From Traces To Proofs: Proving Concurrent Programs Safe

Chinmay Narayan[*], Subodh Sharma[†], Shibashis Guha[‡] and S.Arun-Kumar[§]
Department of Computer Science and Engineering,
Indian Institute of Technology Delhi
Email: [*]chinmay@cse.iitd.ac.in, [†]svs@cse.iitd.ac.in, [‡]shibashis@cse.iitd.ac.in, [§]sak@cse.iitd.ac.in

***Abstract*—Nondeterminism in scheduling is the cardinal reason for difficulty in proving correctness of concurrent programs. A powerful proof strategy was recently proposed [6] to show the correctness of such programs. The approach captured data-flow dependencies among the instructions of an interleaved and error-free execution of threads. These data-flow dependencies were represented by an *inductive data-flow graph* (iDFG), which, in a nutshell, denotes a set of executions of the concurrent program that gave rise to the discovered data-flow dependencies. The iDFGs were further transformed in to alternative finite automatons (AFAs) in order to utilize efficient automata-theoretic tools to solve the problem. In this paper, we give a novel and efficient algorithm to directly construct AFAs that capture the data-flow dependencies in a concurrent program execution. We implemented the algorithm in a tool called `ProofTraPar` to prove the correctness of finite state cyclic programs under the sequentially consistent memory model. Our results are encouraging and compare favorably to existing state-of-the-art tools.**

I. Introduction

The problem of checking whether or not a correctness property (specification) is violated by the program (implementation) is already known to be challenging in a sequential set-up, let alone when programs are implemented exploiting concurrency. The central reason for greater complexity in verification of concurrent implementations is due to the exponential increase in the number of executions. A concurrent program with n threads and k instructions per thread can have $(nk)!/(k!)^n$ executions under a sequentially consistent (SC)[12] memory model. A common approach to address the complexity due to the exponential number of executions is *trace partitioning*.

In [6], a powerful proof strategy was presented which utilized the notion of trace partitioning. Let us take Peterson's algorithm in Figure 1 to convey the central idea behind the trace partitioning approach. In this algorithm, two processes, P_i and P_j, coordinate to achieve an exclusive access to a critical section (CS) using shared variables. A process P_i will busy-wait if P_j has expressed interest to enter its CS and `t` is j.

In order to prove the mutual exclusion (ME) property of Peterson's algorithm, we must consider the boolean conditions of the while loops at control locations 3 and 8. the ME property is established only when at most one of these conditions is false under every execution of the program, *i.e.*, ME must be

$\text{flag}_\text{i} = false, \text{flag}_\text{j} = false, \text{t} = 0;$

P_i	P_j
$While(true)${	$While(true)${
1. flag_i:=true;	6. flag_j:=true;
2. t:=j;	7. t:=i;
3. while(flag_j = true & t = j);	8. while(flag_i = true & t = i);
4. `//Critical Section`	9. `//Critical Section`
5. flag_i:=false;	10. flag_j:=false;
}	}

Fig. 1: Peterson's algorithm for two processes P_i and P_j

shown to hold true on unbounded number of traces (trace is "a sequence of events corresponding to an interleaved execution of processes in the program"[9]) generated due to unbounded number of unfoldings of the loops. Notice that events at control locations 3 and 8 are *data-dependent* on events from control locations $2, 6, 7$ and $1, 2, 7$, respectively. In any finite prefix of a trace of $P_i \| P_j$ (interleaved execution of P_i and P_j) up to the events corresponding to control location 3 or 8, the last instance of event at control location 2, `lst2`, and the last instance of event at control location 7, `lst7`, can be ordered in only one of the following two ways; either `lst2` appears before `lst7` or `lst2` appears after `lst7`. This has resulted in partitioning of an unbounded set of traces to a set with mere two traces.

When `lst2` appears before `lst7`, then the final value of the variable `t` is i, thus making the condition at control location 8 to be `true`. In the other case, when `lst2` appears after `lst7`, the final value of the variable `t` is j, thereby making the condition at control location 3 evaluate to `true`. Hence, in no trace both the conditions are false simultaneously. This informal reasoning indicates that both processes can never simultaneously enter in their critical sections. Thus, proof of correctness for Peterson's algorithm can be demonstrated by picking two traces, as mentioned above, from the set of infinite traces and proving them correct. In general, the intuition is that a proof for a single trace of a program can result in pruning of a large set of traces from consideration. To convert this intuition to a feasible verification method, there is a need to construct a formal structure from a proof of a trace σ such that the semantics of this structure includes a set of all those traces that have proof arguments equivalent to proof of σ. *Inductive Data Flow Graphs* (iDFG) was proposed in [6] to capture data-dependencies among the events of a trace and to perform trace partitioning. All traces that have the same iDFG

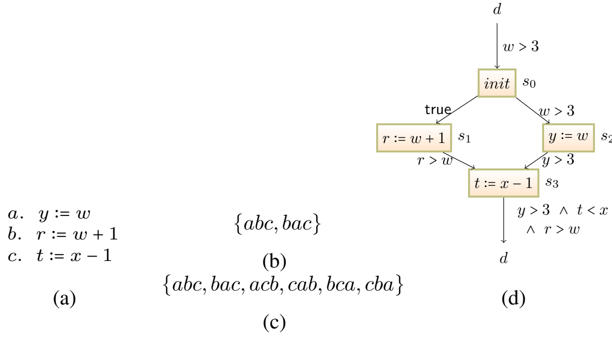


Fig. 2: Comparison with [6]

must have the same proof of correctness. In every iteration of their approach, a trace is picked from the set of *all* traces that is yet to be covered by the *iDFG*. An iDFG is constructed from its proof. The process is repeated until all the traces are either covered in the *iDFG* or a counter-example is found. An intervening step is involved where the iDFG is converted to an *alternating finite automaton* (AFA). While we explain AFA in later sections, it suffices to understand at this stage that the language accepted by this AFA and the set of traces captured by the corresponding iDFG is the same. Their reason for this conversion is to leverage the use of automata-theoretic operations such as subtraction, complement *etc.*, on the set of traces.

Though the goal of paper [6] is verification of concurrent programs which is the same as in this work, our work has some crucial differences: (i) An AFA is constructed directly from the proof of a trace without requiring the iDFG construction, (ii) the verification procedure built on directly constructed AFA is shown to be sound and complete (weakest-preconditions are used to obtain the proof of correctness of a trace), (iii) to the best of our knowledge, we provide the first implementation of the proof strategy discussed in [6]. The example trace of Figure 2(a) highlights the key difference between iDFG to AFA conversion of [6] and the direct approach presented in this work. Note that all three events a, b, and c are data independent, hence every resulting trace after permuting the events in abc also satisfies the same set of pre- and post-conditions. For a Hoare triple $\{w > 3\}\ abc\ \{y > 3\ \wedge\ t < x\ \wedge\ r > w\}$, Figure 2(b) shows the set of traces admitted by an AFA (obtained from iDFG shown in Figure 2(d)) after the first iteration, as computed by [6]. This set clearly does not represent every permutation of abc; consequently, more iterations are required to converge to an AFA that represents all traces admissible under the same set of pre- and post-conditions. In contrast, the AFA that is constructed directly by our approach from the Hoare triple $\{w > 3\}\ abc\ \{y > 3\ \wedge\ t < x\ \wedge\ r > w\}$, admits the set of traces shown in Figure 2(c). Hence, on this example, our strategy terminates in a single iteration.

To summarize, the contributions of this work are as follows:

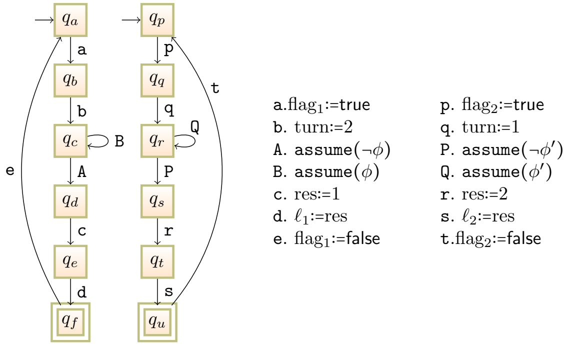
- we present a novel algorithm to directly construct an AFA from a proof of a sequential trace of a finite state (possibly cyclic) concurrent program. This construction is used to give a sound and complete verification procedure along the lines of [6].
- While [6] allowed the use of any sequential verification method to construct a proof of a given trace, the paper does not comment on the performance and the feasibility of their approach due to the lack of an implementation. The second contribution of this paper is an implementation in the form of a tool, `ProofTraPar`. We compare our implementation against other state-of-the-art tools in this domain, such as THREADER [10] and Lazy-CSeq [11] (winners in the concurrency category of the software verification competitions held in 2013, 2014, and 2015). `ProofTraPar`, on average, performed an order of magnitude better than THREADER and 3 times better than Lazy-CSeq.

The paper is organized as follows: Section II covers the notations, definitions and programming model used in this paper; Section III presents our approach with the help of an example to convey the overall idea and describes in detail the algorithms for constructing the proposed alternating finite automaton along with their correctness proofs. This section ends with the overall verification algorithm with the proof of its soundness and completeness for finite state concurrent programs. Section IV presents the experimental results and comparison with existing tools namely THREADER [10] and Lazy-CSeq [11]. Section V presents the related work and Section VI concludes with possible future directions.

II. Preliminaries

A. Program Model

We consider shared-memory concurrent programs composed of a fixed number of deterministic sequential processes and a finite set of shared variables $\mathtt{SV}$. A *concurrent program* is a quadruple $\mathcal{P} = (P, A, \mathcal{I}, \mathcal{D})$ where P is a finite set of processes, $A = \{A_p \mid p \in P\}$ is a set of automata, one for each process specifying their behaviour, $\mathcal{D}$ is a finite set of constants appearing in the syntax of processes and $\mathcal{I}$ is a function from variables to their initial values. Each process $p \in P$ has a disjoint set of local variables $\mathtt{LV}_p$. Let $\mathtt{Exp}_p$ ($\mathtt{BExp}_p$) denote the set of expressions (boolean expressions), ranged over by exp (ϕ) and constructed using shared variables, local variables, $\mathcal{D}$, and standard mathematical operators. Each specification automaton A_p is a quadruple $\langle Q_p, q_p^{init}, \delta_p, \mathrm{Assrn}_p\rangle$ where Q_p is a finite set of control states, q_p^{init} is the initial state, and $\mathrm{Assrn}_p \subseteq Q_p \times \mathtt{BExp}_p$ is a relation specifying the assertions that must hold at some control state. Each transition in δ_p is of the form (q, op_p, q') where $op_p \in \{\mathrm{x{:=}exp}, \mathtt{assume}(\phi), \mathtt{lock}(x)\}$. Here x:=exp evaluates exp in the current state and assigns the value to x where $\mathrm{x} \in \mathtt{SV} \cup \mathtt{LV}_p$. $\mathtt{assume}(\phi)$ is a blocking operation that suspends the execution if the boolean expression ϕ evaluates to false otherwise it acts as `nop`. This instruction is used to encode control path conditions of a program. $\mathtt{lock}(x)$, where $x \in \mathtt{SV}$, is a blocking operation that suspends the execution if the value of x is not equal to 0 otherwise it assigns 1 to x. Operation unlock is achieved by assigning 0 to this shared variable. Each of these operations are deterministic in nature,



$\text{Assrn}_{P_1}(q_f) \stackrel{def}{=} (\ell_1 = 1),\ \text{Assrn}_{P_2}(q_u) \stackrel{def}{=} (\ell_2 = 2)$
$\phi \stackrel{def}{=} \text{flag}_2 = \texttt{true}\ \&\&\ \text{turn} = 2, \phi' \stackrel{def}{=} \text{flag}_1 = \texttt{true}\ \&\&\ \text{turn} = 1$

Fig. 3: Specification of Peterson's algorithm

i.e. execution of any two same operations from the same states always give the same behaviour. In all examples of this paper, we use symbolic labels to succinctly represent program operations. For example, Figure 3 shows the specification of two processes in Peterson's algorithm. Labels $\{\mathtt{a}, \mathtt{b}, \mathtt{p}, \mathtt{q} \cdots\}$ denote operations in the program. Variable res is introduced to specify the mutual exclusion property as a safety property. A process P_i sets this variable to i inside its critical section. Assertions $\texttt{assert}(\text{res} = \text{i})$ is checked in P_i before leaving its critical section. If these assertions hold in every execution of these two processes then the mutual exclusion property holds. These assertions are shown as $\text{Assrn}_{P_1}(q_f)$ and $\text{Assrn}_{P_2}(q_u)$ in Figure 3 and they need to be checked at state q_f and q_u respectively. A tuple, say t, of n elements can be represented as a function such that $t[k]$ returns the kth element of this tuple. Given a function fun, $fun[a \leftarrow b]$ denotes another function same as of fun except at a where it returns b.

a) Parallel Composition in the SC memory model: Given a concurrent program $\mathcal{P} = (P, A, \mathcal{I}, \mathcal{D})$ consisting of n processes $P = \{p_1, \cdots, p_n\}$ we define an automaton $\mathcal{A}(\mathcal{P}) = (\overline{Q}, \overline{q}^{init}, \overline{\delta}, \overline{\text{Assrn}})$ to represent the parallel composition of $\mathcal{P}$ in the SC memory model. Here $\overline{Q} = Q_{p_1} \times \cdots \times Q_{p_n}$ is the set of states ranged over by $\overline{q}$, $\overline{q}^{init} = (q_{p_1}^{init}, \cdots, q_{p_n}^{init})$ is the initial state, and transition relation $\overline{\delta}$ models the interleaving semantics. Formally, $(\overline{q}, op_j, \overline{q}') \in \overline{\delta}$ iff there exists a $j \in \{1 \cdots n\}$ such that $\overline{q}[j] = q_{p_j}$, $\overline{q}' = \overline{q}[j \leftarrow q'_{p_j}]$ and $(q_{p_j}, op_j, q'_{p_j}) \in \delta_{p_j}$. For a state $\overline{q}$, let $T(\overline{q}) = \{\text{Assrn}_{p_i}(\overline{q}[i]) \mid i \in \{1 \cdots n\}\}$. If $T(\overline{q})$ is not empty then $\overline{\text{Assrn}}(\overline{q})$ is the conjunction of assertions in the set $T(\overline{q})$. Relation $\overline{\text{Assrn}}$ captures the assertions which need to be checked in the interleaved traces of P. As our interest lies in analyzing those traces which reach those control points where assertions are specified, we mark all those states where the relation $\overline{\text{Assrn}}$ is defined as accepting states. Every word accepted by $\mathcal{A}(\mathcal{P})$ represents one SC execution leading to a control location where at least one assertion is to be checked.

B. Weakest Precondition

Given an operation $op \in \mathcal{OP}(P)$ and a postcondition formula ϕ, the weakest precondition *of op with respect to* ϕ, denoted by $\texttt{wp}(op, \phi)$, is the weakest formula ψ such that, starting from any program state s that satisfies ψ, the execution of the operation op terminates and the resulting program state s' satisfies ϕ.

$$\texttt{wp}(op,\phi) = \begin{cases} \phi & \text{if } op \stackrel{def}{=} \text{skip} \\ \phi[\texttt{x/exp}] & \text{if } op \stackrel{def}{=} \text{x:=exp} \\ \phi \ \wedge\ \phi' & \text{if } op \stackrel{def}{=} \texttt{assert}(\phi') \\ \texttt{wp}(\texttt{assume}(\text{x} = 0), \texttt{wp}(\text{x:=1}, \phi)) & \text{if } op \stackrel{def}{=} \texttt{lock}(x) \\ \texttt{wp}(op_1, \texttt{wp}(op_2, \phi)) & \text{if } op \stackrel{def}{=} op_1.op_2 \\ \phi' \Rightarrow \phi & \text{if } op \stackrel{def}{=} \texttt{assume}(\phi') \end{cases}$$

Fig. 4: Weakest precondition axioms

Given a formula ϕ, variable X and expression e, let $\phi[\mathtt{X/e}]$ denote the formula obtained after substituting all free occurrences of X by e in ϕ. We assume an equality operator over formulae that represents syntactic equality. Every formula is assumed to be normalized in a conjunctive normal form (CNF). We use $\texttt{true}$ ($\texttt{false}$) to syntactically represent a logically valid (unsatisfiable) formula. Weakest precondition axioms for different program statements are shown in Figure 4. Here empty sequence of statements is denote by skip. We have the following properties about weakest preconditions.

Property 1: If $\texttt{wp}(op, \phi_1) = \psi_1$ and $\texttt{wp}(op, \phi_2) = \psi_2$ then,

- $\texttt{wp}(op, \phi_1 \ \wedge\ \phi_2) = \psi_1 \ \wedge\ \psi_2$, and
- $\texttt{wp}(op, \phi_1 \ \vee\ \phi_2) = \psi_1 \ \vee\ \psi_2$. Note that this property holds only when S is a deterministic operation which is true in our programming model.

Property 2: Let ϕ_1 and ϕ_2 be the formulas such that ϕ_1 logically implies ϕ_2 then for every operation op, the formula $\texttt{wp}(op, \phi_1)$ logically implies $\texttt{wp}(op, \phi_2)$.

We say that a formula ϕ is *stable* with respect to a statement S if $\texttt{wp}(S, \phi)$ is logically equivalent to ϕ. In this paper, we use weakest preconditions to check the correctness of a trace with respect to some safety assertion. A trace σ reaching up to a safety assertion ϕ is safe if the execution of σ starting from the initial state $\mathcal{I}$ either 1) blocks (does not terminate) because of not satisfying some path conditions, or 2) terminates and the resulting state satisfies ϕ. The following lemmas clearly define the conditions, using weakest precondition axioms, for declaring a trace σ either safe or unsafe. Here $\sigma[\texttt{assume/assert}]$ denote the trace obtained by replacing every instruction of the form $\texttt{assume}(\phi)$ by $\texttt{assert}(\phi)$ in σ.

Lemma 1: For a trace σ, an initial program state $\mathcal{I}$ and a safety property ϕ, if $\texttt{wp}(\sigma[\texttt{assume/assert}], \neg\phi) \ \wedge\ \mathcal{I}$ is unsatisfiable then the execution of σ, starting from $\mathcal{I}$, either does not terminate or terminates in a state satisfying ϕ.

Lemma 2: For a trace σ, an initial program state $\mathcal{I}$ and a safety property ϕ, if $\texttt{wp}(\sigma[\texttt{assume/assert}], \neg\phi) \ \wedge\ \mathcal{I}$ is satisfiable then the execution of σ, starting from $\mathcal{I}$, terminates in a state not satisfying ϕ.

C. Alternating Finite Automata (AFA)

Alternating finite automata [1], [3] are a generalization of nondeterministic finite automata (NFA). An NFA is a five tuple $\langle S, \Sigma \cup \{\epsilon\}, \delta, s_0, S_F \rangle$ with a set of states S, ranged over by s, an initial state s_0, a set of accepting states S_F and a transition function $\delta : S \times \Sigma \cup \{\epsilon\} \rightarrow \mathbb{P}(S)$. For any state s of this NFA, the set of words accepted by s is inductively defined as $\texttt{acc}(s) =$

$\{a.\sigma \mid a \in \Sigma \cup \{\epsilon\}, \sigma \in \Sigma^*, \exists s'.\ s' \in \delta(s,a).\ \sigma \in \texttt{acc}(s')\}$ where $\epsilon \in \texttt{acc}(s)$ for all $s \in S_F$. Here, the existential quantifier represents the fact that there should exist at least one outgoing transition from s along which $a.\sigma$ gets accepted. An AFA is a six tuple $\langle S_\forall, S_\exists, \Sigma \cup \{\epsilon\}, \delta, s_0, S_F \rangle$ with Σ, s_0 and $S_F \subseteq S$ denoting the alphabet, initial state and the set of accepting states respectively. $S = S_\forall \bigcup S_\exists$ is the set of all states, ranged over by s and $\delta : S \times \Sigma \cup \{\epsilon\} \to \mathbb{P}(S)$ is the transition function. The set of words accepted by a state of an AFA depends on whether that state is an *existential state* (from the set $S_\exists$) or a *universal state* (from the set $S_\forall$). For an existential state $s \in S_\exists$, the set of accepted words is inductively defined in the same way as in NFA. For a universal state $s \in S_\forall$ the set of accepted words are $\texttt{acc}(s) = \{a.\sigma \mid a \in \Sigma \cup \{\epsilon\}, \forall s' \in \delta(s,a).\ \sigma \in \texttt{acc}(s')\}$ with $\epsilon \in \texttt{acc}(s)$ for all $s \in S_F$. Notice the change in the quantifier from $\exists$ to $\forall$. In the diagrams of AFA used in this paper, we annotate universal states with $\forall$ symbol and existential states with $\exists$ symbol. For a state s, let $\texttt{succ}(s,a) = \{S \mid (s,a,S) \in \delta\}$ be the set of a-successors of s. For an automaton $\mathcal{A}$, let $\mathcal{L}(\mathcal{A})$ be the language accepted by the initial state of that automaton. For any $\sigma \in \Sigma^*$ $|\sigma|$ denote the length of σ and $\text{rev}(\sigma)$ denote the reverse of σ.

III. Our Approach

The overall approach of this paper can be described in the following steps: (i) Given a concurrent program $\mathcal{P}$, construct all its interleaved traces represented by automaton $\mathcal{A}(\mathcal{P})$, as defined in Subsection II-A; (ii) Pick a trace σ and a safety property, say ϕ, to prove for this trace; (iii) Prove σ correct with respect to ϕ using Lemma 1 and Lemma 2 and generate a set of traces which are also *provably correct*. Let us call this set Tr'; (iv) Remove set Tr' from the set of traces represented by $\mathcal{A}(\mathcal{P})$ and repeat from Step (ii) until either all the traces in $\mathcal{P}$ are proved correct or an erroneous trace is found.

Step (iii) of this procedure, correctness of σ, can be achieved by checking the unsatisfiability of $\texttt{wp}(\sigma[\texttt{assume/assert}], \neg\phi) \wedge \mathcal{I}$. However, we are not only interested in checking the correctness of σ but also in constructing a set of traces which have a similar reasoning as of σ. Therefore, instead of computing $\texttt{wp}(\sigma[\texttt{assume/assert}], \neg\phi)$ directly from the weakest precondition axioms of Figure 4, we construct an AFA from σ and $\neg\phi$. Step (iv) is then achieved by applying automata-theoretic operations such as complementation and subtraction on this AFA. Notion of universal and existential states of AFA helps us in finding a set of sufficient dependencies used in the weakest precondition computation so that any other trace satisfying those dependencies gets captured by AFA. Subsequent subsections covers the construction, properties and use of this AFA in detail.

A. Constructing the AFA from a Trace and a Formula

Definition 1: An AFA constructed from a trace σ of a Program P and a formula ϕ is $\hat{\mathcal{A}}_{\sigma,\phi} = \langle S_\forall, S_\exists, \mathcal{OP}_\epsilon, s_0, S_F, \delta, \texttt{AMap}, \texttt{RMap} \rangle$, where,

1) $(\mathcal{OP}_\epsilon = \mathcal{OP} \cup \{\epsilon\})$ is the alphabet ranged over by *op*. Here $\mathcal{OP}$ is the set of instructions used in program P.

$\delta(s, op) =$

$$
\begin{array}{lll}
\{s'\} & \text{if} & \left\{\begin{array}{l} 1.\texttt{AMap}(s') = \texttt{wp}(\texttt{op}[\texttt{assume/assert}], \texttt{AMap}(s)), \\ 2.s \text{ is an existential state, and} \\ 3.\texttt{RMap}(s) = \texttt{RMap}(s').op.\sigma'' \\ \textbf{where } \sigma'' \text{ is the longest sequence s.t.} \\ \texttt{wp}(\sigma''[\texttt{assume/assert}], \texttt{AMap}(s)) = \texttt{AMap}(s) \end{array}\right. \\
 & & \qquad (\text{LITERAL-ASSN}) \\
\{s\} & \text{if} & \left\{\begin{array}{l} 1.\texttt{AMap}(s) = \texttt{wp}(\texttt{op}[\texttt{assume/assert}], \texttt{AMap}(s)), \text{ and} \\ 2.s \text{ is an existential state} \end{array}\right. \\
 & & \qquad (\text{LITERAL-SELF-ASSN}) \\
\{s_1, \cdots, s_k\} & \text{if} & \left\{\begin{array}{l} 1.\texttt{AMap}(s) = \bigwedge_k \phi_k \textbf{ or } \texttt{AMap}(s) = \bigvee_k \phi_k, \\ 2.\texttt{AMap}(s_k) = \phi_k, \\ 3.\forall k, \texttt{RMap}(s) = \texttt{RMap}(s_k), \\ 4.op = \epsilon \end{array}\right. \\
 & & \qquad (\text{COMPOUND-ASSN}) \\
\{\} & & \text{otherwise}
\end{array}
$$

Fig. 5: Transition function used in the Definition 1

Symbol ϵ acts as an identity element of concatenation and $\texttt{wp}(\epsilon, \phi) = \phi$.

2) $S = S_\forall \bigcup S_\exists$ is the largest set of states, ranged over by s s.t.
   a) Every state is annotated with a formula and a prefix of σ denoted by $\texttt{AMap}(s)$ and $\texttt{RMap}(s)$ respectively. State s_0 is the initial state such that $\texttt{AMap}(s_0) = \phi$, $\texttt{RMap}(s_0) = \sigma$.
   b) $s' \in S$ iff either of the following two conditions hold,
      - $\exists s \in S$ such that $\texttt{AMap}(s')$ is $\texttt{wp}(\texttt{op}[\texttt{assume/assert}], \texttt{AMap}(s))$, $\texttt{RMap}(s) = \texttt{RMap}(s').op.\sigma'$ and σ' is the largest suffix of $\texttt{RMap}(s)$ such that formula $\texttt{AMap}(s)$ is stable with respect to $\sigma'[\texttt{assume/assert}]$.
      - $\exists s \in S$ such that $\texttt{AMap}(s) = \bigwedge\{\phi_1, \cdots, \phi_k\}$ or $\texttt{AMap}(s) = \bigvee\{\phi_1, \cdots, \phi_k\}$, $\texttt{RMap}(s) = \texttt{RMap}(s')$, $\texttt{AMap}(s') = \phi'$ and $\phi' \in \{\phi_1, \cdots, \phi_k\}$.
   c) A state $s \in S$ is an existential state (universal state) iff $\texttt{AMap}(s)$ is a literal (compound formula).
3) $S_F \subseteq S$ is a set of accepting states such that $s \in S_F$ iff $\texttt{wp}(\texttt{RMap(s)}[\texttt{assume/assert}], \texttt{AMap}(s))$ is same as $\texttt{AMap}(s)$, i.e. $\texttt{AMap}(s)$ is *stable* with respect to $\texttt{RMap(s)}[\texttt{assume/assert}]$, and
4) Function $\delta : S \times \mathcal{OP}_\epsilon \to \mathbb{P}(S)$ is defined in Figure 5.

Following Point 2b, any state added to S is either annotated with a smaller $\texttt{RMap}$ or a smaller formula compared to the states already present in S. Further, every formula and trace σ is of finite length. Hence the set of states S is finite. By Point 2c of this construction, a state s where $\texttt{AMap}(s)$ is a compound formula, is always a universal state irrespective of whether $\texttt{AMap}(s)$ is a conjunction or a disjunction of clauses. The reason behind this decision will be clear shortly when we will use this AFA to inductively construct the weakest precondition $\texttt{wp}(\sigma[\texttt{assume/assert}], \phi)$. Note that we assume every formula is normalized in CNF.

Figure 7 shows an example trace $\sigma = \texttt{abApqPrcs}$ of Peterson's algorithm. This trace is picked from the Peterson's specification in Figure 3. To prove σ correct with respect to the safety formula $\phi \stackrel{def}{=} (\ell_2 = 2)$ we first construct $\hat{\mathcal{A}}_{\sigma,\neg\phi}$ which

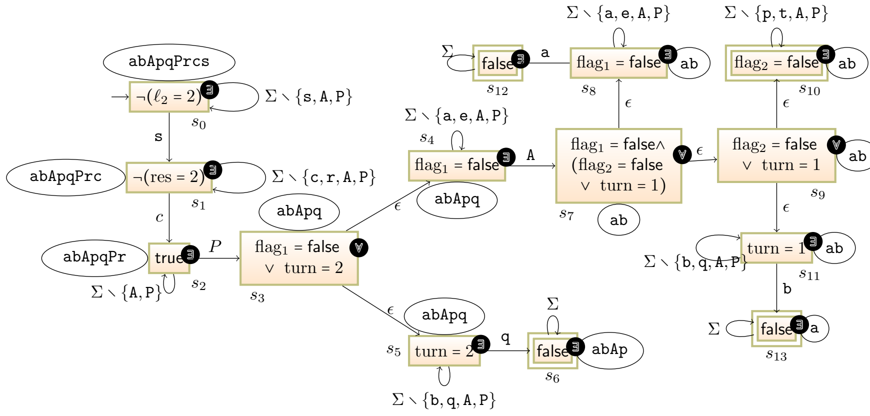


Fig. 6: AFA of trace given in Figure 6(b) and $\phi = \neg(\ell_2 = 2)$

a. flag_1:=true
b. turn:=2
A. assume($\text{flag}_2 = \text{false}$ || turn = 1)
p. flag_2:=true
q. turn:=1
P. assume($\text{flag}_1 = \text{false}$ || turn = 2)
r. res:=2
c. res:=1
s. ℓ_2:=res

Fig. 7: A trace from Peterson's algorithm

will later help us to derive $\texttt{wp}(\sigma[\texttt{assume/assert}], \neg\phi)$. This AFA is shown in Figure 6. For a state s, $\texttt{AMap}(s)$ is written inside the rectangle representing that state and $\texttt{RMap}(s)$ is written inside an ellipse next to that state. We show here some of the steps illustrating this construction.

1) By Definition 1, we have $\texttt{AMap}(s_0) = \neg(\ell_2 = 2)$ and $\texttt{RMap}(s_0) = \sigma = \texttt{abApqPrcs}$ for initial state s_0.
2) In a transition $\delta(s, op) = \{s'\}$ created by Rule LITERAL-ASSN the state s' is annotated with the weakest precondition of an operation *op*, taken from $\texttt{RMap}(s)$, with respect to $\texttt{AMap}(s)$. Operation *op* is picked in such a way that $\texttt{AMap}(s)$ is *stable* with respect to every other operation present after *op* in $\texttt{RMap}(s)$. Such transitions capture the inductive construction of the weakest precondition for a given ϕ and trace σ. Transition $\delta(s_0, \texttt{s}) = \{s_1\}$ in Figure 6 is created by this rule as $\texttt{wp}(\texttt{s}[\texttt{assume/assert}], \texttt{AMap}(s_0)) = \texttt{AMap}(s_1)$, and $\texttt{RMap}(s_0) = \texttt{RMap}(s_1).\texttt{s}$.
3) In any transition created by Rule COMPOUND-ASSN, say from s to $s_1, \cdots, s_k$, the states $s_1, \cdots s_k$ are annotated with the subformulae of $\texttt{AMap}(s)$. For example, transitions $\delta(s_3, \epsilon) = \{s_4, s_5\}$ and $\delta(s_7, \epsilon) = \{s_8, s_9\}$.
4) Transition $\delta(s_8, \texttt{a}) = \{s_{12}\}$ follows from the rule LITERAL-ASSN. Note that $\texttt{RMap}(s_{12})$ is empty and hence by Point 3 of Definition 1, s_{12} is an accepting state. Following the same reasoning, states s_6, s_{10} and s_{13} are also set as accepting states.
5) Rule LITERAL-SELF-ASSN adds a self transition at a state s on a symbol $op \in \mathcal{OP}_\epsilon$ such that $\texttt{AMap}(s)$ is *stable* with respect to $\texttt{op}[\texttt{assume/assert}]$. For example, transitions $\delta(s_0, op) = \{s_0\}$ where $op \in \mathcal{OP}_\epsilon \setminus \{\texttt{s}, \texttt{A}, \texttt{P}\}$.

The following lemma relates $\texttt{RMap}(s)$ at any state to the set of words accepted by s in this AFA.

Lemma 3: Given a $\sigma \in \mathcal{L}(\mathcal{A}(\mathcal{P}))$ and ϕ, let $\hat{\mathcal{A}}_{\sigma,\phi}$ be the AFA satisfying Definition 1. For every state s of this AFA, the condition $\text{rev}(\texttt{RMap}(s)) \in \text{acc}(s)$ holds.
This lemma uses the reverse of $\texttt{RMap}(s)$ in its statement because the weakest precondition of a sequence is constructed by scanning it from the end. This can be seen in the transition rule LITERAL-ASSN. As a corollary, $\text{rev}(\sigma)$ is also accepted by this AFA because by Definition 1, $\texttt{RMap}(s_0)$ is σ. ∎

$$
\texttt{HMap}(s) =
\begin{cases}
\texttt{AMap}(s) & \text{if } s \in S_F \\
& \text{(BASE-CASE)} \\
\bigwedge_k \texttt{HMap}(s_k) & \text{if } \delta(s,\epsilon) = \{s_1, \cdots, s_k\} \text{ and } \texttt{AMap}(s) = \bigwedge_k \texttt{AMap}(s_k) \\
& \text{(CONJ-CASE)} \\
\bigvee_k \texttt{HMap}(s_k) & \text{if } \delta(s,\epsilon) = \{s_1, \cdots, s_k\} \text{ and } \texttt{AMap}(s) = \bigvee_k \texttt{AMap}(s_k) \\
& \text{(DISJ-CASE)} \\
\texttt{HMap}(s') & \text{if } (s, op, \{s'\}) \in \delta \\
& \text{(LIT-CASE)}
\end{cases}
$$

Fig. 8: Rules for HMap construction

B. Constructing the weakest precondition from $\hat{\mathcal{A}}_{\sigma,\phi}$

After constructing $\hat{\mathcal{A}}_{\sigma,\phi}$ the rules given in Figure 8 are used to inductively construct and assign a formula, $\texttt{HMap}(s)$, to every state s of $\hat{\mathcal{A}}_{\sigma,\phi}$. Figure 9 shows the AFA of Figure 6 where states are annotated with formula $\texttt{HMap}(s)$. This formula is shown in the ellipse beside every state. For better readability we do not show $\texttt{RMap}(s)$ in this figure.

Following Rule BASE-CASE, HMap of s_6, s_{12}, and s_{13} are set to false whereas $\texttt{HMap}(s_{10})$ is set to $\text{flag}_2 = \text{false}$. By Rule LIT-CASE, HMap of s_5, s_8 and s_{11} are also set to false. After applying Rule DISJ-CASE for transition $\delta(s_9, \epsilon) = \{s_{10}, s_{11}\}$, $\texttt{HMap}(s_9)$ is set to $\text{flag}_2 = \text{false}$. Similarly, using Rule CONJ-CASE we get $\texttt{HMap}(s_7)$ as false. Finally, $\texttt{HMap}(s_0)$ is also set to false. HMap constructed inductively in this manner satisfies the following property;

Lemma 4: Let $\hat{\mathcal{A}}$ be an AFA constructed from a trace and a post condition as in Definition 1 then for every state s of this AFA and for every word σ accepted by state s, $\texttt{HMap}(s)$ is logically equivalent to $\texttt{wp}(\text{rev}(\sigma)[\texttt{assume/assert}], \texttt{AMap}(s))$.
First consider the accepting states of $\hat{\mathcal{A}}$. For example, states s_6, s_{10}, s_{12} and s_{13} of Figure 9. Following the definition of an accepting state and by the self-loop adding transition rule LITERAL-SELF-ASSN, every word σ accepted by such an accepting state s satisfies $\texttt{wp}(\text{rev}(\sigma)[\texttt{assume/assert}], \texttt{AMap}(s)) = \texttt{AMap}(s)$. Therefore, setting $\texttt{HMap}(s)$ as $\texttt{AMap}(s)$ for these accepting states, as done in Rule BASE-CASE completes the proof for accepting states. Now consider a

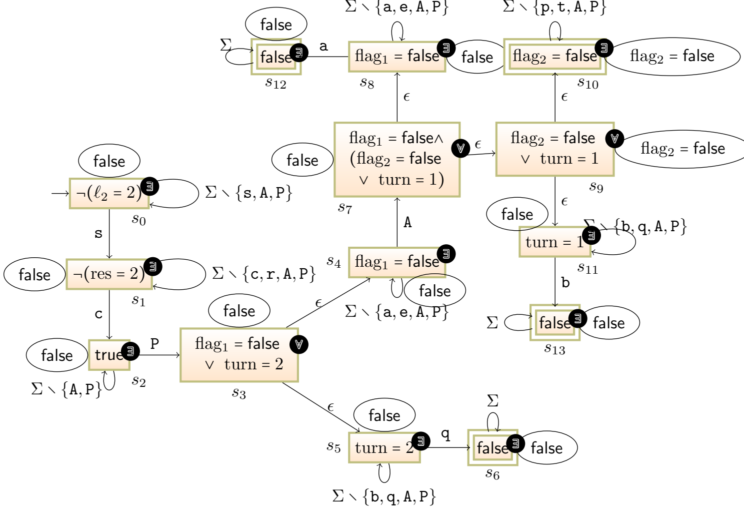


Fig. 9: HMap construction for the running example

Algorithm 1: Converting universal to existential states while preserving Lemma 4

Data: Input AFA $\langle S_\forall, S_\exists, \mathcal{OP} \cup \{\epsilon\}, s_0, S_F, \texttt{AMap}, \texttt{RMap}\rangle$
Result: Modified AFA

1. Let s be a state in AFA such that $s \in S_\forall$, $\delta(s,\epsilon) = \{s_1, \cdots, s_k\}$, $\texttt{HMap}(s)$ is unsatisfiable, and $\texttt{AMap}(s) = \bigwedge_k \texttt{AMap}(s_k)$;
2. Let $\texttt{Unsatcore}(s) \subseteq \mathbb{P}(\{s_1, \cdots, s_k\})$ such that $\{s'_1, \cdots, s'_n\} \in \texttt{Unsatcore}(s)$ iff $\{\texttt{HMap}(s'_1), \cdots, \texttt{HMap}(s'_n)\}$ is a minimal unsat core of $\bigwedge_k \texttt{HMap}(s_k)$;
3. Create an empty set U;
4. **foreach** $\{s'_1, \cdots, s'_n\} \in \texttt{Unsatcore}(s)$ **do**
5. create a new universal state $s_u \in S_\forall$ and add it to the set U;
6. Set $\texttt{AMap}(s_u) = \bigwedge_i \texttt{AMap}(s'_i)$;
7. Set $\texttt{HMap}(s_u) = \bigwedge_i \texttt{HMap}(s'_i)$;
8. Add a transition by setting $\delta(s_u, \epsilon) = \{s'_1, \cdots, s'_n\}$;
9. **end**
10. Remove transition $\delta(s,\epsilon) = \{s_1, \cdots, s_k\}$;
11. Convert s to an existential state;
12. Add a transition from s on ϵ by setting $\delta(s,\epsilon) = \texttt{U}$ where U is the set of universal states created one for each element of $\texttt{Unsatcore}(s)$;

---

state s with transition $\delta(s,\epsilon) = \{s_1, \cdots, s_k\}$, created using Rule COMPOUND-ASSN, and let σ be a word accepted by s. By construction, s must be a universal state and hence σ must be accepted by each of $s_1, \cdots, s_k$ as well. Using this lemma inductively on successor states $s_1, \cdots, s_k$ (induction on the formula size) we get $\texttt{wp}(\sigma[\texttt{assume/assert}], \texttt{AMap}(s_i)) = \texttt{HMap}(s_i)$ for all $i \in \{1 \cdots k\}$. Now we can apply Property 1 depending on whether $\texttt{AMap}(s)$ is a conjunction or a disjunction of $\texttt{AMap}(s_k)$. By replacing $\texttt{AMap}(s)$ with $\bigvee_k \texttt{AMap}(s_k) (\bigwedge_k \texttt{AMap}(s_k))$ and $\texttt{HMap}(s)$ with $\bigvee_k \texttt{HMap}(s_k) (\bigwedge_k \texttt{HMap}(s_k))$ completes the proof. Note that, making s as a universal state when $\texttt{AMap}(s)$ is either a conjunction or a disjunction allowed us to use Property 1 in this proof. Otherwise, if we make s an existential state when $\texttt{AMap}(s)$ is a disjunction of formulae then we can not prove this lemma for states where $\texttt{HMap}(s)$ is constructed using Rule DISJ-CASE. ∎

This lemma serves two purposes. First, it checks the correctness of a trace σ w.r.t. a safety property for which this AFA was constructed. If $\texttt{HMap}(s_0) \land \mathcal{I}$ is unsatisfiable, as in our Peterson's example trace, then σ is declared as correct. Second, it guarantees that every trace accepted by this AFA, that is present in the set of all traces of $\mathcal{P}$, is also safe and hence we can skip proving their correctness altogether. Removing such traces is equivalent to subtracting the language of this AFA from the language representing the set of all traces. Then a natural question to ask is if we can increase the set of accepted words of this AFA while preserving Lemma 4.

C. Enlarging the set of words accepted by $\hat{A}_{\sigma,\phi}$

Converting Universal States to Existential States Figure 10 shows an example trace σ = abcde obtained from the parallel composition of some program P. Figure 11 shows the AFA constructed for σ and ϕ as $S < t \land z < x$. From Lemma 4 we get $\texttt{wp}(\sigma, \phi)$ as false. Note that the $\texttt{wp}(\sigma, S < t)$ and $\texttt{wp}(\sigma, z < x)$ are unsatisfiable, i.e. we have two ways to derive the unsatisfiability of $\texttt{wp}(\sigma, \phi)$; one is due to the operation d, and the other is due to the operation a followed by operation e. In this example, any word that enforces either of these two ways will derive false as the weakest precondition. For example, the sequence σ' = adcbe is not accepted by the AFA of Figure 11 but the condition $\texttt{wp}(\texttt{rev}(\sigma'), \neg\phi)$ = false follows from $\texttt{wp}(\texttt{d}, \neg\phi)$ = false which is already captured in the AFA of Figure 11. Note that states s_1 and s_2 in Figure 11 are annotated with unsatisfiable HMap assertion. It seems sufficient to take any one of these branches to argue the unsatisfiability of $\texttt{HMap}(s_0)$ because $\texttt{HMap}(s_0)$, by definition, is a *conjunction* of $\texttt{HMap}(s_1)$ and $\texttt{HMap}(s_2)$. Therefore, if we convert s_0, a universal state, to an existential state then the modified AFA will accept adcbe. Let us look at Algorithm 1 to see the steps involved in this transformation. This algorithm picks a universal state s such that $\texttt{AMap}(s)$ is a *conjunction* of clauses and only a subset of its successors are sufficient to make $\texttt{HMap}(s)$ unsatisfiable. State s_0 of Figure 11 is one such state. For each such minimal subsets of its successors, this algorithm creates a universal state, as shown in Line 5 of this algorithm. It is easy to see that $\texttt{HMap}(s_u)$ is also unsatisfiable. Before adding $\delta(s_u, \epsilon) = \{s'_1, \cdots, s'_n\}$ transition in AFA this algorithm sets $\texttt{AMap}(s_u)$ as $\bigwedge_i \texttt{AMap}(s'_i)$. By construction, every word accepted by s_u must be accepted by $s'_1, \cdots, s'_n$. Each of these states $s'_1, \cdots, s'_n$ satisfy Lemma 4. Hence Lemma 4 continues to hold for these newly created universal states as well. Now consider a newly created transition $(s, \epsilon, \texttt{U})$ in Line 12. For any state $s'' \in \texttt{U}$, $\texttt{AMap}(s)$ logically implies $\texttt{AMap}(s'')$ because s'' represents a subset of the original successors of s, viz. $s_1, \cdots, s_k$. As s is now an existential state, any word accepted by s, say σ', is accepted by at least one state in U, say s'. Using Lemma 4 on s', $\texttt{HMap}(s')$ is logically equivalent to $\texttt{wp}(\texttt{rev}(\sigma')[\texttt{assume/assert}], \texttt{AMap}(s'))$. Using unsatisfiability of $\texttt{HMap}(s)$ and $\texttt{HMap}(s')$ and the monotonicity property of the weakest precondition, Property 2, we get that $\texttt{HMap}(s)$ is logically equivalent to

a. Y:=x + 1
b. W:=t
c. z:=W
d. S:=t + 1
e. z:=Y

Fig. 10: Example Trace

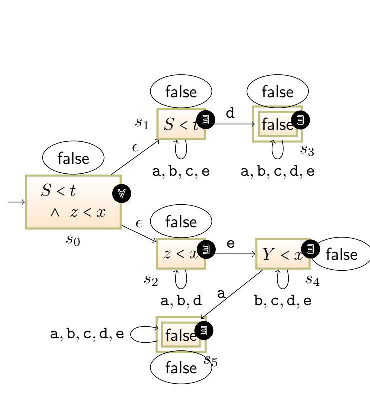


Fig. 11: AFA for σ given in Figure 10

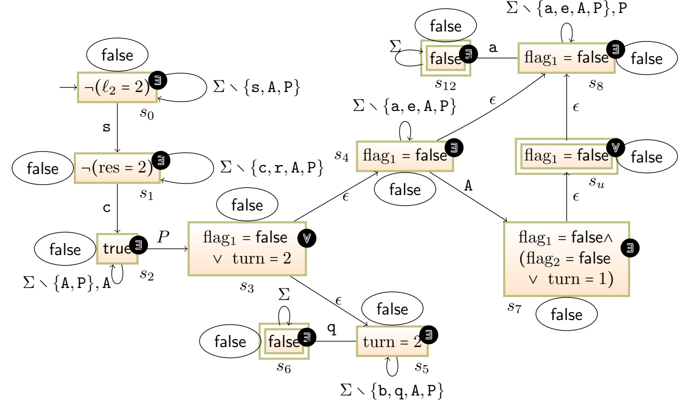


Fig. 12: AFA of Figure 9 after Modification

$\delta(s, op) = \delta(s, op) \cup \{s'\}$ iff

$$\begin{cases} \texttt{HMap}(s) \text{ and } \texttt{HMap}(s') \text{ are unsatisfiable,} \\ (s) \text{ is a literal, and} \\ \texttt{op[assume/assert]AMap}(s) \Rightarrow \texttt{AMap}(s') \\ \qquad\qquad\qquad\qquad \text{(RULE-UNSAT)} \\ \textbf{OR} \\ \texttt{HMap}(s) \text{ and } \texttt{HMap}(s') \text{ are valid} \\ (s) \text{ is a literal, and} \\ (s') \Rightarrow \texttt{wp}(\texttt{op[assume/assert]}, \texttt{AMap}(s)) \\ \qquad\qquad\qquad\qquad \text{(RULE-VALID)} \end{cases}$$

Fig. 13: Rules for adding more edges

$\texttt{wp}(\mathsf{rev}(\sigma')\texttt{[assume/assert]}, \texttt{AMap}(s))$.

Adding More transitions to $\hat{\mathcal{A}}_{\sigma,\phi}$ using the Monotonicity Property of the Weakest Precondition We further modify $\hat{\mathcal{A}}_{\sigma,\phi}$ by adding more transitions. For any two states s and s' such that $\texttt{AMap}(s)$ and $\texttt{AMap}(s')$ are literals, both $\texttt{HMap}(s)$ and $\texttt{HMap}(s')$ are unsatisfiable, and there exists a symbol a (can be ϵ as well) such that $\texttt{wp}(\texttt{a[assume/assert]}, \texttt{AMap}(s))$ logically implies $\texttt{AMap}(s')$, an edge labeled a is added from s to s'. This transformation also preserves Lemma 4 following the same monotonicity property, Property 2 used in the previous transformation. Similar argument holds when $\texttt{HMap}(s)$ and $\texttt{HMap}(s')$ are valid and $\texttt{AMap}(s') \Rightarrow \texttt{wp}(a, \texttt{AMap}(s))$ holds. The rules of adding edges are shown in Figure 13.

Figure 12 shows the AFA of Figure 9 modified by above transformations. Rule RULE-UNSAT adds an edge from s_4 to s_8 on symbol ϵ because $\texttt{HMap}(s_4)$ and $\texttt{HMap}(s_8)$ are unsatisfiable and $\texttt{wp}(\epsilon, \texttt{AMap}(s_4))$ logically implies $\texttt{AMap}(s_8)$. Same rule also adds a self loop at s_8 on operation P and a self loop at s_2 on operation A. Transformation by Algorithm 1 removes the transition from s_7 to s_9 and all other states reachable from s_9. Now consider a trace $\texttt{rev(abpqPArcs)}$ that is accepted by this modified AFA in Figure 12 but was not accepted by the original AFA of Figure 9. Note that $\texttt{wp}(\texttt{abpqPArcs}, \neg(\ell_2 = 2))$ is unsatisfiable and this is a direct consequence of Lemma 4. Because of the transformations presented in this sub-section we do not need to reason about this trace separately.

Algorithm 2: Algorithm to check the safety assertions of a concurrent program P

Input: A concurrent program $\mathcal{P} = \{p_1, \cdots, p_n\}$ with safety property map Assrn
Result: *yes*, if program is safe else a counterexample
1 Let $\mathcal{A}(\mathcal{P})$ bet the automaton that represents the set of all the SC executions of P (as defined in Section II);
2 Set $\texttt{tmp} := \mathcal{L}(\widehat{\mathcal{A}(P)})$;
3 **while** tmp *is not empty* **do**
4 Let $\sigma \in \texttt{tmp}$ with ϕ as a safety assertion to be checked;
5 Let $\hat{\mathcal{A}}_{\sigma,\neg\phi}$ be the AFA constructed from σ and $\neg\phi$;
6 **if** $\mathcal{I}\ \wedge\ \texttt{HMap}(s_0)$ *is satisfiable* **then**
7 σ is a valid counterexample violating ϕ;
8 **return** (σ);
9 **else**
10 Let $\hat{\mathcal{A}}'$ be the AFA modified by proposed transformations;
11 $\texttt{tmp} := \texttt{tmp} \setminus Rev$, where $Rev = \{\mathsf{rev}(\sigma) \mid \sigma \in \mathcal{L}(\hat{\mathcal{A}}')\}$;
12 **end**
13 **end**
14 **return** (*yes*);

D. *Putting All Things Together For Safety Verification*

In Algorithm 2, all the above steps are combined to check if all the SC executions of a concurrent program P satisfy the safety properties specified as assertions.

Theorem 1: Let $P = (p_1, \cdots, p_n)$ be a finite state program (with or without loops) with associated assertion maps Assrn_{p_i}. All assertions of this program hold iff Algorithm 2 returns *yes*. If the algorithm returns a word σ then at least one assertion fails in the execution of σ.
Detailed proofs of the lemmas and theorems of this paper are available at https://arxiv.org/abs/1506.07635

IV. EXPERIMENTAL EVALUATION

We implemented our approach in a prototype tool, ProofTraPar. This tool reads the input program written in a custom format. In future, we plan to use off-the-shelf parsers such as CIL or LLVM to remove this dependency. Individual processes are represented using finite state automata. We use an automata library, libFAUDES [5] to carry out operations on automata. After constructing the AFA from a trace we first remove ϵ transitions from this AFA. This is followed by adding

Program	ProofTraPar	THREADER[10]	Lazy-CSeq[11]
Peterson.safe	**0.3**	3.2	3.1
Dekker.safe	**1.1**	1.7	4.2
Lamport.safe	**2.4**	47	5.1
Szymanksi.safe	**3**	12.8	4
TimeVarMutex.safe	**0.76**	8.56	4.2
RWLock.safe (2R+2W)	8.8	140	**6.7**
RWLock.unsafe (2R+2W)	3.8	153	**0.7**
Qrcu.safe (2R+1W)	**20**	–	41
Qrcu.unsafe (2R+1W)	13.8	76	**1.1**

Fig. 14: Comparison with THREADER[10], and Lazy-CSeq [11] (Time in seconds)

additional edges in AFA using proposed transformations. Instead of reversing this AFA (as in Line 11 of Algorithm 2) we subtract it with an NFA that represents the reversed language of the set of all traces. This avoids the need of reversing an AFA. Note that we do not convert our AFA to NFA but rather carry out intersection and complementation operations (needed for language subtraction operation) directly on AFA. Our tool uses the Z3 [4] theorem prover to check the validity of formulae during AFA construction. `ProofTraPar` can be accessed from the repository https://github.com/chinuhub/ProofTraPar.git.

Figure 14 tabulates the result of verifying *pthread-atomic* category of SV-COMP benchmarks using our tool, THREADER [10] and Lazy-CSeq [11]. These tools were the winners in the concurrency category of the software verification competition of 2013 (THREADER), 2014 and 2015 (Lazy-CSeq). Dash (–) denotes that the tool did not finish the analysis within 15 minutes. Numbers in bold text denote the best time of that experiment. Safe/Unsafe versions of these programs are labeled with *.safe*/*.unsafe*. Except on Reader-Writer Lock and on unsafe version of QRCU(Quick Read Copy Update), our tool performed better than the other two tools. On unsafe versions, our approach took more time to find out an erroneous trace as compared to Lazy-CSeq [11]. Context-bounded exploration by Lazy-CSeq [11] and the presence of bugs at a shallow depth seem to be a possible reason behind this performance difference. Introducing priorities while picking traces in order to make our approach efficient in bug-finding is left open for future work.

V. Related Work

Verifying the safety properties of a concurrent program is a well studied area. Automated verification tools which use model checking based approaches employ optimizations such as Partial Order Reductions (POR) [13], [8], [7] to handle larger number of interleavings. These optimizations also selectively check a representative set of traces among the set of all interleavings. POR based methods were traditionally used in bug finding but recently they have been extended efficiently, using abstraction and interpolants, for proving programs correct [14]. The technique presented in this paper, using AFA, can possibly be used to keep track of partial orders in POR based methods. In [15], a formalism called concurrent trace program (CTP) is defined to capture a set of interleavings corresponding to a concurrent trace. CTP captures the partial orders encoded in that trace. Corresponding to a CTP, a formula ϕ_{ctp} is defined such that ϕ_{ctp} is satisfiable iff there is a feasible linearization of the partial orders encoded in CTP that violates the given property. Our AFA is also constructed from a trace but unlike CTP it only captures those different interleavings which guarantee the same proof outline. Recently in [9], a formalism called *HB-formula* has been proposed to capture the set of happens-before relations in a set of executions. This relation is then used for multiple tasks such as synchronization synthesis[2], bug summarization and predicate refinement. Since the AFA constructed by our algorithm can also be represented as a boolean formula (universal states correspond to conjunction and existential states correspond to disjunction) that encodes the ordering relations among the participating events, it will be interesting to explore other usages of this AFA along the lines of [9].

VI. Conclusion and Future Work

We presented a trace partitioning based approach for verifying safety properties of a concurrent program. To this end, we introduced a novel construction of an alternating finite automaton to capture the proof of correctness of a trace in a program. We also presented an implementation of our algorithm which compared competitively with existing state-of-the-art tools. We plan to extend this approach for parameterized programs and programs under relaxed memory models. We also plan to investigate the use of interpolants with weakest precondition axioms to incorporate abstraction for handling infinite state programs.

References

[1] J.A. Brzozowski and L. L. Ernst. On equations for regular languages, finite automata, and sequential networks. *TCS*, 10:19–35, 1980.

[2] P. Cerný, E.M. Clarke, T. A. Henzinger, A. Radhakrishna, L. Ryzhyk, R. Samanta, and T. Tarrach. From non-preemptive to preemptive scheduling using synchronization synthesis. In *CAV*, 2015.

[3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.

[5] Bernd Opitz et al. libfaudes-discrete event system library.

[6] A. Farzan, Z. Kincaid, and A. Podelski. Inductive data flow graphs. In *POPL*, pages 129–142, 2013.

[7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[8] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer, 1996.

[9] A. Gupta, T. A. Henzinger, A. Radhakrishna, R. Samanta, and T. Tarrach. Succinct representation of concurrent trace sets. In *POPL*, 2015.

[10] A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *CAV*, pages 412–417, 2011.

[11] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *CAV*, volume 8559 of *LNCS*, pages 585–602. Springer, 2014.

[12] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, September 1979.

[13] D. Peled. All from one, one for all: On model checking using representatives. In *CAV*, pages 409–423. Springer-Verlag, 1993.

[14] B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with Impact. In *FMCAD*, pages 210–217. IEEE, 2013.

[15] C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In Ana Cavalcanti and DennisR. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *LNCS*, pages 256–272. Springer Berlin Heidelberg, 2009.