Intel Xeon based Linux machine running at 1.70GHz with 2GB memory.

Example	AGp		$AG(EFp)$
	LEVER	FAST	LEVER
dragon	1.279s	1.058s	2.810s
firefly	0.849s	0.647s	1.269s
illinois	1.337s	0.714s	3.202s
mesi	0.748s	0.337s	1.381s
moesi	1.322s	0.417s	2.372s
prod/consumer	21.142s	0.316s	25.289s
synapse	0.192s	0.221s	0.543s
bakery2	7.019s	84.17s	21.567s
barber	1.780s	1.387s	2.199s
berkeley	0.265s	0.270s	0.558s
consistency	6.013s	142.6s	9.661s
efm	0.261s	0.591s	0.279s
futurbus	11.256s	1.599s	14.048s
lamport	1.433s	1.805s	2.081s
lastinfirstserved	13.860s	1.436s	15.910s
lift	7.445s	4.123s	11.678s
manufacturing	0.790s	2.331s	0.824s
noaccel	0.024s	$\uparrow$	0.025s
peterson	2.808s	3.448s	3.844s
rtp	1.015s	1.578s	15.984s
ticket2i	3.779s	0.669s	15.368s

Table 1. Running times for CTL formula AGp and $AG(EFp)$

Example	$AG(\text{req} \rightarrow AF\text{resp})$
	LEVER
bakery2	36.56s
barber	5.29s
consistency	8.03s
lamport	6.47s
lift	17.31s
ticket2i	45.92s

Table 2. Running times for CTL formula $AG(\text{req} \rightarrow AF\text{resp})$ with fairness constraint

6 Conclusions and Future Work

We presented a learning based CTL model checking algorithm, which is the first CTL model checker for infinite state systems that handles fairness constraints. We built a prototype model checker LEVER based on these ideas and demonstrated that on many practical examples the tool performs well.

The work presented here can be extended in many ways. We plan to compare the performance of LEVER with other infinite state model checkers such as ALV [4] and TREX [28] in future. Another interesting direction of future work would be develop a learning based model checker for the full μ-calculus. The main challenge is that μ-calculus allows to express fixpoints of functions with more than one argument variable; we will need to extend the ideas presented here that work for fixpoints with a single argument to this more general case.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *Computer-Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 236–248. Springer, 2003.
2. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *2nd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages*, 2005.
3. R. Alur, P. Madhusudan, and W. Nam. Symbolic compositional verification by learning assumptions. In *Proceedings of the 17th International Conference on Computer Aided Verification (to appear)*, 2005.
4. ALV. Action language verifier. `http://www.cs.ucsb.edu/~bultan/composite/`, 2004.
5. G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16, Jan. 2002.
6. D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, Nov. 1987.
7. A. Annichini, A. Bouajjani, and M. Sighireanu. TReX: a tool for reachability analysis of complex systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the international conference on computer aided verification (CAV'01),* Paris, France, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2001.
8. S. Bardin, A. Finkel, and J. Leroux. FASTer acceleration of counter automata in practice. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume

2988 of *Lecture Notes in Computer Science*, pages 576–590, Barcelona, Spain, Mar. 2004. Springer.
9. C. Bartzis and T. Bultan. Widening arithmetic automata. In *Computer Aided Verification, 2004.*, 2004.
10. B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Collection des Publications de la Faculté des Sciences Appliquées de l'Université de Liége, 1999.
11. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV: International Conference on Computer Aided Verification*, 2003.
12. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.
13. T. Bultan. *Automated symbolic analysis of reactive systems*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, Md., 1998.
14. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state programs using presburger arithmetic. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, pages 400–411, 1997.
15. T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 382–386, 2001.
16. E. Clarke, S. Chaki, N. Sinha, and P. Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proceedings of the 17th International Conference on Computer Aided Verification (to appear)*, 2005.
17. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. Number ISBN:0262032708. The MIT Press, 2000.
18. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 331–346, 2003.
19. G. Delzanno and A. Podelski. Model checking in CLP. *LNCS*, 1579:223–239, 1999.
20. FAST. Fast acceleration of symbolic transition systems. `http://www.lsv.ens-cachan.fr/fast/`, 2004.
21. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1–2):63–92, 2001.
22. P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In *Proc. of Infinity'04, London, UK*, 2004.
23. M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts, 1994.
24. N. Klarlund and A. Møller. Mona. `http://www.brics.dk/mona/`, 2004.
25. LEVER. Learning to verify tool. `http://osl.cs.uiuc.edu/~vardhan/lever.html`, 2004.
26. R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inform. Comput.*, 103(2):299–347, Apr. 1993.
27. T. Touili. Regular model checking using widening techniques. In *ENTCS*, volume 50. Elsevier, 2001.
28. TReX. A tool for reachability analysis of complex systems. `http://www.liafa.jussieu.fr/~sighirea/trex`, 2004.
29. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively learning to verify safety for FIFO automata. In *LNCS 3328, Proc. of FSTTCS'04, Chennai, India*, pages 494–505, 2004.
30. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *LNCS 3308, Proc. of ICFEM'04, Seattle, USA*, pages 274–288, 2004.
31. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440, pages 45–60, Edinburgh, UK, April 2005. Springer.