Property Checking Without Invariant Generation

Eugene Goldberg
eu.goldberg@gmail.com

***Abstract*—We introduce $ProveProp$, a procedure for proving safety properties. $ProveProp$ 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 $ProveProp$ is twofold. First, it can prove a property without generating an inductive invariant. This is an implication of the fact that computing the reachability diameter of a system reduces to PQE. Second, PQE enables depth-first search, so $ProveProp$ can be used to find very deep bugs. To prove property true, $ProveProp$ has to consider traces of length up to the reachability diameter. This may slow down property checking for systems with a large diameter. We describe a variation of $ProveProp$ that can prove a property without generation of long traces.**

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.) A straightforward method of property checking is to use Quantifier Elimination (QE) for computing reachable states. Current QE algorithms still have problems ranging from memory explosion (QE by BDDs [4]) to poor performance (QE by SAT) [17], [3], [9], [10], [14]). So, the main focus of research has shifted towards methods that avoid QE [18], [2].

Nevertheless, we have at least two reasons to be optimistic about developing efficient procedures for quantified formulas. The first reason is that in many cases QE can be replaced with Partial QE (PQE) introduced in [12]. In contrast to QE, in PQE, only a (small) part of the formula is taken out of the scope of quantifiers. So, PQE can be dramatically more efficient than QE. The list of problems where one can effectively employ PQE includes combinational equivalence checking [7], simulation [6], invariant generation in property checking [8], computing states reachable *only* from a particular set of states [11]. In this paper, we expand this list with a procedure for property checking without invariant generation. The second reason for our optimism is the introduction of the machinery of D-sequents [9], [10], [13]. This machinery has shown a great promise for boosting algorithms on quantified formulas. In particular, it allows one to exploit the power of transformations that preserve equisatisfiability rather than equivalence with the original formula.

B. Problem we consider

Let ξ be a transition system specified by transition relation $T(S, S')$ and formula $I(S)$ describing initial states. Here S and S' is are sets of variables specifying the present and next states respectively. We will assume that T and I are propositional formulas in Conjunctive Normal Form (CNF). Let $\boldsymbol{s}$ be a state i.e. an assignment to S. Henceforth, by an assignment $\boldsymbol{q}$ to a set of variables Q we mean a *complete* assignment unless otherwise stated i.e. all variables of Q are assigned in $\boldsymbol{q}$.

Let $P(S)$ be a property of ξ. We will call a state $\boldsymbol{s}$ *bad* or *good* if $P(\boldsymbol{s}) = 0$ or $P(\boldsymbol{s}) = 1$ respectively. The problem we consider is to check if a bad state is reachable in ξ. 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 $K(S) \wedge T(S, S') \rightarrow K(S')$.

C. Property checking without invariant generation

In this paper, we consider an approach where the safety problem is solved without invariant generation. 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. Let $Rch(I, T, n)$ denote a formula specifying the set of states reachable in an (I, T)-system in n transitions. One can partition the safety problem into two subproblems.

1) Find value n such that $n \geq Diam(I, T)$.
2) Check that $Rch(I, T, n) \rightarrow P$.

We describe a procedure called $ProveProp$ that solves the two subproblems above. We will refer to the first problem (i.e. checking if the diameter is reached) as the **RD problem** where RD stands for Reachability Diameter. The RD problem can be easily formulated in terms of quantified formulas. Usually the RD problem is solved by performing quantifier elimination. In this paper, we show that the RD problem can be cast as an instance of the PQE problem. So, to solve the RD problem one actually needs to take only a small part of the formula out of the scope of the quantifiers.

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 good states i.e. by replacing I with I^{exp} such that $I \rightarrow I^{exp}$ and $I^{exp} \rightarrow P$.

D. The merits of $ProveProp$

The merits of $ProveProp$ are as follows. $ProveProp$ is based on PQE and as we mentioned above, the latter can

be dramatically *more efficient than QE*. Importantly, employing PQE facilitates depth-first search. This feature enables $ProveProp$ to search for *deep bugs*. Besides, due to depth-first search, $ProveProp$ does not need to compute the set of all states reachable in n transitions to decide if the diameter is reached. Probably, the most important advantage of $ProveProp$ is that is solves the RD problem in terms of quantified formulas. So $ProveProp$ and/or its variation $ProveProp*$ can potentially prove a property when an inductive invariant is prohibitively large or too difficult to find.

E. Contributions and structure of the paper

The contribution of the paper is twofold. First, we introduce a new method of property checking that does not require generation of an inductive invariant. Second, we give one more evidence that development of efficient PQE-algorithms is of great importance.

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 bad state 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. 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 $\exists W[A(V,W) \wedge B(V,W)]$, the problem of **Partial Quantifier Elimination** (**PQE**) is to find a quantifier-free formula $A^*(V)$ such that $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. 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 $\exists W[A \wedge B]$. Importantly, there is a strong relation between PQE and the notion of *redundancy* of a clause in a quantified formula. (A **clause** is a disjunction of literals.) In particular, solving the PQE problem above comes down to finding a formula $A^*(V)$ implied by $A \wedge B$ that makes the clauses of A redundant in $A^* \wedge \exists W[A \wedge B]$. Then $\exists W[A \wedge B] \equiv A^* \wedge \exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$.

Let $G(V)$ be a formula implied by B. Then $\exists W[A \wedge B] \equiv A^* \wedge G \wedge \exists W[B]$ implies that $\exists W[A \wedge B] \equiv A^* \wedge \exists W[B]$. 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. A summary of our results on this topic published in [9], [10], [12] can be found in [7]. In [5], we describe a "noise-free" PQE algorithm.

III. DEFINITIONS AND NOTATION

A. Basic definitions

Definition 1: Let ξ be an (I,T)-system. An assignment $\boldsymbol{s}$ to state variables S is called **a state**. A sequence of states $(\boldsymbol{s_0},\ldots,\boldsymbol{s_n})$ is called a **trace**. This trace is called **valid** if

- $I(\boldsymbol{s_0}) = 1$,
- $T(\boldsymbol{s_i}, \boldsymbol{s_{i+1}}) = 1$ where $i = 0,\ldots,n-1$.

Henceforth, we will drop the word *valid* if it is obvious from the context whether or not a trace is valid.

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

Definition 3: Let ξ be an (I,T)-system and P be a property to be checked. Let $(\boldsymbol{s_0},\ldots,\boldsymbol{s_n})$ be a valid trace such that

- every state $\boldsymbol{s_i}$, $i = 0,\ldots,n-1$ is **good** i.e. $P(\boldsymbol{s_i}) = 1$.
- state $\boldsymbol{s_n}$ is **bad** i.e. $P(\boldsymbol{s_n}) = 0$.

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

Remark 1: We will use the notions of a CNF formula $C_1 \wedge .. \wedge C_p$ and the set of clauses $\{C_1,\ldots,C_p\}$ *interchangeably*. In particular, we will consider saying that a CNF formula F has no clauses (i.e. $F = \emptyset$) as equivalent to $F \equiv 1$ and vice versa.

B. Some notation conventions

- S_j denotes the state variables of j-th time frame.
- $\mathbb{S}_j$ denotes $S_0 \cup \cdots \cup S_j$.
- $T_{j,j+1}$ denotes $T(S_j, S_{j+1})$.
- $\mathbb{T}_j$ denotes $T_{0,1} \wedge \ldots T_{j-1,j}$.
- I_0 and I_1 denote $I(S_0)$ and $I(S_1)$ respectively.
- Let $A(X')$ and $B(X'')$ be formulas where X', X'' are sets of variables such that $|X'| = |X''|$. Then $A(X') \hat{=} B(X'')$ means that A and B are equal modulo renaming variables. For instance, $I \hat{=} I_0 \hat{=} I_1$ holds.

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

ξ has only sequential variables. However, one can easily extend explanation to the case where ξ has combinational variables.)

On the one hand, introduction of stuttering does not affect the reachability of a bad state. On the other hand, stuttering guarantees that ξ has two nice properties. First, $\exists S[T(S,S')] \equiv 1$ holds since for every next state $\boldsymbol{s'}$, there is a "stuttering transition" from $\boldsymbol{s}$ to $\boldsymbol{s'}$ where $\boldsymbol{s} = \boldsymbol{s'}$. Second, if a state is unreachable in ξ in n transitions it is also unreachable in m transitions if $m < n$. Conversely, if a state is reachable in ξ 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 $n \geq Diam(I,T)$. In this subsection, we show that one can solve these two problems by PQE.

Given number n, the RD problem is to find the value of predicate $Diam(I,T) \leq n$. It reduces to checking if 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 $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$ i.e. whether $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}] \equiv \exists \mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$. If I_1 is redundant, then $Diam(I,T) \leq n$. This is a special case of the PQE problem. Instead, of finding a formula $H(S_{n+1})$ such that $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}] \equiv H \wedge \exists \mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$ one just needs to decide if $H \equiv 1$.

Here is an informal explanation of why redundancy of I_1 means $Diam(I,T) \leq n$. The set of states reachable in $n+1$ transitions is specified by $\exists \mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$. Adding I_1 to $I_0 \wedge \mathbb{T}_{n+1}$ shortcuts the initial time frame and so $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$ specifies the set of states reachable in n transitions. Redundancy of I_1 means that the sets of states reachable in $n+1$ and n transitions are the same. Proposition 1 states that the intuition above is correct.

Proposition 1: Let ξ be an (I,T)-system. Then $Diam(I,T) \leq n$ iff formula I_1 is redundant in $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$ where $I_0 \hat{=} I_1 \hat{=} I$ (i.e. iff $\exists \mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$).

Proposition 2 below shows that one can look for bugs by checking if I_1 is redundant in $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$ i.e. similarly to solving the RD problem.

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

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 $Diam(I,T) > n$ 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 $Diam(I,T) > n$ comes down to showing that I_1 is not redundant in $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$. This can be done by

a) generating a clause $C(S_{n+1})$ implied by $I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}$

b) finding a trace that satisfies $I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{C}$

Let $(\boldsymbol{s_0},\ldots,\boldsymbol{s_{n+1}})$ be a trace satisfying $I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{C}$. On the one hand, conditions a) and b) guarantee that $\exists \mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \neq \exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$ under $\boldsymbol{s_{n+1}}$. So I_1 is not redundant in $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$. On the other hand, condition a) shows that $\boldsymbol{s_{n+1}}$ is not reachable in n transitions (see Proposition 3 of the appendix) and condition b) proves $\boldsymbol{s_{n+1}}$ 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 $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$ i.e. by solving the PQE problem. In the context of PQE-solving, condition b) requires C not be a "noise" clause implied by $I_0 \wedge \mathbb{T}_{n+1}$ i.e. the part of the formula that remains quantified (see Section II). In this paper, we assume that one employs PQE algorithms that may generate noise clauses. (Note, however, that building a noise-free PQE-solver is possible [5].) So, to make sure that condition b) holds, one must prove $I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{C}$ satisfiable.

VI. *ProveProp* procedure

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

1) Show that $Diam(I,T) \leq n$ for some value of n (i.e. solve the RD problem).
2) Check that $Rch(I,T,n) \rightarrow P$.

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 the diameter

Proposition 1 entails that proving $Diam(I,T) \leq n$ comes down to showing that formula I_1 is redundant in $\exists \mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$. 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 $I_0 \wedge I_1 \wedge \mathbb{T}_i$. One can view formulas $H_1,\ldots,H_n$ 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 $\exists \mathbb{S}_{n-1}[I_0 \wedge I_1 \wedge \mathbb{T}_n] \equiv H_n \wedge \exists \mathbb{S}_{n-1}[I_0 \wedge \mathbb{H}_{n-1} \wedge \mathbb{T}_n]$ where $\mathbb{H}_{n-1} = H_1 \wedge \cdots \wedge H_{n-1}$.

$ProveProp$ 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 $\exists \mathbb{S}_1[I_0 \wedge H_1^* \wedge C \wedge \mathbb{T}_2] \equiv H_2 \wedge \exists \mathbb{S}_1[I_0 \wedge H_1^* \wedge \mathbb{T}_2]$ where $H_1^* = H_1 \setminus \{C\}$. Formula H_1 is replaced with H_1^*. 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 $\exists\mathbb{S}_1[I_0 \wedge H_1 \wedge \mathbb{T}_2]$ and $Diam(I,T) \leq 2$. If H_2 is not empty, then *ProveProp* picks a clause C of H_2 and builds formula $H_3(S_3)$ such that $\exists\mathbb{S}_2[I_0 \wedge H_1 \wedge H_2^* \wedge C \wedge \mathbb{T}_3] \equiv H_3 \wedge \exists\mathbb{S}_1[I_0 \wedge H_1 \wedge H_2^* \wedge \mathbb{T}_2]$ where $H_2^* = H_2 \setminus \{C\}$. Formula H_2 is replaced with H_2^*. 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 $n = Diam(I,T)$, any clause $C(S_{n+1})$ that is a descendant of I_1 is implied by $I_0 \wedge \mathbb{T}_{n+1}$. 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 bad state is reached. This is done as follows. Recall that H_{n+1} satisfies $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}] \equiv H_{n+1} \wedge \exists\mathbb{S}_n[I_0 \wedge \mathbb{H}_n \wedge \mathbb{T}_{n+1}]$. 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 $I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}$ but not $I_0 \wedge \mathbb{T}_{n+1}$. This means that the very fact that C is derived guarantees that there is *some* state (good or bad) that is reached in $(n+1)$-th iteration for the first time. So if a bad state $\boldsymbol{s}$ falsifies C, there is a chance that $\boldsymbol{s}$ 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 $\boldsymbol{s}$ being reachable.

A bad state falsifying C is generated by *ProveProp* as an assignment satisfying $\overline{C} \wedge \overline{P} \wedge R_{n+1}$. 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 $I_0 \wedge \mathbb{T}_{n+1}$ is derived, it is added to R_{n+1}. To find out if $\boldsymbol{s}$ is indeed reachable, one needs to check the satisfiability of formula $I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{A_{\boldsymbol{s}}}$ where $A_{\boldsymbol{s}}$ is the longest clause falsified by $\boldsymbol{s}$. An assignment satisfying this formula is a counterexample. If this formula is unsatisfiable, the SAT-solver returns a clause C^* implied by $I_0 \wedge \mathbb{T}_{n+1}$ and falsified by $\boldsymbol{s}$. This clause is added to R_{n+1} and *ProveProp* looks for a new state satisfying $\overline{C} \wedge \overline{P} \wedge R_{n+1}$. If another bad state $\boldsymbol{s}$ 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 *nil* if P does. *ProveProp* consists of three parts separated by the dotted line. The first part (lines 1-5) 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). *ProveProp* concludes the first part by setting formula H_1 to I_1 and parameter n to 1. Parameter n stores the index of the latest time frame where the corresponding formula H_n is not empty.

```
// R_n = R_1 ∧ ··· ∧ R_n
//
ProveProp(I, T, P){
1  T := MakeStutter(T);
2  Cex := Unsat(I_0 ∧ T_{0,1} ∧ P̄);
3  if (Cex ≠ nil) return(Cex);
4  H_1 := I_1;
5  n := 1;
   – – – – – – – – – –
6  while (H_1 ≢ 1) {
7    if (H_n ≡ 1) {
8      n := n − 1;
9      continue; }
10   if (FirstVisit(n + 1)) R_{n+1} := 1;
11   C := PickClause(H_n);
12   H_n := H_n \ {C};
13   H_{n+1} := PQE(∃S_n[I_0 ∧ C ∧ H_n ∧ T_{n+1}]);
14   RemNoise(H_{n+1}, R_{n+1}, I, T);
15   if (H_{n+1} ≡ 1) continue;
16   Cex := ChkBadSt(H_{n+1}, R_{n+1}, I, T, P);
17   if (Cex ≠ nil) return(Cex);
18   n := n + 1; }
   – – – – – – – – – –
19 return(nil); }
```

Fig. 1. The *ProveProp* procedure

The second part consists of a *while* loop (lines 6-18). 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 7-10) 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 $H_n \not\equiv 1$, *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 Subsection VI-B, R_{n+1} is used to accumulate clauses implied by $I_0 \wedge \mathbb{T}_{n+1}$.

ProveProp starts the second piece of the *while* loop (lines 11-13) 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 $\exists\mathbb{S}_n[I_0 \wedge \mathbb{H}_n \wedge C \wedge \mathbb{T}_{n+1}] \equiv H_{n+1} \wedge \exists\mathbb{S}_n[I_0 \wedge \mathbb{H}_n \wedge \mathbb{T}_{n+1}]$.

In the third piece, (lines 14-18), *ProveProp* analyzes formula H_{n+1}. First, it calls procedure ***RemNoise*** described in Subsection VI-D. It drops noise clauses of H_{n+1} i.e. ones implied by $I_0 \wedge \mathbb{T}_{n+1}$. If the resulting formula H_{n+1} is empty,

$RemNoise(H_i, \mathbb{R}_i, I, T)\{$
1 for each clause $C \in H_i$ {
2 if ($Unsat(I_0 \wedge \mathbb{T}_i \wedge \mathbb{R}_i \wedge \overline{C})$) {
3 $H_i := H_i \setminus \{C\}$;
4 $R_i := R_i \wedge C$; }}

Fig. 2. The $RemNoise$ procedure

$ChkBadSt(H_i, \mathbb{R}_i, I, T, P)\{$
1 for each clause $C \in H_i$ {
2 while ($true$) {
3 $\boldsymbol{s} := SatAssgn(\overline{C} \wedge \overline{P} \wedge R_i)$;
4 if ($\boldsymbol{s} = nil$) break;
5 $(Cex, C^*) := Unsat(I_0 \wedge \mathbb{T}_i \wedge \mathbb{R}_i \wedge \overline{A_{\boldsymbol{s}}})$;
6 if ($Cex \neq nil$) return(Cex);
7 $R_i := R_i \wedge C^*$;}}
8 return(nil);}

Fig. 3. The $ChkBadSt$ procedure

$ProveProp$ starts a new iteration. Otherwise, $ProveProp$ calls procedure *ChkBadSt* also described in Subsection VI-D. *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 and starts a new iteration.

The third part of $ProveProp$ consists of line 19. $ProveProp$ gets to this line if I_1 is proved redundant and no bad state is reachable in $Diam(I, T)$ transitions. This means that property P holds and so $ProveProp$ returns nil.

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 $I_0 \wedge \mathbb{T}_i$. So for every clause C of H_i, *RemNoise* checks if formula $I_0 \wedge \mathbb{T}_i \wedge \mathbb{R}_i \wedge \overline{C}$ is satisfiable. (Here $\mathbb{R}_i$= $R_1 \wedge \cdots \wedge R_i$. It specifies clauses implied by $I_0 \wedge \mathbb{T}_i$ 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 state $\boldsymbol{s}$ satisfying formula $\overline{C} \wedge \overline{P} \wedge R_i$ is reachable in i transitions. The inner loop iterates until this formula becomes unsatisfiable.

Finding out if $\boldsymbol{s}$ is reachable in i transitions comes down to checking the satisfiability of formula $I_0 \wedge \mathbb{T}_i \wedge \mathbb{R}_i \wedge \overline{A_{\boldsymbol{s}}}$. (Here $A_{\boldsymbol{s}}$ is the longest clause falsified by $\boldsymbol{s}$.) An assignment satisfying this formula specifies a counterexample. If this formula is unsatisfiable, a clause $C^*(S_i)$ is returned that is implied by $I_0 \wedge \mathbb{T}_i$ and falsified by $\boldsymbol{s}$. 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 good 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 invariant (as opposed to building the strongest invariant consisting of all reachable states).

$ProveProp*(I, T, P)\{$
1 $T := MakeStutter(T)$;
2 $Cex := Unsat(I_0 \wedge T_{0,1} \wedge \overline{P})$;
3 if ($Cex \neq nil$) return(Cex);
4 $I^{exp} := ExpandInitStates(I, P)$;
– – – – – – – – –
5 while ($true$) {
6 $Cex := ProveProp(I^{exp}, T, P)$;
7 if ($Cex \neq nil$) {
8 $\boldsymbol{s_0} := ExtractInitState(Cex)$;
9 if ($I(\boldsymbol{s_0}) = 1$) return(Cex);
10 $ExcludeState(I^{exp}, \boldsymbol{s_0})$;
11 continue; }
12 return(nil);}}

Fig. 4. The $ProveProp*$ procedure

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). If $ProveProp$ returns nil, property P holds and $ProveProp*$ returns nil (line 12). Otherwise, $ProveProp*$ analyzes the counterexample $Cex = (\boldsymbol{s_0}, \ldots, \boldsymbol{s_n})$ returned by $ProveProp$ (lines 7-11). If state $\boldsymbol{s_0}$ of Cex, satisfies I, then P does not hold and $ProveProp*$ returns Cex as a counterexample (line 9). If $I(\boldsymbol{s_0}) = 0$, $ProveProp*$ excludes $\boldsymbol{s_0}$ by conjoining I^{exp} with a clause C such that $C(\boldsymbol{s_0}) = 0$ 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 $\boldsymbol{s_1}$ of Cex can be reached from a state $\boldsymbol{s_0^*}$ such that $I^{exp}(\boldsymbol{s_0^*})$=1, $I(\boldsymbol{s_0^*}) = 0$ and $\boldsymbol{s_0^*} \neq \boldsymbol{s_0}$. Then one may try to pick clause C so that it is falsified by both $\boldsymbol{s_0}$ and $\boldsymbol{s_0^*}$.

$ProveProp*$ is a complete procedure i.e. it eventually proves P or finds a counterexample. In the worst case, $ProveProp*$ will have to reduce I^{exp} to I.

VIII. Some Background

The original methods of property checking were based on BDDs and computed strong invariants by quantified elimination [16]. Since BDDs frequently get prohibitively large, SAT-based methods of property checking have been introduced. Some of them, like interpolation [18] and IC3 [2] have achieved a great boost in performance. Among *incomplete* SAT-based methods (i.e. those that can do only bug hunting), Bounded Model Checking (BMC) [1] has enjoyed a lot of success.

In spite of great progress in property checking, the existing methods have at least two flaws. The first flaw is that they cannot find "deep" bugs. In case of complete algorithms of property checking, this flaw is a consequence of breadth-first search employed by these algorithms. For instance, IC3 builds a sequence of formulas $F_1, \ldots, F_n$ where F_i specifies a superset of the set of states reachable in i transitions. Formula F_i is built *after* formulas $F_1, \ldots, F_{i-1}$ have been constructed i.e. in a breadth-first manner. In case of BMC, the problem is as follows. To build a counterexample of n transitions, one needs to find an assignment satisfying a formula whose size grows linearly with n. So finding a counterexample for large values of n is infeasible.

The second flaw is that the ability of existing methods to build an inductive invariant is based on assumptions that do not always hold. For instance, IC3 builds an inductive invariant by tightening up the property P to be proved. So an implicit assumption of IC3 is that there is an inductive invariant that is close to P. Interpolation based methods extract an interpolant from a proof that a property holds for a specified number of transitions. The quality of an interpolant strongly depends on that of the proof it is extracted from. So these methods work under the assumption that a SAT-solver can generate a high-quality and structure-aware proof. In many cases, the assumptions above do not hold. For instance, neither assumption above holds for an instance of property checking specifying sequential equivalence checking. An inductive invariant may be very far away from the property describing sequential equivalence and a conflict driven SAT-solver builds proofs of poor quality for equivalence checking formulas.

After the introduction of PQE [12], we formulated a few approaches addressing the two flaws above. In [11], we described a PQE-based procedure for property checking meant for finding deep bugs. However, the procedure we described there is incomplete. In [8], we showed how one can combine transition relation relaxation and PQE to build invariants that may be far away from the property at hand. In this paper, we continue that line of research. Similarly to the approach of [11], the $ProveProp$ procedure can find deep bugs. However, in contrast to the former, $ProveProp$ is complete. Besides, in case property P holds, convergence of $ProveProp$ depends on proximity of P to an inductive invariant even less than that of algorithm of [8]. This is because $ProveProp$ does not build an explicit inductive invariant.

The idea of proving a property without generating an invariant is not new. For instance, earlier it was proposed to combine BMC with finding a *recurrence* diameter [15]. 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 a new procedure for checking safety properties called $ProveProp$. It is based on a technique called Partial Quantifier Elimination (PQE) and so the efficiency of $ProveProp$ is predicated on that of PQE. This assumption is not far from the truth due to two facts. First, PQE can be dramatically faster than complete quantifier elimination because only a small part of the formula is taken out of the scope of quantifiers. Second, the introduction of the machinery of D-sequents has a promise of boosting the performance of PQE even further. The advantage of $ProveProp$ is twofold. First, due to using PQE, it enjoys depth-first search and so can be used for finding very deep bugs. Second, $ProveProp$ can prove that a property holds without generation of an inductive invariant. This may turn out to be extremely useful when inductive invariants are prohibitively large or are hard to find.

References

[1] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[2] A. R. Bradley. Sat-based model checking without unrolling. In *VMCAI*, pages 70–87, 2011.

[3] J. Brauer, A. King, and J. Kriener. Existential quantificationnn as incremental sat. In *Proc. of CAV-11*, pages 191–207. Springer-Verlag, July 2011.

[4] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] E. Goldberg. On noise-free algorithm of partial quantifier elimination. To be submitted.

[6] E. Goldberg. Equivalence checking and simulation by computing range reduction. Technical Report arXiv:1507.02297 [cs.LO], 2015.

[7] E. Goldberg. Equivalence checking by logic relaxation. Technical Report arXiv:1511.01368 [cs.LO], 2015.

[8] E. Goldberg. Property checking by logic relaxation. Technical Report arXiv:1601.02742 [cs.LO], 2016.

[9] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. In *FMCAD-12*, pages 34–44, 2012.

[10] E. Goldberg and P. Manolios. Quantifier elimination via clause redundancy. In *FMCAD-13*, pages 85–92, 2013.

[11] E. Goldberg and P. Manolios. Bug hunting by computing range reduction. Technical Report arXiv:1408.7039 [cs.LO], 2014.

[12] E. Goldberg and P. Manolios. Partial quantifier elimination. In *Proc. of HVC-14*, pages 148–164. Springer-Verlag, 2014.

[13] E. Goldberg and P. Manolios. Quantifier elimination by dependency sequents. *Formal Methods in System Design*, 45(2):111–143, 2014.

[14] W. Klieber, M. Janota, J.Marques-Silva, and E. M. Clarke. Solving qbf with free variables. In *CP*, pages 415–431, 2013.

[15] D. Kroening and O. Strichman. Efficient computation of recurrence diameters. In *VMCAI-2003, Lecture Notes in Computer Science, vol. 2575*, pages 298–309, 2003.

[16] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[17] K. McMillan. Applying sat methods in unbounded symbolic model checking. In *Proc. of CAV-02*, pages 250–264. Springer-Verlag, 2002.

[18] K. L. Mcmillan. Interpolation and sat-based model checking. In *CAV-03*, pages 1–13. Springer, 2003.

APPENDIX

Lemma 1 below is used in proving Proposition 1.

Lemma 1: Let ξ be an (I,T)-system. Then $Diam(I,T) \leq n$ iff $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$ where $I_0 \hat{=} I_1 \hat{=} I$ and $n \geq 0$.

Proof: **If part:** Given $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$, let us prove $Diam(I,T) \leq n$. Assume the contrary, i.e. $Diam(I,T) > n$. Then there is a state $\boldsymbol{a_{n+1}}$ that is reachable only in $(n+1)$-th time frame. Hence, there is a trace $t_a = (\boldsymbol{a_0},\ldots,\boldsymbol{a_{n+1}})$ satisfying $I_0 \wedge \mathbb{T}_{n+1}$. Equality $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$ entails the existence of a trace $t_b = (\boldsymbol{b_0},\ldots,\boldsymbol{b_{n+1}})$ satisfying $I_1 \wedge \mathbb{T}_{n+1}$ where $\boldsymbol{b_{n+1}} = \boldsymbol{a_{n+1}}$.

Let t_c be a trace $(\boldsymbol{c_0},\ldots,\boldsymbol{c_n})$ where $\boldsymbol{c_i} = \boldsymbol{b_{i+1}}$, $i = 0,\ldots,n$. The fact that t_b satisfies $I_1 \wedge \mathbb{T}_{n+1}$ implies that t_c satisfies $I_0 \wedge \mathbb{T}_n$. Since $\boldsymbol{c_n} = \boldsymbol{b_{n+1}} = \boldsymbol{a_{n+1}}$, state $\boldsymbol{a_{n+1}}$ is reachable in n transitions. So we have a contradiction.

Only if part: Given $Diam(I,T) \leq n$, let us prove that $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$.

First, let us show that if $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$=1 under assignment $\boldsymbol{s_{n+1}}$ to S_{n+1}, then $\exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$=1 under assignment $\boldsymbol{s_{n+1}}$. The fact that $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$=1 under $\boldsymbol{s_{n+1}}$ means that state $\boldsymbol{s_{n+1}}$ is reachable in $n+1$ transitions. Since $Diam(I,T) \leq n$, there has to be a trace $t_a = (\boldsymbol{a_0},\ldots,\boldsymbol{a_k})$ where $k \leq n$ and $\boldsymbol{a_k} = \boldsymbol{s_{n+1}}$. Let m be equal to $n+1-k$. Let $t_b = (\boldsymbol{b_0},\ldots,\boldsymbol{b_{n+1}})$ be a trace defined as follows: $\boldsymbol{b_i}$=$\boldsymbol{a_0}$, $i = 0,\ldots,m$, and $\boldsymbol{b_i}$=$\boldsymbol{a_{i-m}}$, $i = m+1,\ldots,n+1$. It is not hard to see that t_b satisfies $I_1 \wedge \mathbb{T}_{n+1}$ and $\boldsymbol{b_{n+1}} = \boldsymbol{a_k}$=$\boldsymbol{s_{n+1}}$. So formula $\exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$ evaluates to 1 under assignment $\boldsymbol{s_{n+1}}$ to S_{n+1}.

Now, let us show that if $\exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$=1 under assignment $\boldsymbol{s_{n+1}}$, then $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$=1 under assignment $\boldsymbol{s_{n+1}}$. The fact that $\exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$=1 under assignment $\boldsymbol{s_{n+1}}$ means that $\boldsymbol{s_{n+1}}$ is reachable in at most n transitions. Due to the stuttering feature of ξ, state $\boldsymbol{s_{n+1}}$ is also reachable in $n+1$ transitions. This means that there is a trace $t = (\boldsymbol{s_0},\ldots,\boldsymbol{s_{n+1}})$ satisfying $I_0 \wedge \mathbb{T}_{n+1}$. So $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}]$=1 under assignment $\boldsymbol{s_{n+1}}$. ■

Proposition 1: Let ξ be an (I,T)-system. Then $Diam(I,T) \leq n$ iff formula I_1 is redundant in $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$ where $I_0 \hat{=} I_1 \hat{=} I$ (i.e. iff $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$).

Proof: Lemma 1 entails that to prove the proposition at hand it is sufficient to show that $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$ iff formula I_1 is redundant in $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$.

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

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

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

Only if part: Given $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \equiv \exists\mathbb{S}_n[I_1 \wedge \mathbb{T}_{n+1}]$, let us show that I_1 is redundant in $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$. Assume that I_1 is not redundant i.e. $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] \not\equiv \exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}]$. Then there is an assignment $\boldsymbol{s_{n+1}}$ to variables of S_{n+1} such that $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1}] = 1$ and $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}] = 0$. This means that

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

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

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

Proof: **If part:** Assume the contrary i.e. I_1 is redundant in $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$ but there is a bad state $\boldsymbol{s_{n+1}}$ that is reachable in $(n+1)$-th time frame for the first time. Then there is an assignment $t_a = (\boldsymbol{a_0},\ldots,\boldsymbol{a_{n+1}})$ satisfying $I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{P}$ where $\boldsymbol{a_{n+1}}$ = $\boldsymbol{s_{n+1}}$. Redundancy of I_1 means that $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}] \equiv \exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$. Then there is an assignment $t_b = (\boldsymbol{b_0},\ldots,\boldsymbol{b_{n+1}})$ where $\boldsymbol{b_{n+1}}$ = $\boldsymbol{s_{n+1}}$ that satisfies $I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}$. Let $t_c = (\boldsymbol{c_0},\ldots,\boldsymbol{c_n})$ where $\boldsymbol{c_i} = \boldsymbol{b_{i+1}}$, $i = 0,\ldots,n$. Then $I(\boldsymbol{c_0}) = 1$ and $P(\boldsymbol{c_n}) = 0$ since $\boldsymbol{c_n}$=$\boldsymbol{b_{n+1}}$=$\boldsymbol{s_{n+1}}$. Taking into account that t_c is a valid trace we conclude that state $\boldsymbol{s_{n+1}}$ 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 $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$. This means that $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}] \not\equiv \exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$. Then there is an assignment $\boldsymbol{s_{n+1}}$ to variables of S_{n+1} such that $\exists\mathbb{S}_n[I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$=1 and $\exists\mathbb{S}_n[I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}]$ = 0 under $\boldsymbol{s_{n+1}}$. The means that there is an assignment $(\boldsymbol{a_0},\ldots,\boldsymbol{a_{n+1}})$ satisfying $I_0 \wedge \mathbb{T}_{n+1} \wedge \overline{P}$ where $\boldsymbol{a_{n+1}}$ = $\boldsymbol{s_{n+1}}$. Hence, $\boldsymbol{s_{n+1}}$ is a bad state that is reachable in $(n+1)$-th time frame.

Let us show that $\boldsymbol{s_{n+1}}$ is not reachable in a previous time frame. Assume the contrary i.e. $\boldsymbol{s_{n+1}}$ is reachable in k-th time

frame where $k < n+1$. Then there is an assignment $t_b = (\boldsymbol{b_0},\ldots,\boldsymbol{b_k})$ satisfying $I_0 \wedge \mathbb{T}_k \wedge \overline{P}$ where $\boldsymbol{b_k}$=$\boldsymbol{s_{n+1}}$. Let $t_c = (\boldsymbol{c_0},\ldots,\boldsymbol{c_{n+1}})$ be defined as follows: $\boldsymbol{c_0}$=$\boldsymbol{c_1}$=$\boldsymbol{b_0}$, $\boldsymbol{c_i}$ = $\boldsymbol{b_{i-1}}$, $i = 2,\ldots,k+1$, $\boldsymbol{c_i}$ = $\boldsymbol{b_k}$, $i = k+2,\ldots,n+1$. Informally, t_c specifies the same sequence of states as t_b plus stuttering in the initial state and after reaching state $\boldsymbol{c_{k+1}}$ equal to $\boldsymbol{b_k}$ (and so to $\boldsymbol{s_{n+1}}$). Then t_c satisfies $I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \wedge \overline{P}$ under assignment $\boldsymbol{s_{n+1}}$ and so we have a contradiction. ■

Proposition 3: Let ξ be an (I,T)-system and $H(S_{n+1})$ be a formula. Then $I_0 \wedge I_1 \wedge \mathbb{T}_{n+1} \rightarrow H$ entails $I_1 \wedge T_{1,2} \wedge \ldots T_{n,n+1} \rightarrow H$.

Proof: Assume the contrary i.e. $I_1 \wedge T_{1,2} \wedge \ldots T_{n,n+1} \rightarrow H$ does not hold. Then there is trace t_a=$(\boldsymbol{a_1},\ldots,\boldsymbol{a_{n+1}})$ that satisfies $I_1 \wedge T_{1,2} \wedge \ldots T_{n,n+1}$ but falsifies H. The latter means that $\boldsymbol{a_{n+1}}$ falsifies H. Let trace $t_b = (\boldsymbol{b_0},\ldots,\boldsymbol{b_{n+1}})$ be obtained from t_a as follows: $\boldsymbol{b_0} = \boldsymbol{b_1}$, $\boldsymbol{b_i}$=$\boldsymbol{a_i}$, $i = 1,\ldots,n+1$. Since $I(\boldsymbol{a_1}) = 1$, then $I(\boldsymbol{b_0}) = I(\boldsymbol{b_1}) = 1$. Due to stuttering, $T(\boldsymbol{b_0},\boldsymbol{b_1}) = 1$. So trace t_b satisfies $I_0 \wedge I_1 \wedge \mathbb{T}_{n+1}$. Since $\boldsymbol{b_{n+1}} = \boldsymbol{a_{n+1}}$, then $H(\boldsymbol{b_{n+1}}) = 0$ and $I_0 \wedge I_1 \wedge \mathbb{T}_n \not\rightarrow H$. So we have a contradiction. ■