Abstract—We introduce a procedure for proving safety properties. This procedure is based on a technique called Partial Quantifier Elimination (PQE). In contrast to complete quantifier elimination, in PQE, only a part of the formula is taken out of the scope of quantifiers. So, PQE can be dramatically more efficient than complete quantifier elimination. The appeal of our procedure is twofold. First, it can prove a property without generating an inductive invariant. Second, it employs depth-first search and so can be used to find deep bugs.

I. INTRODUCTION

A. Motivation

Property checking is an important part of hardware verification. (In this paper, by property checking we mean verification of safety properties.) Arguably, the most popular way to prove a property is to build an inductive invariant. While the recent SAT-based methods for generating inductive invariants have enjoyed great success [1], [2], this approach has some problems. First, for some properties, compact inductive invariants may not exist. Second, even if such inductive invariants exist, they may be very hard to find. Third, if a property does not hold due to a deep bug, running an algorithm for building an inductive invariant may not be the best way to find this bug. In this paper, we describe a method for proving properties without generation of inductive invariants. This method employs depth-first search and so can be useful for finding deep bugs.

B. Partial quantifier elimination

This paper is a part of our effort to develop a technique called partial quantifier elimination (PQE)[3]. In contrast to regular (i.e. “complete”) quantifier elimination, only a part of the formula is taken out of the scope of quantifiers in PQE. The appeal of PQE is twofold. First, it provides a language for incremental computing. Second, PQE can be dramatically more efficient than complete quantifier elimination.

In our work on PQE we combine bottom-up and top-down approaches. The bottom-up part is to develop algorithms for efficiently solving PQE [3], [4], [5], [6]. The top-down part is to create PQE based methods for solving verification problems. For instance, we have described such methods for SAT solving [3], [7], [6], equivalence checking [8], model checking [9], testing [6], [10], checking the completeness of specification [11]. This paper is an addition to the top-down part of our research. Namely, it describes a PQE based algorithm for property checking that does not generate an inductive invariant.

C. Problem we consider

Let $\xi$ be a transition system specified by transition relation $T(S, S')$ and formula $I(S)$ describing initial states. Here $S$ and $S'$ are sets of variables specifying the present and next states respectively. Let $s$ be a state i.e. an assignment to $S$. Henceforth, by an assignment $q$ to a set of variables $Q$ we mean a complete assignment unless otherwise stated i.e. all variables of $Q$ are assigned in $q$.

Let $P(S)$ be a property of $\xi$. We will call a state $s$ a $P$-state if $P(s) = 1$. We will refer to a $\neg P$-state (i.e. a state where $P$ fails) as a bad state. The problem we consider is to check if a bad state is reachable in $\xi$. We will refer to this problem as the safety problem. Usually, the safety problem is solved by finding an inductive invariant i.e. a formula $K(S)$ such that $K \rightarrow P$ and $K(S) \land T(S, S') \rightarrow K(S')$.

D. Property checking without inductive invariants

In this paper, we consider an approach where the safety problem is solved without generation of an inductive invariant. We will refer to a system with initial states $I$ and transition relation $T$ as an $(I, T)$-system. Let $Diam(I, T)$ denote the reachability diameter of an $(I, T)$-system. That is $n = Diam(I, T)$ means that every state of this system is reachable in at most $n$ transitions. One can partition the problem of checking if property $P$ holds into two subproblems.

1) Find the value of $Diam(I, T)$.
2) Check if a $\neg P$-state is reachable in $n$ transitions where $n \leq Diam(I, T)$.

We describe a procedure called ProveProp that solves the two subproblems above. We will refer to the first subproblem as the RD problem where RD stands for Reachability Diameter. ProveProp is formulated in terms of PQE, which has the following advantages. First, the RD problem above is solved without generating the set of all reachable states. Second, the property $P$ is proved without generation of an inductive invariant. Third, due to using PQE, ProveProp performs depth-first search, which is beneficial for finding deep bugs. Importantly, as we mentioned above, PQE can be much more efficient than complete quantifier elimination.

To prove a property $P$ true, ProveProp has to consider traces of length up to $Diam(I, T)$. This may slow down property checking for systems with a large diameter. We describe a variation of ProveProp called ProveProp* that can prove a property without generation of long traces. ProveProp* achieves faster convergence by expanding the set of initial states with $P$-states i.e. by replacing $I$ with $I^{exp}$ such that $I \rightarrow I^{exp}$ and $I^{exp} \rightarrow P$. 

Property Checking Without Inductive Invariants

Eugene Goldberg

eu.goldberg@gmail.com
E. Contributions and structure of the paper

The contribution of this paper is twofold. First, we present a procedure for finding the reachability diameter without computing the set of all reachable states. Second, we describe a procedure for proving a property without generating an inductive invariant.

This paper is structured as follows. We recall PQE in Section II. Basic definitions and notation conventions are given in Section III. Section IV describes how one can look for a counterexample and solve the RD problem by PQE. In Section V, we show that PQE enables depth-first search in property checking. The ProveProp and ProveProp* procedures are presented in Sections VI and VII respectively. Section VIII provides some background. We make conclusions in Section IX.

II. PARTIAL QUANTIFIER ELIMINATION

In this paper, by a quantified formula we mean one with existential quantifiers. We assume that all formulas are propositional formulas in Conjunctive Normal Form (CNF). The latter is a conjunction of clauses, a clause being a disjunction of literals.

Given a quantified formula $\exists W[A(V,W)]$, the problem of quantifier elimination is to find a quantifier-free formula $A^*(V)$ such that $A^* \equiv \exists W[A]$. Given a quantified formula $\exists W[A(V,W) \land B(V,W)]$, the problem of Partial Quantifier Elimination (PQE) is to find a quantifier-free formula $A^*(V)$ such that $\exists W[A \land B] \equiv A^* \land \exists W[B]$. Note that formula $B$ remains quantified (hence the name partial quantifier elimination). We will say that formula $A^*$ is obtained by taking $A$ out of the scope of quantifiers in $\exists W[A \land B]$. We will call $A^*$ a solution to the PQE problem above.

Let $G(V)$ be a formula implied by $B$. Then $\exists W[A \land B] \equiv A^* \land G \land \exists W[B]$ implies that $\exists W[A \land B] \equiv A^* \land \exists W[B]$ . In other words, clauses implied by the formula that remains quantified are noise and can be removed from a solution to the PQE problem. So, when building $A^*$ by resolution it is sufficient to use only the resolvents that are descendants of clauses of $A$. For that reason, in the case formula $A$ is much smaller than $B$, PQE can be dramatically faster than complete quantifier elimination. In this paper, we do not discuss PQE solving. This information can be found in [3], [6].

III. DEFINITIONS AND NOTATION

A. Basic definitions

Definition 1: Let $\xi$ be an $(I,T)$-system. An assignment $s$ to state variables $S$ is called a state. A sequence of states $(s_0, \ldots, s_n)$ is called a trace. This trace is called valid if
- $I(s_0) = 1$,
- $T(s_i, s_{i+1}) = 1$ where $i = 0, \ldots, n - 1$.

Henceforth, we will drop the word valid if it is obvious from the context whether a trace is valid.

Definition 2: Let $(s_0, \ldots, s_n)$ be a valid trace of an $(I,T)$-system. State $s_n$ is said to be reachable in this system in $n$ transitions.

Definition 3: Let $\xi$ be an $(I,T)$-system and $P$ be a property to be checked. Let $(s_0, \ldots, s_n)$ be a valid trace such that
- every state $s_i$, $i = 0, \ldots, n-1$ is a $P$-state i.e. $P(s_i) = 1$.
- state $s_n$ is a $P^\perp$-state i.e. $P(s_n) = 0$.

Then this trace is called a counterexample for property $P$.

Remark 1: We will use the notions of a CNF formula $C_1 \land \ldots \land C_p$ and the set of clauses $\{C_1, \ldots, C_p\}$ interchangeably. In particular, the fact that $F$ has no clauses (i.e. $F = \emptyset$) also means $F \equiv 1$ and vice versa.

B. Some notation conventions

- $S_j$ denotes the state variables of $j$-th time frame.
- $S_j$ denotes $S_0 \cup \ldots \cup S_j$.
- $T_{j,j+1}$ denotes $T(S_j, S_{j+1})$.
- $T_j$ denotes $T_{0,1} \land \ldots \land T_{j-1,j}$.
- $I_0$ and $I_1$ denote $I(S_0)$ and $I(S_1)$ respectively.

IV. PROPERTY CHECKING BY PQE

In this section, we explain how the ProveProp procedure described in this paper proves a property without generating an inductive invariant. This is achieved by reducing the RD problem and the problem of finding a bad state to PQE. To simplify exposition, we consider systems with stuttering. This topic is discussed in Subsection IV-A. There we also explain how one can introduce stuttering by a minor modification of the system. The main idea of ProveProp and two propositions on which it is based are given in Subsection IV-B.

A. Stuttering

Let $\xi$ denote an $(I,T)$-system. The ProveProp procedure we describe in this paper is based on the assumption that $\xi$ has the stuttering feature. This means that $T(s,s)=1$ for every state $s$ and so $\xi$ can stay in any given state arbitrarily long. If $\xi$ does not have this feature, one can introduce stuttering by adding a combinational input variable $v$. The modified system $\xi$ works as before if $v=1$ and remains in its current state if $v=0$. (For the sake of simplicity, we assume that $\xi$ has only sequential variables. However, one can easily extend explanation to the case where $\xi$ has combinational variables.)

On the one hand, introduction of stuttering does not affect the reachability of a bad state and does not affect the value of $Diam(I,T)$. On the other hand, stuttering guarantees that $\xi$ has two nice properties. First, $\exists S[T(S,S')] = 1$ holds since for every next state $s'$, there is a “stuttering transition” from $s$ to $s'$ where $s$ and $s'$ coincide. Second, if a state is unreachable in $\xi$ in $n$ transitions it is also unreachable in $m$ transitions if $m < n$. Conversely, if a state is reachable in $\xi$ in $n$ transitions, it is also reachable in $m$ transitions where $m > n$.

B. Solving the RD problem and finding a bad state by PQE

As we mentioned in the introduction, one can reduce property checking to solving the RD problem and checking whether a bad state is reachable in $n$ transitions where $n \leq Diam(I,T)$. In this subsection, we show that one can solve these two problems by PQE.
The RD problem is to compute $\text{Diam}(I, T)$. It reduces to finding the smallest $n$ such that the sets of states reachable in $n$ and $(n+1)$ transitions are identical. The latter, as Proposition 1 below shows, comes down to checking if formula $I_1$ is redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$ i.e. whether $\exists S_n[I_1] = \exists S_n[I_0 \land T_{n+1}]$. If $I_1$ is redundant, then $\text{Diam}(I, T) \leq n$. This is a special case of the PQE problem. Instead, of finding a formula $H(S_{n+1})$ such that $\exists S_n[I_0 \land I_1 \land T_{n+1}] \equiv H \land \exists S_n[I_0 \land T_{n+1}]$, one just needs to decide if a tautological formula $H$ (i.e. $H \equiv 1$) is a solution to the PQE problem above.

Here is an informal explanation of why redundancy of $I_1$ is equivalent to $\exists S_n[I_0 \land I_1 \land T_{n+1}]$. Namely, $\exists$ (see Proposition 3 of the appendix) and condition b) proves $I_1$ is not redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$. Hence it facilitates finding a bad state. Hence it facilitates finding a bad state. Hence it facilitates finding a bad state.

Proposition 1: Let $\xi$ be an $(I, T)$-system. Then $\text{Diam}(I, T) \leq n$ iff formula $I_1$ is redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$ (i.e. iff $\exists S_n[I_0 \land T_{n+1}] = \exists S_n[I_0 \land I_1 \land T_{n+1}]$).

Proposition 2 below shows that one can look for bugs by checking if $I_1$ is redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1} \land T]$ i.e. similarly to solving the RD problem.

Proposition 2: Let $\xi$ be an $(I, T)$-system and $P$ be a property of $\xi$. No $\overline{T}$-state is reachable in $(n+1)$-th time frame for the first time iff $I_1$ is redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1} \land \overline{T}]$.

V. DEPTH-FIRST SEARCH BY PQE

In this section, we demonstrate that Proposition 1 enables depth-first search when solving the RD-problem. Namely, we show that one can prove that $\text{Diam}(I, T) > n$ without generation of all states reachable in $n$ transitions. In a similar manner, one can show that Proposition 2 enables depth-first search when looking for a bad state. Hence it facilitates finding deep bugs.

Proposition 1 entails that proving $\text{Diam}(I, T) > n$ comes down to showing that $I_1$ is not redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$. This can be done by:

a) generating a clause $C(S_{n+1})$ implied by $I_0 \land I_1 \land T_{n+1}$

b) finding a trace that satisfies $I_0 \land T_{n+1} \land C$

Let $(s_0, \ldots, s_{n+1})$ be a trace satisfying $I_0 \land T_{n+1} \land \overline{C}$. On the one hand, conditions a) and b) guarantee that $\exists S_n[I_0 \land T_{n+1}]$ is not redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$ under $s_{n+1}$. So $I_1$ is not redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$. On the other hand, condition a) shows that $s_{n+1}$ is not reachable in $n$ transitions (see Proposition 3 of the appendix) and condition b) proves $s_{n+1}$ reachable in $(n+1)$-transitions.

Note that satisfying conditions a) and b) above does not require breadth-first search i.e. computing the set of all states reachable in $n$ transitions. In particular, clause $C$ of condition a) can be found by taking $I_1$ out of the scope of quantifiers in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$ i.e. by solving the PQE problem. In the context of PQE-solving, condition b) requires $C$ not to be a “noise” clause implied by $I_0 \land T_{n+1} \land \overline{C}$.

VI. PROVEPROP PROCEDURE

In this section, we describe procedure ProveProp. As we mentioned in Subsection 1-D, when proving that a safety property $P$ holds, ProveProp solves the two problems below.

1) Find the value of $\text{Diam}(I, T)$ (i.e. solve the RD problem).

2) Check that no $\overline{T}$-state is reachable in $n$ transitions where $n \leq \text{Diam}(I, T)$.

ProveProp returns a counterexample if $P$ does not hold, or the value of $\text{Diam}(I, T)$ if $P$ holds. To simply find the value of $\text{Diam}(I, T)$, one can call ProveProp with the trivial property $P$ (i.e. $P \equiv 1$).

A description of how ProveProp solves the RD problem is given in Subsection VI-A. Solving the RD problem is accompanied in ProveProp by checking if a bad state is reached. That is the problems above are solved by ProveProp together. A description of how ProveProp checks if a counterexample exists is given in Subsection VI-B. The pseudo-code of ProveProp is described in Subsections VI-C and VI-D.

A. FINDING DIAMETER

Proposition 1 entails that proving $\text{Diam}(I, T) > n$ comes down to showing that formula $I_1$ is redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$. To this end, ProveProp builds formulas $H_1, \ldots, H_n$. Here $H_1$ is a subset of clauses of $I_1$ and formula $H_i(S_i)$, $i = 2, \ldots, n$ is obtained by resolving clauses of $I_0 \land I_1 \land T_i$. One can view formulas $H_3, \ldots, H_n$ as a result of “pushing” clauses of $I_1$ and their descendants (obtained by resolution) to later time frames. The main property satisfied by these formulas is that $\exists S_{n-1}[I_0 \land I_1 \land T_n] \equiv H_n \land \exists S_{n-1}[I_0 \land \overline{H}_{n-1} \land T_n]$ where $\overline{H_{n-1}} = H_1 \land \cdots \land H_{n-2}$.

ProveProp starts with $n = 1$ and $H_1 = I_1$. Then it picks a clause $C$ of $H_1$ and finds formula $H_2(S_2)$ such that $\exists S_1[I_0 \land H_1 \land C \land T_2] \equiv H_2 \land \exists S_1[I_0 \land H_1 \land T_2]$ where $H_1^i = H_1 \land \{C\}$. Formula $H_1$ is replaced with $H_1^i$. Computing $H_2$ can be viewed as pushing $C$ to the second time frame. If $H_2 \equiv 1$, ProveProp picks another clause of $H_1$ and computes $H_2$ for this clause. If $H_2 \equiv 1$ for every clause of $H_1$, then eventually $H_1$ becomes empty (and so $H_1 \equiv 1$). This means that $I_1$ is redundant in $\exists S_1[I_0 \land H_1 \land T_2]$ and $\text{Diam}(I, T) \leq 2$. If $H_2$ is not empty, then ProveProp picks a clause $C$ of $H_2$ and builds formula $H_3(S_3)$ such that $\exists S_2[I_0 \land H_1 \land H_2 \land C \land T_3] \equiv H_3 \land \exists S_1[I_0 \land H_1 \land H_2 \land T_3]$ where $H_2 = H_2 \land \{C\}$. Formula $H_2$ is replaced with $H_3$. If $H_3 \equiv 1$ for every clause of $H_2$, then $H_2$ becomes empty and ProveProp picks a new clause of $H_1$. This goes on until all descendants of $I_1$ proved redundant.

The procedure above is based on the observation that if $n = \text{Diam}(I, T)$, any clause $C(S_{n+1})$ that is a descendant of $I_1$ is implied by $I_0 \land T_{n+1}$. So one does not need to add
clauses depending on $S_{n+1}$ to make formula $H_n$ redundant. In other words, pushing the descendants of $I_1$ to later time frames inevitably results in making them redundant. The value of $Diam(I, T)$ is given by the largest index $n$ among the time frames where a descendant of $I_1$ was not redundant yet.

B. Finding a counterexample

Every time $ProveProp$ replaces a clause of $H_n(S_n)$ with formula $H_{n+1}(S_{n+1})$, it checks if a $\mathcal{P}$-state is reached. This is done as follows. Recall that $H_{n+1}$ satisfies $\exists S_n[I_0 \land I_1 \land T_{n+1}] \equiv H_n \land \exists S_n[I_0 \land \mathbb{H}_n \land T_{n+1}]$. Let $C$ be a clause of $H_{n+1}$. Assume for the sake of simplicity that $ProveProp$ employs a noise-free PQE-solver. Then clause $C$ is derived only if it is implied by $I_0 \land I_1 \land T_{n+1}$ but not $I_0 \land T_{n+1}$. This means that the very fact that $C$ is derived guarantees that there is some state that is reached in $(n+1)$-th iteration for the first time. So if a $\mathcal{P}$-state $s$ falsifies $C$, there is a chance that $s$ is reachable. Now, suppose that $H_{n+1}$ is generated by a PQE-solver that may generate noise clauses but the amount of noise is small. In this case, generation of clause $C$ above still implies that there is a significant probability of $s$ being reachable.

A bad state falsifying $C$ is generated by $ProveProp$ as an assignment satisfying $C \land \mathcal{P} \land R_{n+1}$. Here $R_{n+1}$ is a formula meant to help to exclude states that are unreachable in $n+1$ transitions. Originally, $R_{n+1}$ is empty. Every time a clause implied by $I_0 \land T_{n+1}$ is derived, it is added to $R_{n+1}$. To find out if $s$ is indeed reachable, one needs to check the satisfiability of formula $I_0 \land T_{n+1} \land \mathcal{P}_s$ where $\mathcal{P}_s$ is the longest clause falsified by $s$. An assignment satisfying this formula is a counterexample. If this formula is unsatisfiable, the SAT-solver returns a clause $C^*$ implied by $I_0 \land T_{n+1}$ and falsified by $s$. This clause is added to $R_{n+1}$ and $ProveProp$ looks for a new state satisfying $C \land \mathcal{P} \land R_{n+1}$. If another bad state $s$ is found, $ProveProp$ proceeds as above. Otherwise, $C$ does not specify any bad states reachable in $(n+1)$ transitions. Then $ProveProp$ picks a new clause of $H_{n+1}$ to check if it specifies a reachable bad state.

C. Description of $ProveProp$

Pseudo-code of $ProveProp$ is given in Figure 1. $ProveProp$ accepts formulas $I$, $T$, and $P$ specifying initial states, transition relation and the property to be verified respectively. $ProveProp$ returns a counterexample if $P$ does not hold, or $Diam(I, T)$ if $P$ holds. As we mentioned above, to simply compute the value of $Diam(I, T)$, it suffices to call $ProveProp$ with the trivial property $P$ that is always true.

$ProveProp$ consists of three parts separated by the dotted line. The first part (lines 1-6) starts with modifying the transition relation to introduce stuttering (see Subsection IV-A). Then $ProveProp$ checks if a bad state is reachable in one transition (lines 2-3). After that, $ProveProp$ sets formula $H_1$ to $I_1$ and parameter $n$ to 1 (lines 4-5). The parameter $n$ stores the index of the latest time frame where the corresponding formula $H_n$ is not empty. $ProveProp$ concludes the first part by setting the value of the diameter to 1 (line 6).

```
// $R_n = R_1 \land \cdots \land R_n$
// $ProveProp(I, T, P)\{
1 T := MakeStutter(T)
2 Cex := Unsat(I_0 \land \mathbb{T}_{0,1} \land T)
3 if (Cex \neq nil) return(Cex, nil)
4 H_1 := I_1
5 n := 1
6 Diam := 1
7 while (H_1 \neq 1) \{ 
8 \quad if (H_n \equiv 1) \{ 
9 \quad \quad n := n - 1
10 \quad \quad continue \}
11 \quad if (FirstVisit(n+1)) R_{n+1} := 1
12 \quad C := PickClause(H_n)
13 \quad H_n := H_n \\setminus \{C\}
14 \quad H_{n+1} := PQE(\exists S_n[I_0 \land C \land \mathbb{H}_n \land T_{n+1}])
15 \quad RemNoise(H_{n+1}, R_{n+1}, I, T)
16 \quad if (H_{n+1} \equiv 1) continue
17 \quad Cex := ChkBadSt(H_{n+1}, R_{n+1}, I, T, P)
18 \quad if (Cex \neq nil) return(Cex, nil)
19 \quad n := n + 1
20 \quad if (Diam < n) Diam := n \}
21 return(nil, Diam) \}
```

Fig. 1. The $ProveProp$ procedure

The second part consists of a while loop (lines 7-20). In this loop, $ProveProp$ pushes formula $I_1$ and its descendants to later time frames. This part consists of three pieces separated by vertical spaces. The first piece (lines 8-11) starts by checking if formula $H_n$ has no clauses (and so $H_n \equiv 1$). If this is the case, then all descendants of $H_n$ have been proved redundant. So $ProveProp$ decreases the value of $n$ by 1 and starts a new iteration. If $H_{n+1} \neq 1$, $ProveProp$ checks if $(n+1)$-th time frame is visited for the first time. If so, $ProveProp$ sets formula $R_{n+1}$ to 1. As we mentioned in the previous subsection, $R_{n+1}$ is used to accumulate clauses implied by $I_0 \land T_{n+1}$.

$ProveProp$ starts the second piece of the while loop (lines 12-14) by picking a clause $C$ of formula $H_n$ and removing it from $H_n$. After that, $ProveProp$ builds formula $H_{n+1}$ such that $\exists S_n[I_0 \land \mathbb{H}_n \land C \land T_{n+1}] \equiv H_{n+1} \land \exists S_n[I_0 \land \mathbb{H}_n \land T_{n+1}]$.

In the third piece, (lines 15-20), $ProveProp$ analyzes formula $H_{n+1}$. First, it calls procedure $RemNoise$ described in the next subsection. It drops noise clauses of $H_{n+1}$ i.e. ones implied by $I_0 \land T_{n+1}$. If the resulting formula $H_{n+1}$ is empty, $ProveProp$ starts a new iteration. Otherwise, $ProveProp$ calls procedure $ChkBadSt$ also described in the next subsection. $ChkBadSt$ checks if clauses of $H_{n+1}$ exclude a bad state reachable in $n+1$ transitions. If not, i.e. if no counterexample is found, $ProveProp$ increments the value of $n$ by 1. If the value of $n$ is greater than the current diameter $Diam$, the latter is set to $n$ (line 20). After that a new iteration begins.

The third part of $ProveProp$ consists of line 21.
addresses this problem. In particular, describe a variation of transition systems with a large diameter. In this section, we reachability diameter. This strategy may be inefficient for in Section VI has to examine traces of length up to the checks if formula
\[ I, T \]

gets to this line if
\[ ProveProp \]

is proved redundant and no
\[ Cex \]

bad state is reachable in \( Diam(I, T) \) transitions. This means that property \( P \) holds and \( ProveProp \) returns the value of \( Diam(I, T) \).

D. Description of RemNoise and ChkBadSt procedures

Pseudo-code of RemNoise is given in Figure 2. The objective of RemNoise is to remove noise clauses of \( H_i \) i.e. ones implied by \( I_0 \land \exists \). So for every clause \( C \) of \( H_i \), RemNoise checks if formula \( I_0 \land \exists \land \forall \land C \) is satisfiable. (Here \( \exists \) denotes \( R_i \land \exists \land R_i \). It specifies clauses implied by \( I_0 \land \exists \) that have been generated earlier.) If the formula above is unsatisfiable, \( C \) is removed from \( H_i \) and added to \( R_i \).

Pseudo-code of ChkBadSt is given in Figure 3. It checks if a clause of \( H_i \) specifies a bad state reachable in \( i \) transitions for the first time. The idea of ChkBadSt was described in Subsection VI-B. ChkBadSt consists of two nested loops. In the outer loop, ChkBadSt enumerates clauses of \( H_i \). In the inner loop, ChkBadSt checks if a bad state \( s \) satisfying formula \( \forall \land \exists \land R_i \) is reachable in \( i \) transitions. The inner loop iterates until this formula becomes unsatisfiable.

Finding out if \( s \) is reachable in \( i \) transitions comes down to checking the satisfiability of formula \( I_0 \land \exists \land \forall \land \exists \). (Here \( \exists \) is the longest clause falsified by \( s \).) An assignment satisfying this formula specifies a counterexample. If this formula is unsatisfiable, a clause \( C^*(S_i) \) is returned that is implied by \( I_0 \land \exists \) and falsified by \( s \). This clause is added to \( R_i \) and a new iteration of the inner loop begins.

VII. THE PROVEPROP* PROCEDURE

When a property holds, the \( ProveProp \) procedure described in Section VI has to examine traces of length up to the reachability diameter. This strategy may be inefficient for transition systems with a large diameter. In this section, we describe a variation of \( ProveProp \) called \( ProveProp* \) that addresses this problem. In particular, \( ProveProp* \) can prove a property by examining traces that are much shorter than the diameter. The main idea of \( ProveProp* \) is to expand the set of initial states by adding \( P \)-states that may not be reachable at all. So faster convergence is achieved by expanding the set of allowed behaviors. This is similar to boosting the performance of existing methods of property checking by looking for a weaker inductive invariant (as opposed to building the strongest inductive invariant satisfied only by reachable states).

The pseudo-code of \( ProveProp* \) is given in Figure 4. It consists of two parts separated by the dotted line. \( ProveProp* \) starts the first part (lines 1-4) by introducing stuttering. Then it checks if there is a bad state reachable in one transition. Finally, it generates a formula \( I^{exp} \) specifying an expanded set of initial states that satisfies \( I \rightarrow I^{exp} \) and \( I^{exp} \rightarrow P \) (line 4). Here \( I \) is the initial set of states and \( P \) is the property to be proved. A straightforward way to generate \( I^{exp} \) is to simply set it to \( P \).

The second part (lines 5-12) consists of a while loop. In this loop, \( ProveProp* \) repeatedly calls the \( ProveProp \) procedure described in Section VI (line 6). It returns \( Cex, Diam \). If \( Cex = nil \), property \( P \) holds and \( ProveProp* \) returns \( nil \) (line 12). Otherwise, \( ProveProp* \) analyzes the counterexample \( Cex = (s_0, . . . , s_n) \) returned by \( ProveProp \) (lines 7-11). If state \( s_0 \) of \( Cex \), satisfies \( I \), then \( P \) does not hold and \( ProveProp* \) returns \( Cex \) as a counterexample (line 9). If \( I(s_0) = 0 \), \( ProveProp* \) excludes \( s_0 \) by conjoining \( I^{exp} \) with a clause \( C \) such that \( C(s_0) = 0 \) and \( I \rightarrow C \). Then \( ProveProp* \) starts a new iteration. When constructing clause \( C \) it makes sense to analyze \( Cex \) to find other states of \( I^{exp} \) to be excluded. Suppose, for instance, that one can easily prove that state \( s_1 \) of \( Cex \) can be reached from a state \( s_0^* \) such that \( I^{exp}(s_0^*) = 1 \), \( I(s_0^*) = 0 \) and \( s_0^* \neq s_0 \). Then one may try to pick clause \( C \) so that it is falsified by both \( s_0 \) and \( s_0^* \).

\( ProveProp* \) is a complete procedure i.e. it eventually proves \( P \) or finds a counterexample.

VIII. SOME BACKGROUND

The first methods of property checking were based on BDDs and computed the set of reachable states [12]. Since BDDs frequently get prohibitively large, SAT-based methods
of property checking have been introduced. Some of them, like interpolation [1] and IC3 [2] have achieved a great boost in performance. Among incomplete SAT-based methods (that can do only bug hunting), Bounded Model Checking (BMC) [13] has enjoyed a lot of success.

As we mentioned in the introduction, the problem with inductive invariants is that they can be too large to generate or too hard to find. Besides, if a property is false due to a deep bug, looking for an inductive invariant may not be the best strategy to find this bug. After the introduction of PQE [3], we formulated a few approaches addressing the problems above. In particular, in [10], we described a PQE-based procedure for property checking meant for finding deep bugs. However, that procedure was incomplete. Here, we continue this line of research. Similarly to the procedure of [10], ProveProp performs depth-first search meant to facilitate finding deep bugs. However, in contrast to the former, ProveProp is complete.

The idea of proving a property without generating an inductive invariant is not new. For instance, earlier it was proposed to combine BMC with finding a recurrence diameter [14]. The latter is equal to the length of the longest trace that does not repeat a state. Obviously, the recurrence diameter is larger or equal to the reachability diameter. In particular, the former can be drastically larger than the latter. In this case, finding the recurrence diameter is of no use.

IX. CONCLUSIONS

In this paper, we present ProveProp, a new procedure for checking safety properties. It is based on a technique called Partial Quantifier Elimination (PQE). In contrast to regular quantifier elimination, in PQE, only a small part of the formula is taken out of the scope of quantifiers. In [4], [5], [6], we developed the machinery of redundancy based reasoning meant for building efficient PQE solvers. The advantage of ProveProp is twofold. First, it can prove that a property holds without generation of an inductive invariant. This can be very useful when inductive invariants are prohibitively large or are hard to find. Second, ProveProp performs depth-first search and so can be used for finding deep bugs.

REFERENCES

[1] K. L. McMillan, “Interpolation and sat-based model checking,” in CAV-03. Springer, 2003, pp. 1-13.
[2] A. Bradley, “Sat-based model checking without unrolling,” in VMCAI, 2011, pp. 70-87.
[3] E. Goldberg and P. Manolios, “Partial quantifier elimination,” in Proc. of HVC-14. Springer-Verlag, 2014, pp. 148-164.
[4] E. Goldberg and P. Manolios, “Quantifier elimination by dependency sequents,” in FMCAD-12, 2012, pp. 34-44.
[5] E. Goldberg and P. Manolios, “Quantifier elimination via clause redundancy,” in FMCAD-13, 2013, pp. 85-92.
[6] E. Goldberg, “Partial quantifier elimination by certificate clauses,” Tech. Rep. arXiv:2003.09667 [cs.LO], 2020.
[7] E. Goldberg and P. Manolios, “Software for quantifier elimination in propositional logic,” in ICMS-2014, Seoul, 2014, pp. 291-294.
[8] E. Goldberg, “Equivalence checking by logic relaxation,” in FMCAD-16, 2016, pp. 49-56.
[9] E. Goldberg, “Property checking by logic relaxation,” Tech. Rep. arXiv:1601.02742 [cs.LO], 2016.
[10] E. Goldberg and P. Manolios, “Bug hunting by computing range reduction,” Tech. Rep. arXiv:1408.7039 [cs.LO], 2014.
[11] E. Goldberg, “Generation of a complete set of properties,” Tech. Rep. arXiv:2004.05853 [cs.LO], 2020.
[12] K. McMillan, Symbolic Model Checking. Norwell, MA, USA: Kluwer Academic Publishers, 1993.
[13] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, “Symbolic model checking using sat procedures instead of bdds,” in DAC, 1999, pp. 317-320.
[14] D. Kroening and O. Strichman, “Efficient computation of recurrence diameters,” in VMCAI-2003, Lecture Notes in Computer Science, vol. 2575, 2003, pp. 298-309.

APPENDIX

Lemma 1 below is used in proving Proposition 1.

Lemma 1: Let $\xi$ be an $(I, T)$-system. Then $Diam(I, T) \leq n$ if $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$ where $n \geq 0$.

Proof: If part: Given $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$, let us prove $Diam(I, T) \leq n$. Assume the contrary, i.e. $Diam(I, T) > n$. Then there is a state $a_{n+1}$ reachable only in $(n+1)$-th time frame. Hence, there is a trace $t_n = (a_0, \ldots, a_{n+1})$ satisfying $I_0 \land T_{n+1}$. Then due to $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$ there exists a trace $t_b = (b_0, \ldots, b_{n+1})$ satisfying $I_1 \land T_{n+1}$ where $b_{n+1} = a_{n+1}$.

Let $t_b$ be a trace $c_0, \ldots, c_n$ where $c_i = b_i+1, i = 0, \ldots, n$. The fact that $t_b$ satisfies $I_1 \land T_{n+1}$ implies that $t_b$ satisfies $I_0 \land T_n$. Since $c_n = b_{n+1} = a_{n+1}$, state $a_{n+1}$ is reachable in $n$ transitions. So we have a contradiction.

Only if part: Given $Diam(I, T) \leq n$, let us prove that $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$.

First, let us show that $\exists S_n[I_0 \land T_{n+1}]$ implies $\exists S_n[I_1 \land T_{n+1}]$. Let $\exists S_n[I_0 \land T_{n+1}]$ be under an assignment $s_n+1$ to $S_n+1$. Then the state $s_n+1$ is reachable in $n + 1$ transitions. Since $Diam(I, T) \leq n$, there has to be a trace $t_a = (a_0, \ldots, a_k)$ where $k \leq n$ and $a_k = s_n+1$. Let $m$ be equal to $n + 1 - k$. Let $t_b = (b_0, \ldots, b_{n+1})$ be a trace defined as follows: $b_i = a_i$, $i = 0, \ldots, m$, and $b_i = a_i - m$, $i = m + 1, \ldots, n + 1$. Due to the stuttering feature of $\xi$, the trace $t_b$ satisfies $I_1 \land T_{n+1}$ and $b_{n+1}+1 = s_{n+1}$. So, $\exists S_n[I_1 \land T_{n+1}]$ is under the assignment $s_n+1$ to $S_n+1$.

Now, we show that $\exists S_n[I_1 \land T_{n+1}]$ implies $\exists S_n[I_0 \land T_{n+1}]$. Let $\exists S_n[I_1 \land T_{n+1}]$ be under an assignment $s_n+1$ to $S_n+1$. Then $s_n+1$ is reachable in $n$ transitions. Due to the stuttering feature of $\xi$, the state $s_n+1$ is also reachable in $n + 1$ transitions. So, a trace $(s_0, \ldots, s_{n+1})$ satisfies $I_0 \land T_{n+1}$. Hence, $\exists S_n[I_0 \land T_{n+1}]$ is under the assignment $s_{n+1}$.

Proposition 1: Let $\xi$ be an $(I, T)$-system. Then $Diam(I, T) \leq n$ if formula $I_1$ is redundant in $\exists S_n[I_0 \land T_{n+1}]$ (i.e. iff $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$).

Proof: Lemma 1 entails that to prove the proposition at hand it is sufficient to show that $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$ if formula $I_1$ is redundant in $\exists S_n[I_0 \land T_{n+1}]$.

If part: Given $I_1$ is redundant in $\exists S_n[I_0 \land T_{n+1}]$, let us show that $\exists S_n[I_0 \land T_{n+1}] \equiv \exists S_n[I_1 \land T_{n+1}]$. Redundancy of $I_1$ means that $\exists S_n[I_0 \land I_1 \land T_{n+1}] \equiv \exists S_n[I_0 \land T_{n+1}]$. Let us show that $I_0$ is redundant in $\exists S_n[I_0 \land I_1 \land T_{n+1}]$ and hence $\exists S_n[I_1 \land T_{n+1}] \equiv \exists S_n[I_0 \land T_{n+1}]$. Assume the contrary i.e. $I_0$ is not redundant and hence $\exists S_n[I_0 \land I_1 \land T_{n+1}]$
for $\exists S_{n+1}$ reduces to $\exists I_0 \land T_{n+1} \land \neg P$. Then there is an assignment $s_{n+1}$ to variables of $S_{n+1}$ for which $\exists S_{n}[I_0 \land T_{n+1}] = 1$ and $\exists S_{n}[I_0 \land I_1 \land T_{n+1}] = 0$. (The opposite is not possible since $I_0 \land I_1 \land T_{n+1}$ implies $I_1 \land T_{n+1}$.) This means that

- there is a valid trace $t_0 = (a_0, \ldots, a_{n+1})$ where $a_1$ satisfies $I_1$ and $a_{n+1} = s_{n+1}$.
- there is no trace $t_0 = (b_0, \ldots, b_{n+1})$ where $b_0$ satisfies $I_0$, $b_1$ satisfies $I_1$ and $b_{n+1} = s_{n+1}$.

Let us pick $t_0$ as follows. Let $b_0 = a_k$ for $1 \leq k \leq n+1$ and $b_0 = b_1$. Let us show that $t_0$ satisfies $I_0 \land I_1 \land T_{n+1}$ and so we have a contradiction. Indeed, $b_0$ satisfies $I_0$ because $b_1$ satisfies $I_1$ and $b_0 = b_1$. Besides, $(b_0, b_1)$ satisfies $T_{0,1}$ because the system at hand has the stuttering feature. Hence $t_0$ satisfies $I_0 \land I_1 \land T_{n+1}$.

**Only if part:** Given $\exists S_{n}[I_0 \land T_{n+1}] \subseteq \exists S_{n}[I_0 \land I_1 \land T_{n+1}]$, let us show that $I_1$ is redundant in $\exists S_{n}[I_0 \land I_1 \land T_{n+1}]$. Assume the contrary i.e. $\exists S_{n}[I_0 \land I_1 \land T_{n+1}] \neq \exists S_{n}[I_0 \land I_1 \land T_{n+1}]$. Then there is an assignment $s_{n+1}$ to variables of $S_{n+1}$ such that $\exists S_{n}[I_0 \land T_{n+1}] = 1$ and $\exists S_{n}[I_0 \land I_1 \land T_{n+1}] = 0$. This means that

- there is a valid trace $t_o = (a_0, \ldots, a_{n+1})$ where $a_0$ satisfies $I_0$ and $a_{n+1} = s_{n+1}$.
- there is no trace $t_0 = (b_0, \ldots, b_{n+1})$ where $b_0$ satisfies $I_0$, $b_1$ satisfies $I_1$ and $b_{n+1} = s_{n+1}$.

Let us show that then $\exists S_{n}[I_1 \land T_{n+1}]$ evaluates to 0 for $s_{n+1}$. Indeed, assume the contrary i.e. there is an assignment $t_o = (c_0, \ldots, c_{n+1})$ satisfying $I_1 \land T_{n+1}$ where $c_1$ satisfies $I_1$ and $c_{n+1} = s_{n+1}$. Let $t_o = (d_0, \ldots, d_{n+1})$ be obtained from $t_o$ as follows: $d_0 = d_1 = c_1$, $d_i = c_i$, $i = 2, \ldots, n+1$. Then $t_o$ satisfies $I_0 \land I_1 \land T_{n+1}$ which contradicts the claim above that there is no trace $t_0$. Hence, $\exists S_{n}[I_0 \land T_{n+1}] = 1$ and $\exists S_{n}[I_0 \land T_{n}] = 0$ under assignment $s_{n+1}$. So we have a contradiction.

**Proposition 2:** Let $\xi$ be an $(I, T)$-system and $P$ be a property of $\xi$. No $\neg P$ state is reachable in $(n+1)$-th time frame for the first time iff $I_1$ is redundant in $\exists S_{n}[I_0 \land I_1 \land T_{n+1} \land \neg P]$.

**Proof:** If part: Assume the contrary i.e. $I_1$ is redundant in $\exists S_{n}[I_0 \land I_1 \land T_{n+1} \land \neg P]$ but there is a bad state $s_{n+1}$ that is reachable in $(n+1)$-th time frame for the first time. Then there is an assignment $a_0 = (a_0, \ldots, a_{n+1})$ satisfying $I_0 \land T_{n+1} \land \neg P$ where $a_{n+1} = s_{n+1}$. Redundancy of $I_1$ means that $\exists S_{n}[I_0 \land I_1 \land T_{n+1} \land \neg P] = \exists S_{n}[I_0 \land T_{n+1} \land \neg P]$. Then there is an assignment $b_0 = (b_0, \ldots, b_{n+1})$ where $b_{n+1} = s_{n+1}$ that satisfies $I_0 \land I_1 \land T_{n+1} \land \neg P$. Let $c = (c_0, \ldots, c_n)$ where $c_{n+1} = b_{n+1}$, $i = 0, \ldots, n$. Then $I(c_0) = 1$ and $P(c_n) = 0$ since $c_{n+1} = b_{n+1} = s_{n+1}$. The fact that $t_o$ is a valid trace entails that the state $s_{n+1}$ is reachable in $n$-th time frame as well. So we have a contradiction.

Only if part: Assume the contrary i.e. no bad state is reachable in $(n+1)$-th time frame for the first time but $I_1$ is not redundant in $\exists S_{n}[I_0 \land I_1 \land T_{n+1} \land \neg P]$. This means that $\exists S_{n}[I_0 \land I_1 \land T_{n+1} \land \neg P] \neq \exists S_{n}[I_0 \land T_{n+1} \land \neg P]$. Then there is an assignment $s_{n+1}$ to variables of $S_{n+1}$ such that $\exists S_{n}[I_0 \land T_{n+1} \land \neg P] = 1$ and $\exists S_{n}[I_0 \land I_1 \land T_{n+1} \land \neg P] = 0$ under $s_{n+1}$. The means that there is an assignment $(a_0, \ldots, a_{n+1})$ satisfying $I_0 \land T_{n+1} \land \neg P$ where $a_{n+1} = s_{n+1}$. Hence, $s_{n+1}$ is a bad state that is reachable in $(n+1)$-th time frame.