Incremental Verification Using Trace Abstraction[⋆]

Bat-Chen Rothenberg[1], Daniel Dietsch[2], and Matthias Heizmann[2]

[1] Technion - Israel Institute of Technology, Israel
batg@cs.technion.ac.il
[2] University of Freiburg, Germany
{dietsch,heizmann}@informatik.uni-freiburg.de

Abstract. Despite the increasing effectiveness of model checking tools, automatically re-verifying a program whenever a new revision of it is created is often not feasible using existing tools. Incremental verification aims at facilitating this re-verification, by reusing partial results. In this paper, we propose a novel approach for incremental verification that is based on trace abstraction. Trace abstraction is an automata-based verification technique in which the program is proved correct using a sequence of automata. We present two algorithms that reuse this sequence across different revisions, one eagerly and one lazily. We demonstrate their effectiveness in an extensive experimental evaluation on a previously established benchmark set for incremental verification based on different revisions of device drivers from the Linux kernel. Our algorithm is able to achieve significant speedups on this set, compared to both stand-alone verification and previous approaches.

1 Introduction

Manual detection of bugs in software is extremely time consuming and requires expertise and close acquaintance with the code. Yet, for some applications, delivering a bug-free product is crucial. Using automated program verification tools is a useful means to ease the burden. Despite the increasing effectiveness of such tools, advancements in technology of the past decade have given rise to new challenges. Modern software consists of thousands of lines of code and is developed by dozens of developers at a time. As a result, the software update rate is extremely high and dozens or even hundreds of successive program versions (also called revisions) are created every day. Automatically re-verifying the entire program whenever a new revision is created is often not feasible using existing tools.

Incremental verification is a methodology designed to make re-verification realistic. When a program revision undergoes incremental verification, changes made from the previous revision are taken into account in an attempt to limit

[⋆] This research was partially supported by the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber bureau.

the analysis to only the parts of the program that need to be reanalyzed. Partial verification results obtained from previous revisions can help accomplish this task and can also be used to make analysis more effective.

The development of incremental verification techniques is a long-standing research topic (e.g., see [22,6,20,8,23,19,15]). The main challenge these techniques face is deciding which information to pass on from the verification of one revision to another, and to find effective ways to reuse this information. The proposed solutions vary, based on the underlying non-incremental verification technique used. For example, the technique proposed by He, Mao, and Wang [15] is based on assume-guarantee reasoning, and thus suggests reusing contextual assumptions, whereas the technique by Sery, Fedyukovich, and Sharygina [23] is based on bounded model checking using function summaries, and thus suggests reusing these summaries.

In this paper, we propose a new technique for incremental verification, which is based on the verification method of Heizmann, Hoenicke and Podelski [16,17]. At the basis of this verification method is the idea of looking at the basic statements of the program, i.e., its assignments and conditions, as letters of a finite alphabet. Following this point of view, the paths of the program can be seen as words over this alphabet; the program itself can be seen as a finite automaton whose states are the program locations, and whose language is a set of paths. The way the method works is by constructing an abstraction of the set of infeasible program paths, called a *trace abstraction*, which is a sequence of automata over the alphabet of statements. Our suggestion is to use this trace abstraction for incremental verification. We believe that some of its properties, which we will present in later sections, make it an ideal candidate for reuse.

The paper is organized as follows: In Section 2 we will provide notations and formal definitions. Then, in Section 3, we will briefly review the work of [16,17] on which our incremental approach is based. Next, in Section 4, we will present our approach, and in Section 5 we will discuss our implementation details, and present extensive experimental results. Finally, in Section 6 we will survey related work, and in Section 7 we will conclude.

2 Preliminaries

In this section we will present the formal setting of our work. Basic concepts from the world of verification, such as a program and program correctness, will be defined in terms of formal languages and automata.

Traces. Throughout the paper, we assume the existence of a fixed set of statements, ST. The reader should think of this set as the set of all possible statements one can compose in a given programming language. An *alphabet* is a finite non-empty subset of ST. A *trace* over the alphabet Σ, denoted π, is an arbitrary word over Σ (i.e., $\pi \in \Sigma^*$).

Programs. It is common to represent a program using its control flow graph (CFG). The set of vertices of the CFG is the set of program locations L, which

```
$\ell_0$: assume p!=0;
$\ell_1$: while(n>=0) {
$\ell_2$:      assert p!=0;
          if(n==0) {
$\ell_3$:          p:=0;
          }
$\ell_4$:      n--;
     }
```

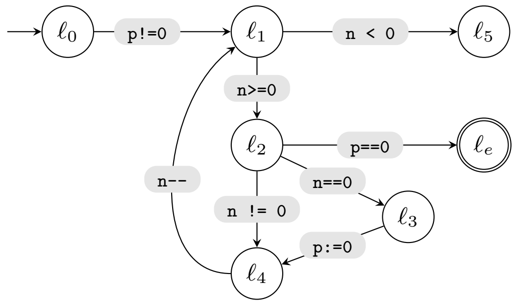


Fig. 1: Pseudo-code of a program P_{ex1} and its control-flow automaton $\mathcal{A}_{\mathcal{P}_{\text{ex1}}}$.

contains a distinguished initial location, l_i, and a subset of distinguished error locations, L_e. Edges of the CFG are labeled with statements of the program. An edge (l_j, s, l_k) appears in the graph iff the control of the program reaches location l_j, i.e., iff it is possible to continue to location l_k if the statement s executes successfully. A trace is an *error trace* of the program if it labels a path from l_i to some error location $l_e \in L_e$ in this graph.

In our setting, we prefer to view the program as an automaton over the alphabet of statements, instead of a graph. Formally, we define a *program* $\mathcal{P}$ as an automaton $(Q, \Sigma, q_0, \delta, F)$, called a *control-flow automaton*, where:

1. Q, the (finite) set of automaton states, is the set of all program locations L.
2. Σ, the alphabet of the automaton, is the set of all statements that appear in the program. Note that this set is indeed an alphabet according to our previous definition (i.e., $\Sigma \subseteq ST$).
3. q_0, the initial state of the automaton, is the initial location l_i.
4. δ, the transition relation, is a subset of $L \times \Sigma \times L$ containing exactly those triples that are edges of the CFG.
5. F, the set of final states, is the set of error locations, L_e.

By construction, the language of this automaton, $\mathcal{L}(\mathcal{P})$, is the set of error traces of the program.

Example 1. Figure 1 presents the pseudo-code of a program P_{ex1}, along with its control-flow automaton, $\mathcal{A}_{\mathcal{P}_{\text{ex1}}}$. The correctness of this program is specified via the assert statement at location ℓ_2: every time this location is reached, the value of the variable `p` must not equal 0. Thus, modeling of the assert statement is done using an edge labeled with the negation of the assertion (here, `p==0`) to a fresh error location, ℓ_e. The initial state of the automaton is the entering point of the program, ℓ_0, and the only accepting state is ℓ_e.

Correctness. We assume a fixed set of predicates Φ, which comes with a binary entailment relation. If the pair (φ_1, φ_2) belongs to the entailment relation, we say that φ_1 *entails* φ_2 and we write $\varphi_1 \models \varphi_2$. We also assume a fixed set HT of triples of the form $(\varphi_1, s, \varphi_2)$, where $\varphi_1, \varphi_2 \in \Phi$ and $s \in ST$. A triple $(\varphi_1, s, \varphi_2)$ is said to be a *valid Hoare triple* if it belongs to HT. In this case, we write $\{\varphi_1\}s\{\varphi_2\}$. The set of valid Hoare triples with $s \in \Sigma$ is denoted HT_Σ. Given a set $S \subseteq HT$, we denote by Φ_S the set of predicates that appear in S (i.e., all predicates that are the first or the last element of some triple in S).

Next, we extend the notion of validity from statements to traces. Given a trace $\pi = s_1 \cdots s_n$, where $n \geq 1$, the triple $(\varphi_1, \pi, \varphi_{n+1})$ is *valid* (and we write $\{\varphi_1\}\pi\{\varphi_{n+1}\}$), iff there exists a sequence of predicates $\varphi_2 \cdots \varphi_n$ s.t. $\{\varphi_i\}s_i\{\varphi_{i+1}\}$ for all $1 \leq i \leq n$. For an empty trace π (a trace of length 0), the triple (φ, π, φ') is *valid* iff φ entails φ'.

In order to define correctness, we also assume the existence of a pair of specific predicates from Φ, *true* and $false$. A trace π is *infeasible* if $\{true\}\pi\{false\}$. The set of all infeasible traces over the alphabet Σ is denoted INFEASIBLE_Σ. Finally, a program $\mathcal{P}$ is said to be *correct* if all error traces of it are infeasible. That is, if $\mathcal{L}(\mathcal{P}) \subseteq \text{INFEASIBLE}_\Sigma$, where Σ is the alphabet of the program.

3 Verification Using Trace Abstraction

In this section we will review the work of [16] and [17], which presents an automata-based approach for verification, upon which our incremental verification scheme is based. Even though some of the notions had to be adapted to our setting, all relevant theorems remain valid.

3.1 Floyd-Hoare Automata

We begin by introducing the notion of a Floyd-Hoare automaton, presented in [17], and describing some of its key properties. Intuitively, a Floyd-Hoare automaton is an automaton over an alphabet Σ whose states can be mapped to predicates from Φ and whose transitions can be mapped to valid Hoare triples. The motivation behind this definition is that we want Floyd-Hoare automata to accept only infeasible traces, by construction. Formally, we use the following definition:

Definition 1 (Floyd-Hoare automaton) *A Floyd-Hoare automaton is a tuple*

$$\mathcal{A} = (Q, \Sigma, q_0, \delta, F, \theta)$$

where Q is a finite set of states, Σ is an alphabet, $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $F \subseteq Q$ is the set of final states, and $\theta : Q \to \Phi$ is a mapping from states to predicates s.t. the following conditions hold:

1. *$\theta(q_0) = true$.*
2. *For every $q \in F$, $\theta(q) = false$.*
3. *For every triple $(q_i, s, q_j) \in \delta$, $\{\theta(q_i)\}s\{\theta(q_j)\}$.*

The function θ is called the *annotation* of $\mathcal{A}$. The image of θ (i.e., the set of all predicates $\varphi \in \Phi$ s.t. there exists a $q \in Q$ for which $\theta(q) = \varphi$) is called the *predicate set* of $\mathcal{A}$ and is denoted $\Phi_{\mathcal{A}}$.

Theorem 1 ([17, page 12]) *Every trace accepted by a Floyd-Hoare automaton $\mathcal{A}$ is infeasible. That is, for every Floyd-Hoare automaton $\mathcal{A}$ over Σ,*

$$L(\mathcal{A}) \subseteq \text{INFEASIBLE}_{\Sigma}$$

· In what follows, we define a mapping from Floyd-Hoare automata to sets of valid Hoare triples, and vice versa, using a pair of functions, α and β. The function α is a function from sets of valid Hoare triples to Floyd-Hoare automata. A set S of valid Hoare triples over Σ is mapped by α to the Floyd-Hoare automaton $\mathcal{A}_S = (Q_S, \Sigma, q_{0S}, \delta_S, F_S, \theta_S)$ where:

- $Q_S = \{q_\varphi | \varphi \in \Phi_S\} \cup \{q_{true}, q_{false}\}$.
- $q_{0S} = q_{true}$
- $\delta_S = \{(q_{\varphi_1}, s, q_{\varphi_2}) | (\varphi_1, s, \varphi_2) \in S\}$
- $F_S = \{q_{false}\}$
- $\forall q_\varphi \in Q_S \quad \theta_S(q_\varphi) = \varphi$

Note that this is indeed a Floyd-Hoare automaton according to definition 1, since S contains only valid Hoare triples.

The function β is a function from Floyd-Hoare automata to sets of valid Hoare triples. Given a Floyd-Hoare automaton $\mathcal{A} = (Q, \Sigma, q_0, \delta, F, \theta)$, β maps $\mathcal{A}$ to the set $\{(\theta(q_i), s, \theta(q_j)) \,|\, (q_i, s, q_j) \in \delta\}$. By definition 1 (specifically, by requirement number 3 of θ), this set contains only valid Hoare triples.

3.2 Automata-Based Verification

Next, we describe how Floyd-Hoare automata can be used to verify programs via trace abstraction [16]. Formally, a *trace abstraction* is a tuple of Floyd-Hoare automata $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ over the same alphabet Σ. The alphabet Σ is referred to as *the alphabet of the trace abstraction.* We say that a program $\mathcal{P}$ *is covered by* $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ if $\mathcal{P}$ and $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ are over the same alphabet and $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(A_1) \cup \ldots \cup \mathcal{L}(A_n)$.

Theorem 2 ([16, page 7]) *Given a program $\mathcal{P}$, if there exists a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ s.t. $\mathcal{P}$ is covered by $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$, then $\mathcal{P}$ is correct.*

Theorem 2 implies a way to verify a program $\mathcal{P}$, namely, by constructing a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ s.t. $\mathcal{P}$ is covered by $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$. This is realized in [16] in an algorithm that is based on the counter-example guided abstraction refinement (CEGAR) paradigm (Fig. 2). Initially, the trace abstraction is an empty sequence of automata, and then it is iteratively refined by adding automata, until the program is covered by the trace abstraction.

Each iteration consists of two phases: validation and refinement. During the validation phase, we check whether the equation $\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n}) = \emptyset$ holds.

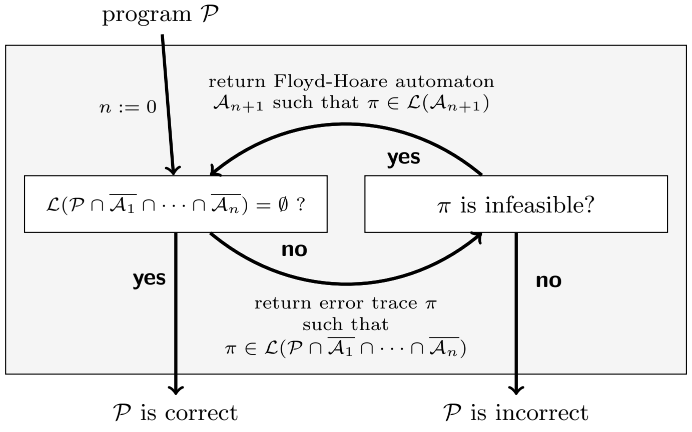

program $\mathcal{P}$
$n := 0$
return Floyd-Hoare automaton $\mathcal{A}_{n+1}$ such that $\pi \in \mathcal{L}(\mathcal{A}_{n+1})$
yes
$\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n}) = \emptyset$?
π is infeasible?
no
yes
no
return error trace π such that $\pi \in \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$
$\mathcal{P}$ is correct
$\mathcal{P}$ is incorrect

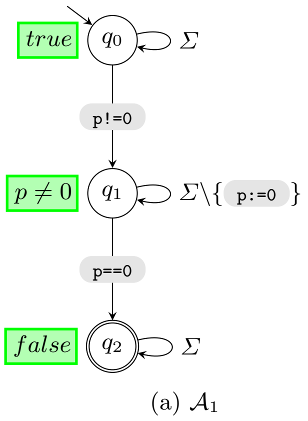
Fig. 2: [16] CEGAR-based scheme for non-incremental verification using trace abstraction.

The overline notation stands for computing automata complementation and the $\cap$ notation stands for computing automata intersection. Note that complementation, intersection and emptiness checking, can all be done efficiently for finite automata. Checking whether this equation holds is semantically equivalent to checking whether $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(A_1) \cup \ldots \cup \mathcal{L}(A_n)$, so if the answer is "yes", we can state that the program is correct (Theorem 2). If the answer is "no", then we get a witness in the form of a trace π s.t. $\pi \in \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$, which is passed on to the refinement phase.

During the refinement phase, π is semantically analyzed to decide whether it is infeasible or not. If it is not, we can state that the program is incorrect, since π is a feasible error trace of $\mathcal{P}$, i.e., an execution of $\mathcal{P}$ that leads to an error. If it is, then the proof of its infeasibility can be used to construct a Floyd-Hoare automaton $\mathcal{A}_{n+1}$ that accepts π (in particular, the way this is done in [17], is by obtaining a set of valid Hoare triples from the proof and applying α on it). This automaton is then added to the produced trace abstraction, and the process is repeated.

Example 2. Recall program P_{ex1} from Figure 1. We claim that an assertion violation is not possible in this program. A convincing argument for this claim can be made by considering separately those executions that visit ℓ_3 at least once and those who do not. For the later, `p` is never assigned during the execution, and the assume statement makes sure that initially `p` does not equal 0, so every time the assertion is reached the condition `p!=0` must hold. For the former, since ℓ_3 is reached, the true branch of the `if` statement was taken during that iteration, so `n` equals 0 at ℓ_4. Therefore, after the execution of `n--`, `n` will equal -1, and thus the loop will be exited and the assertion will not be reached.

Program P_{ex1} is successfully verified using the scheme of Figure 2. The trace abstraction obtained is the tuple $(\mathcal{A}_1, \mathcal{A}_2)$, presented in Figure 3. Observe that



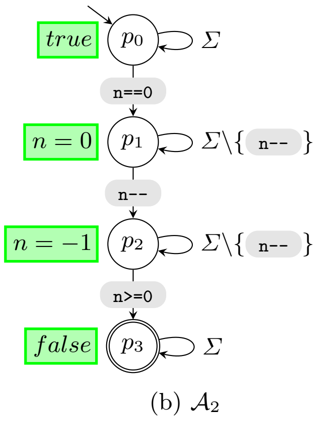


Fig. 3: Floyd-Hoare automata $\mathcal{A}_1$ and $\mathcal{A}_2$ with their respective accepting states q_2 and p_3. The gray frames labeling transitions represent letters from Σ, where an edge labeled with $G \subseteq \Sigma$ means a transition reading any letter from G. The green frames labeling states represent predicates assigned to states by the annotation θ.

the language of $\mathcal{A}_1$ consists of all traces that contain the statement `p!=0` followed by the statement `p==0`, without an assignment to `p` in between. The language of $\mathcal{A}_2$ consists of all traces that contain the statement `n==0` followed by the statement `n--` and the statement `n>=0`, without an assignment to `n` between any of these three statements. As we have just explained, all error traces of P_{ex1} fall into one of these categories (which one depends on whether or not ℓ_3 is visited), so the inclusion $\mathcal{L}(\mathcal{A}_{\mathcal{P}_{\text{ex1}}}) \subseteq \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ indeed holds.

4 Incremental Verification Using Trace Abstraction

In the previous section we saw a CEGAR-based algorithm for verification that constructed a new trace abstraction. In this section, we show how incremental verification can be done by reusing a given trace abstraction. For this incremental setting, in addition to the program $\mathcal{P}$, the algorithm also gets as input a trace abstraction TA^R, which we call the *reused trace abstraction.* We call the trace abstraction TA^C that is constructed by the algorithm the *current trace abstraction.* In addition to the verification result, the algorithm also returns TA^C which might be reused in subsequent verification tasks. The alphabet of the TA^R, which we call *reused alphabet* and denote by Σ^R, may be different from the alphabet of the program $\mathcal{P}$. We call the alphabet of the program *current alphabet* and denote it by Σ^C. While there is no restriction on the reused alphabet Σ^R, the performance of the algorithm is expected to improve the more similar it is to the current alphabet Σ^C (i.e., the larger the set $\Sigma^R \cap \Sigma^C$ is).

4.1 Translation of Floyd-Hoare Automata

The rationale for reusing a trace abstraction TA^R is that each Floyd-Hoare automaton in it forms a proof that the set of traces it accepts is infeasible (see Theorem 1), and therefore we do not need to analyze any trace in this set. The organization of the information in the form of an automaton, gives us a convenient way to get rid of all error traces of $\mathcal{P}$ that belong to this set: simply by subtracting the automaton from the program (which is also an automaton). Still, the above subtraction can not be done straight away, since the program $\mathcal{P}$ and the reused trace abstraction TA^R are not necessarily over the same alphabet.

Traces of the reused trace abstraction TA^R that contain statements that are not from the current alphabet Σ^C are definitely not error traces of our program and hence rather useless for us. Therefore, we would like to "translate" the reused trace abstraction from the reused Σ^R to the current alphabet Σ^C. We first define our notion of such a "translation" for valid Hoare triples and lift the translation to Floyd-Hoare automaton afterwards.

Definition 2 (Translation of a set of valid Hoare triples) *Given a set of valid Hoare triples $S_{\Sigma^R} \subseteq HT_{\Sigma^R}$ over the reused alphabet, we call a set of valid Hoare triples $S_{\Sigma^C} \subseteq HT_{\Sigma^C}$ over the current alphabet a* translation *of S_{Σ^R} to the current alphabet Σ^C, if all valid Hoare triples in S_{Σ^R} are also in S_{Σ^C}. In other words, $S_{\Sigma^C} \subseteq HT_{\Sigma^C}$ is a translation if the following inclusion holds.*

$$S_{\Sigma^R} \cap HT_{\Sigma^R \cap \Sigma^C} \subseteq S_{\Sigma^C}$$

In order to lift our notion of "translation" to Floyd-Hoare automata we use function β which was defined in the previous chapter and maps a Floyd-Hoare automata to a set of valid Hoare triples.

Definition 3 (Translation of a Floyd-Hoare automaton) *Given a Floyd-Hoare automaton A_{Σ^R} over the reused alphabet Σ^R, we call a Floyd-Hoare automaton A_{Σ^C} over the alphabet Σ^C a* translation *of A_{Σ^R} to Σ^C, if the set of valid Hoare triples $\beta(A_{\Sigma^C})$ is a translation of $\beta(A_{\Sigma^R})$ to Σ^C.*

Given a Floyd-Hoare automaton A_{Σ^R} over the reused alphabet Σ^R and a set S_{Σ^C} of valid Hoare triples over the current alphabet Σ^C, we use the procedure depicted in Fig. 4 to translate A_{Σ^R} to a Floyd-Hoare automaton A_{Σ^C} over the current alphabet Σ^C.

Proposition 1 *Every Floyd-Hoare automaton A_{Σ^C} that is constructed using the procedure* TranslateAutomaton*, is a translation of the reused Floyd-Hoare automaton A_{Σ^R} to the current alphabet Σ^C.*

Proof. Since all valid Hoare triples removed from S_1 when creating S_2 were over $\Sigma^R \setminus \Sigma^C$, then $S_1 \cap HT_{\Sigma^R \cap \Sigma^C} \subseteq S_2$. Therefore, by Definition 2, S_2 is a translation of S_1 to Σ^C. Now, $S_1 = \beta(A_{\Sigma^R})$, so we conclude that S_2 is a translation of $\beta(A_{\Sigma^R})$ to Σ^C. Next, we want to claim that $\beta(A_{\Sigma^C}) = S_2$. This

```
Input:  A Floyd-Hoare automaton $A_{\Sigma^R}$ over $\Sigma^R$ and
        a set of valid Hoare triples $S_{\Sigma^C} \subseteq HT_{\Sigma^C}$
Output: A Floyd-Hoare automaton $A_{\Sigma^C}$ over $\Sigma^C$

  1. Construct the set of valid Hoare triples $S_1 = \beta(A_{\Sigma^R})$.
  2. Construct the set of valid Hoare triples $S_2 = (S_1 \setminus HT_{\Sigma^R \setminus \Sigma^C}) \cup S_{\Sigma^C}$.
  3. Return the Floyd-Hoare automaton $A_{\Sigma^C} = \alpha(S_2)$.
```

Fig. 4: Procedure TRANSLATEAUTOMATON.

is correct because, according to the definitions of β and α, for every set S, $\beta(\alpha(S)) = S$, so in particular $\beta(\alpha(S_2)) = S_2$. Thus, we conclude that $\beta(A_{\Sigma^C})$ is a translation of $\beta(A_{\Sigma^R})$ to Σ^C. By Definition 3, this means that A_{Σ^C} is a translation of A_{Σ^R} to Σ^C.

The procedure TRANSLATEAUTOMATON enables us to translate the reused trace abstraction TA^R into the alphabet of the program, but the question that remains is the choice of S_{Σ^C}. I.e., the question how many and which valid Hoare triples we should add in addition to the valid Hoare triples that are obtained from TA^R. The set S_{Σ^C} can be any subset of HT_{Σ^C} and obviously the larger S_{Σ^C} is, the more error traces of $\mathcal{P}$ (and other programs that occur in subsequent verification tasks) are proven infeasible.

We note that we do not only have the costs for the construction S_{Σ^C} itself. If S_{Σ^C} is larger, the automaton A_{Σ^C} will have more transitions and the costs for automata operations (e.g., complementation and intersection) and translations in future verification tasks will be higher.

Thus, the choice of S_{Σ^C} is a trade-off between how much effort we are willing to spend on building the translated automata and using them, and how useful they will be for proving the new program correct.

In our implementation we considered the following three options for S_{Σ^C}:

$$\begin{aligned}
S^{\mathsf{empty}}_{\Sigma^C} &= \emptyset \\
S^{\mathsf{unseen}}_{\Sigma^C} &= \{\{\varphi_1\}s\{\varphi_2\} \mid s \in \Sigma^C \setminus \Sigma^R,\ \varphi_1, \varphi_2 \in \Phi_{A_{\Sigma^R}}\} \\
S^{\mathsf{all}}_{\Sigma^C} &= \{\{\varphi_1\}s\{\varphi_2\} \mid s \in \Sigma^C,\ \varphi_1, \varphi_2 \in \Phi_{A_{\Sigma^R}}\}
\end{aligned}$$

Note that all three sets are indeed subsets of HT_{Σ^C}, and all of them only use predicates from $\Phi_{A_{\Sigma^R}}$. As a result, the procedure TRANSLATEAUTOMATON with either of these sets as S_{Σ^C} yields an Floyd-Hoare automaton A_{Σ^C} whose states were also states of the input A_{Σ^R} (i.e., states are only removed and not added). Also, in all three cases, transitions with irrelevant letters (i.e., letters in $\Sigma^R \setminus \Sigma^C$) are removed, while transitions with relevant letters (i.e., letters in Σ^C) remain.

The difference between the three options for S_{Σ^C} lies in the transitions that are added to A_{Σ^R}. In the case of $S^{\mathsf{empty}}_{\Sigma^C}$, no transitions are added at all. In this case, translated automata are only useful to prove infeasibility of error traces that remained unchanged from the previous version of the program $\mathcal{P}$, but we do not have any costs for the construction of S_{Σ^C}. On the other end of the spectrum

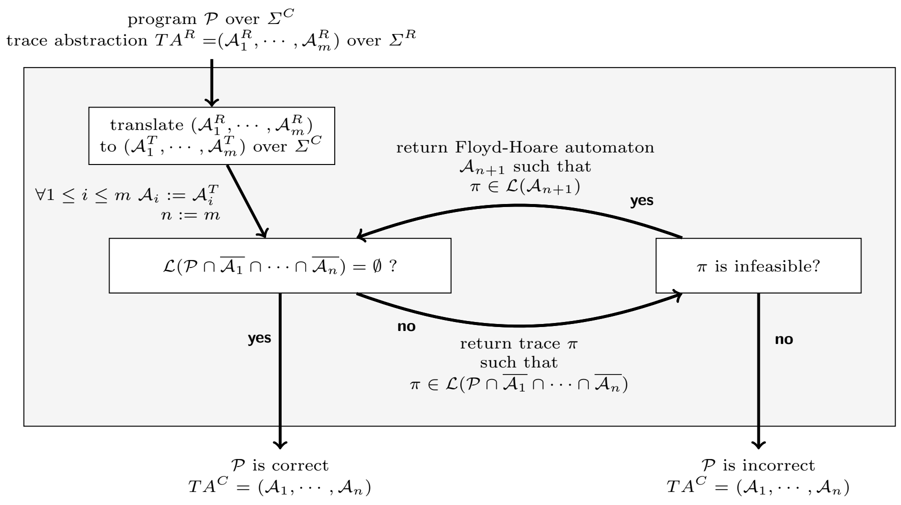

program $\mathcal{P}$ over Σ^C
trace abstraction $TA^R = (\mathcal{A}_1^R, \cdots, \mathcal{A}_m^R)$ over Σ^R
translate $(\mathcal{A}_1^R, \cdots, \mathcal{A}_m^R)$ to $(\mathcal{A}_1^T, \cdots, \mathcal{A}_m^T)$ over Σ^C
$\forall 1 \leq i \leq m\ \mathcal{A}_i := \mathcal{A}_i^T$
$n := m$
return Floyd-Hoare automaton $\mathcal{A}_{n+1}$ such that $\pi \in \mathcal{L}(\mathcal{A}_{n+1})$
yes
$\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n}) = \emptyset$?
π is infeasible?
yes
no
return trace π such that $\pi \in \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$
no
$\mathcal{P}$ is correct
$TA^C = (\mathcal{A}_1, \cdots, \mathcal{A}_n)$
$\mathcal{P}$ is incorrect
$TA^C = (\mathcal{A}_1, \cdots, \mathcal{A}_n)$

Fig. 5: Scheme for incremental verification using an **eager** approach.

there is $S^{\mathsf{all}}_{\Sigma^C}$, in which all valid Hoare triples over Σ^C are added as transitions to A_{Σ^C}. Here, any error trace that can be proved infeasible using predicates from $\Phi_{A_{\Sigma^R}}$ will be accepted by A_{Σ^C}. However, in this case the construction of S_{Σ^C} is expensive and the resulting automata are often rather large.

The option $S^{\mathsf{unseen}}_{\Sigma^C}$ suggests an intermediate solution, by considering only valid Hoare triples over the difference $\Sigma^C \setminus \Sigma^R$. The rationale is that most valid Hoare triples over the intersection $\Sigma^C \cap \Sigma^R$ that are relevant to prove infeasibility of error traces were already added when the reused Floyd-Hoare automaton A_{Σ^R} was constructed. In pracitice, there are only error traces whose infeasibility can be shown with option $S^{\mathsf{all}}_{\Sigma^C}$ but not with option $S^{\mathsf{unseen}}_{\Sigma^C}$ if statements in the program $\mathcal{P}$ have been reordered or existing statements were also added at other positions of the program.

We have performed experiments with all three of these options. The set that gave the best overall results on average, was $S^{\mathsf{all}}_{\Sigma^C}$ and hence we used as $S_{\Sigma^C} := S^{\mathsf{all}}_{\Sigma^C}$ in our experimental evaluation (see Section 5.1). The fact that $S^{\mathsf{all}}_{\Sigma^C}$ outperforms $S^{\mathsf{unseen}}_{\Sigma^C}$ suggests, perhaps, that changes such as reordering code and adding preexisting code (i.e., copy-pasting), on which $S^{\mathsf{unseen}}_{\Sigma^C}$ has bad results, are frequent in software evolution.

4.2 Reuse Algorithms

We now present two schemes for incremental verification, that differ in the strategy they use for subtraction of Floyd-Hoare automata from the program. In both schemes, any subtraction $\mathcal{P} - \mathcal{A}$ is replaced with $\mathcal{P} \cap \overline{\mathcal{A}}$, which results in the same language but uses different automata operations that more faithfully represent our implementation.

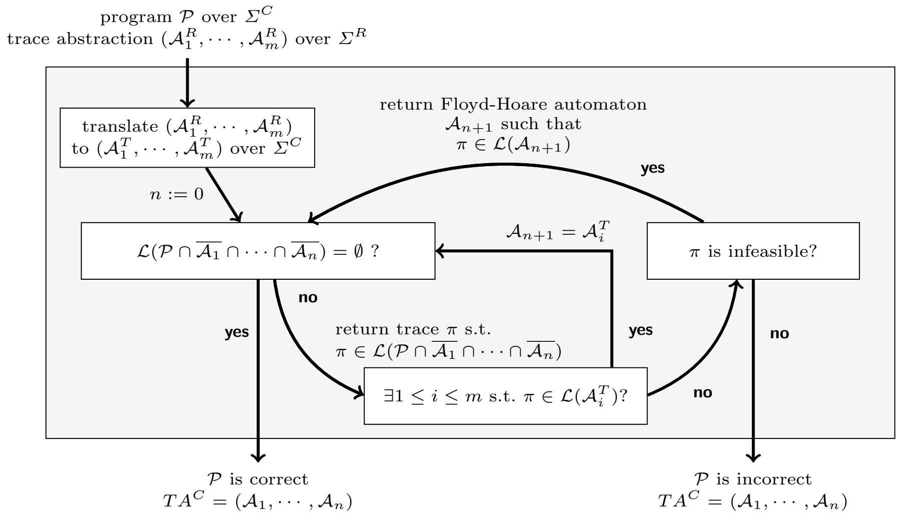

program $\mathcal{P}$ over Σ^C
trace abstraction $(\mathcal{A}_1^R, \cdots, \mathcal{A}_m^R)$ over Σ^R
translate $(\mathcal{A}_1^R, \cdots, \mathcal{A}_m^R)$ to $(\mathcal{A}_1^T, \cdots, \mathcal{A}_m^T)$ over Σ^C
$n := 0$
return Floyd-Hoare automaton $\mathcal{A}_{n+1}$ such that $\pi \in \mathcal{L}(\mathcal{A}_{n+1})$
yes
$\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n}) = \emptyset$?
$\mathcal{A}_{n+1} = \mathcal{A}_i^T$
π is infeasible?
no
yes
return trace π s.t. $\pi \in \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$
yes
no
$\exists 1 \leq i \leq m$ s.t. $\pi \in \mathcal{L}(\mathcal{A}_i^T)$?
no
$\mathcal{P}$ is correct $TA^C = (\mathcal{A}_1, \cdots, \mathcal{A}_n)$
$\mathcal{P}$ is incorrect $TA^C = (\mathcal{A}_1, \cdots, \mathcal{A}_n)$

Fig. 6: Scheme for incremental verification using a **lazy** approach.

Eager Reuse. The first scheme, presented in Figure 5, suggests an eager approach for the reuse of Floyd-Hoare automata. Here, subtraction of Floyd-Hoare automata is done straight away, and entirely (all Floyd-Hoare automata in the trace abstraction are subtracted). Then, the CEGAR-based algorithm continues as in the non-incremental case. The current trace abstraction, TA^C, contains all automata translated from the reused trace abstraction TA^R along with all other automata obtained during the CEGAR loop.

An advantage of this scheme is that all traces whose infeasibility is shown by a Floyd-Hoare automaton from TA^R are are excluded right at the beginning. On the other hand, we may have done some subtractions (or, in fact, intersections) that did not change the language at all and hence were not useful. For example, it is possible that for some automaton $\mathcal{A}_i^T$ translated from TA^R, $\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1^T} \cap \cdots \cap \overline{\mathcal{A}_i^T}) = \mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1^T} \cap \cdots \cap \overline{\mathcal{A}_{i-1}^T})$ and so the computation of the intersection with $\overline{\mathcal{A}_i^T}$ was done in vain. Note that all Floyd-Hoare automata are added to TA^C, regardless of whether they were useful or not, since retrieving this information is prohibitively expensive due to technical reasons.

Lazy Reuse. The second scheme, presented in Figure 6, suggests a lazy approach for the reuse of Floyd-Hoare automata. A Floyd-Hoare automaton is only subtracted once we know that it is useful, i.e., that its subtraction will remove at least one trace from the set of traces we have not yet proven infeasible.

In this scheme, the current trace abstraction is initially the empty sequence, as in the non-incremental case. Then the CEGAR loop begins, but with an additional phase, which we call the *reuse phase*, inserted between the validation and refinement phases (which themselves are not changed). If the validation

```
ℓ0: if(p==0) {
ℓ1:     n:=-2;
    }
ℓ2: while(n>=0) {
ℓ3:     assert p!=0;
        if(n==0) {
ℓ4:         p:=0;
        }
ℓ5:     n--;
    }
```

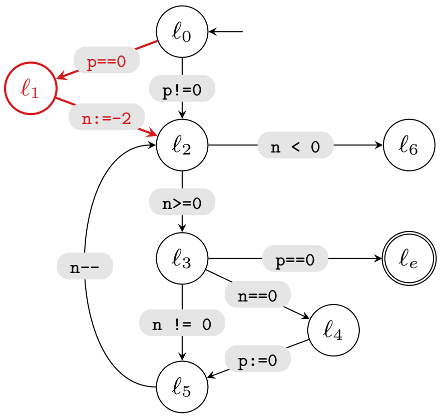


Fig. 7: Program P_{ex2}, which is a modified version of program P_{ex1}. Changes from P_{ex1} appear in red.

phase finds a trace π in $\mathcal{L}(\mathcal{P} \cap \overline{\mathcal{A}_1} \cap \cdots \cap \overline{\mathcal{A}_n})$, then the reuse phase first checks whether this trace is accepted by some automaton $\mathcal{A}_i^T$ which was translated from the reused TA^R. If it is, then $\mathcal{A}_i^T$ is added to the current trace abstraction and we return to the validation phase again. If it is not, then we pass π to the refinement phase and proceed as before. The current trace abstraction in this case includes only those automata translated from the reused TA^R that were added to it during the reuse phase, in addition to all those created during the refinement phase.

Example 3. Figure 7 presents the source code and the control-flow automaton of a program P_{ex2}. This program is an updated version of P_{ex1} (see Figure 1), where instead of assuming that `p` is initially different than 0, the variables `n` is set to -2 if `p` equals 0. The alphabet Σ^C of the control-flow automaton $\mathcal{A}_{\mathcal{P}_{\text{ex2}}}$ is the set of P_{ex2}'s statements (i.e., $\Sigma^C = \Sigma^R \cup \{$ `n:=-2` $\}$, where the reused alphabet Σ^R is the alphabet of $\mathcal{A}_{\mathcal{P}_{\text{ex1}}}$).

You will notice that despite the changes made, the assertion still can not be violated. For executions who visit ℓ_4 (formerly ℓ_3) at least once, we can make the same argument as we did in Example 2. For executions who do not visit ℓ_4, the argument we used in Example 2 relied on `p` being initially different than 0, so now it only applies to those executions beginning in a transition from ℓ_0 to ℓ_2. For executions going from ℓ_0 to ℓ_1, we need a new argument. For them, we can say that the visit in ℓ_1 guarantees `n` will be equal to -2 upon reaching ℓ_2, and thus the loop will not be entered and the assertion will not be reached.

Figure 8 presents the current trace abstraction $TA^C = (\mathcal{A}_1^C, \mathcal{A}_2^C, \mathcal{A}_3^C)$ produced by our algorithm, in both the Eager and the Lazy variants, when using the tuple $(\mathcal{A}_1, \mathcal{A}_2)$ from Figure 3 as the reused trace abstraction TA^R. The first two automata, $\mathcal{A}_1^C$ and $\mathcal{A}_2^C$, are the translations of automata $\mathcal{A}_1$ and $\mathcal{A}_2$ to the current alphabet Σ^C, resp. The translation of the trace abstraction, in this case, amounts to adding transitions with the new letter, `n:=-2`, where appropriate. Specifically, `n:=-2` was added to the 3 self-loops in $\mathcal{A}_1$, and to the self loops from p_0 and p_3 in $\mathcal{A}_2$. The third automaton, $\mathcal{A}_3^C$, is a new Floyd-Hoare automaton, obtained during the refinement phase.

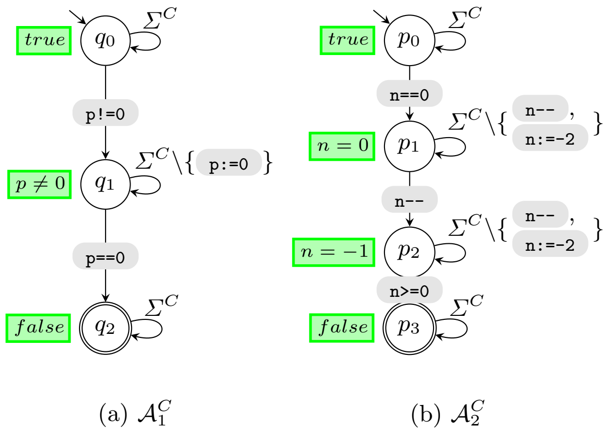


Fig. 8: Trace abstraction ($\mathcal{A}_1^C$,$\mathcal{A}_2^C$,$\mathcal{A}_3^C$), which is the output of our algorithm for P_{ex2}, when using the tuple $(\mathcal{A}_1, \mathcal{A}_2)$ from Figure 3 as TA^R.

5 Evaluation

We have implemented our incremental verification algorithms on top of the Ultimate Automizer software verification tool, which is part of the Ultimate program analysis framework [3]. The source code is available on Github [4]. We currently support incremental verification of C and Boogie programs with respect to safety properties (e.g., validity of assertions or memory-access safety).

On-the-fly Computation For simplicity of presentation, schemes of our algorithms in Figure 5 and Figure 6 show a stand-alone translation phase that precedes the CEGAR loop. According to these schemes, each automaton $\mathcal{A}_j^R$ in the reused trace abstraction is first translated into an automaton $\mathcal{A}_j^T$ over the current alphabet Σ^C. In practice, computing $\mathcal{A}_j^T$ entirely can be quite expensive, depending on the set of valid Hoare triples S_{Σ^C}, as previously discussed. Also, the computation of many transitions may turn out to be redundant, as we may not need these transitions at any point during the CEGAR loop. Therefore, our implementation translates automata on-the-fly, adding transitions only as

[3] https://ultimate.informatik.uni-freiburg.de
[4] https://github.com/ultimate-pa

soon as the need for them emerges. On-the-fly translation may happen during the reuse phase in the Lazy reuse algorithm, and during the validation phase in both algorithms. Additionally, creation of Floyd-Hoare automaton $\mathcal{A}_{n+1}$ in case a trace is found infeasible during the refinement phase is already done on-the-fly in the preexisting implementation of Ultimate Automizer. That is, transitions are added to $\mathcal{A}_{n+1}$ only if they are needed during the following validation phase.

5.1 Experimental Results

We have performed an extensive experimental evaluation of our approach on a set of benchmarks previously established in [4], available on-line[5]. This benchmark set is based on industrial source code from the Linux kernel, and contains 4,193 verification tasks from 1,119 revisions of 62 device drivers. A verification task is a combination of driver name, revision number, and specification, where the specification is one of six different rules for correct Linux kernel core API usage (more details can be found in [4]). We excluded those tasks where Ultimate Automizer was unable to parse the input program successfully, and were left with a total of 2,660 verification tasks.

Our experiments were made on a machine with a 4GHz CPU (Intel Core i7-6700K). We used Ultimate Automizer version 0.1.23-bb20188 with the default configuration, which was also used in SV-COMP'18 [6], [18]. In this configuration Ultimate Automizer first uses SMTInterpol[7] with Craig interpolation for the analysis of error traces during the refinement phase, and if this fails, falls back on Z3 [8] with trace interpolation [11]. Validity of Hoare triples is also checked with Z3. A timeout of 90s was set to all verification tasks and the Java heap size was limited to 6GB.

For each verification task we verified the revision against the specification three times: first, without any reuse, and then with reuse, using both the Eager and the Lazy algorithms. The output trace abstraction of each revision was used as the input trace abstraction of the next revision. The results of these experiments are summarized in Table 1.

These results clearly show that our method, both when used with the Eager algorithm and with the Lazy one, manages to save the user a considerable amount of time, for the vast majority of these benchmarks. The difference in performance between the Eager and Lazy algorithms on these benchmarks was quite negligible; both obtain a nontrivial speedup of around $\times 4.7$ in analysis time, and $\times 3.6$ in overall time, on average. When comparing mean analysis speedups of our approach and that of [4], we get a speedup that is $\times 1.5$ larger. But, what is additionally interesting to note, is that we do not succeed on the same benchmarks as [4] does; the best 15 series in our work and theirs are completely disjoint. This suggests that the two methods are orthogonal.

[5] `https://www.sosy-lab.org/research/cpa-reuse/regression-benchmarks`

[6] `https://sv-comp.sosy-lab.org/2018/`

[7] `https://ultimate.informatik.uni-freiburg.de/smtinterpol`, version 2.1-441-gf99e49f

[8] `https://github.com/Z3Prover/z3`, version master 450f3c9b

Table 1: The results of our evaluation. Each row contains the results for a series of revisions of a driver and one type of specification. The table only shows those series where we could parse all files, allowing for a comparison in speedup with [4]. We also limited the display to the best 15 and the worst 10 series in terms of speedup. The number of tasks specifies the number of files including the first revision. The settings "Eager" and "Lazy" are divided in overall and analysis time, where analysis time is the overall time without the time it took writing the output trace abstraction to file. As the "Default" setting does not write an output trace abstraction, its analysis time is the same as its overall time. All times are given as seconds of wall time and do not include the time for the first revision. The speedup colums compare the relative speedup between the Default setting and the Lazy setting. The rows "Sum" and "Mean" show the sum and mean of all the series where we were able to parse all the tasks, whereas the rows "Sum (All)" and "Mean (All)" show the sum and the mean of all the tasks we could parse. We adjusted the mean speedup of [4] for our subset by recomputing their speedup relative to our shared subset, but their mean speedup in the "Mean (All)" row refers to the original 4,193 tasks.

			Default	Eager		Lazy				
Driver	Spec	Tasks	Overall	Overall	Analysis	Overall	Analysis	Speedup Overall	Speedup Analysis	[4] Speedup
dvb-usb-rtl28xxu	08_1a	10	20.509	0.352	0.187	0.416	0.257	49.30	79.80	3.6
dvb-usb-rtl28xxu	39_7a	10	110.893	4.081	1.992	4.059	2.546	27.32	43.55	6.3
dvb-usb-rtl28xxu	32_7a	10	35.551	1.306	0.725	1.550	0.844	22.93	42.12	4.9
dvb-usb-az6007	08_1a	5	4.620	0.173	0.118	0.187	0.132	24.70	35.00	3.5
dvb-usb-az6007	39_7a	5	17.952	1.378	0.862	1.425	0.989	12.59	18.15	4.9
cx231xx-dvb	08_1a	13	3.330	0.303	0.206	0.323	0.228	10.30	14.60	1.8
panasonic-laptop	08_1a	16	3.466	0.337	0.222	0.384	0.257	9.02	13.48	2.4
spcp8x5	43_1a	13	5.531	0.632	0.437	0.618	0.432	8.94	12.80	1.6
panasonic-laptop	32_1	4	0.623	0.100	0.061	0.072	0.051	8.65	12.21	3.4
panasonic-laptop	39_7a	16	18.961	2.377	1.654	2.617	1.906	7.24	9.94	3.6
leds-bd2802	68_1	4	1.039	0.180	0.112	0.191	0.123	5.43	8.44	4.4
leds-bd2802	32_1	4	0.484	0.089	0.057	0.097	0.064	4.98	7.56	3.9
wm831x-dcdc	32_1	3	0.330	0.063	0.044	0.066	0.047	5.00	7.02	2.1
cx231xx-dvb	39_7a	13	17.536	3.389	2.425	3.464	2.517	5.06	6.96	3.2
ems_usb	08_1a	21	2.334	0.502	0.327	0.543	0.362	4.29	6.44	2.9
. . . (for full results cf. http://batg.cswp.cs.technion.ac.il/publications/)										
ar7part	32_7a	6	0.071	0.067	0.056	0.074	0.063	0.95	1.12	1.3
metro-usb	08_1a	25	0.394	0.497	0.330	0.518	0.356	0.76	1.10	2.1
rtc-max6902	32_7a	9	0.133	0.124	0.106	0.147	0.126	0.90	1.05	1.1
i2c-algo-pca	43_1a	7	0.012	0.018	0.018	0.019	0.019	1.00	1.00	1.0
dvb-usb-vp7045	43_1a	2	0.001	0.002	0.002	0.027	0.027	1.00	1.00	2.6
cfag12864b	43_1a	2	0.036	0.039	0.036	0.040	0.037	0.90	0.97	1.0
rtc-max6902	43_1a	5	0.278	0.273	0.262	0.303	0.291	0.91	0.95	1.1
magellan	32_7a	2	0.015	0.018	0.016	0.018	0.016	0.83	0.93	0.93
vsxxxaa	43_1a	2	0.030	0.037	0.033	0.036	0.032	0.83	0.93	6.8
ar7part	43_1a	2	0.036	0.043	0.038	0.044	0.039	0.81	0.92	1.2
Sum		1,177	529.258	142.856	107.543	146.275	112.225			
Mean		13	5.881	1.587	1.195	1.625	1.247	3.618	4.716	3.17
Sum (All)		2,660	3,048.373	434.853	334.603	448.424	349.69			
Mean (All)		15	16.749	2.389	1.838	2.464	1.921	6.798	8.717	4.3

Slowdowns are demonstrated for our worst 7 results. On the other hand, our top 7 results all demonstrate speedups of more than an order of magnitude, with an impressive max value of $\times 79.80$. For each pair of successive revisions, we have computed their edit-distance by summing up the number of added, modified and deleted lines, and dividing by the total number of lines in the file. To compute the edit-distance of a series, we have computed the mean edit-distance of all revisions in it. We expected to see a correlation between the edit-distance of a series and the speedup obtained for it. In general, such a correlation does seems to exist; a speedup of greater than 4 is achieved mostly for revisions where the edit distance is small. But, this correlation is not definitive. For example, we had one series where the mean edit-distance was over 90 percent, but the speedup was over $\times 60$. Also, cases with slowdowns distribute evenly over the mean edit-distance size.

6 Related Work

The validation of evolving software has been the subject of extensive research over the years (see the book by Chockler et al. [10]). Several different problems have been studied in this context, e.g., analyzing the semantic difference between successive revisions [26] or determining which revision is responsible for a bug [21,1]. In this section, we will focus on the problem of formally verifying all program revisions.

A dominant approach to solve this problem is to only verify the first revision, and then prove that every pair of successive revisions is equivalent. It was suggested by Godlin and Strichman in [24], where they gave it the name *regression verification* and introduced an algorithm that is based on the theory of uninterpreted functions. Papers about regression verification are concerned with improving equivalence checking and increasing its applicability. In [2], a summary of program behaviors impacted by the change is computed for both programs, and then equivalence is checked on summaries alone. Similarly, in [5], checking equivalence is done gradually by partitioning the common input space of programs and checking equivalence separately for each set in the partition. In [13], a reduction is made from equivalence checking to Horn constraint solving. In [25] applicability is extended to pairs of recursive functions that are not in lock-step, and in [7] to multi-threaded concurrent programs. The work of [3] is focused on Programmable Logic Controllers, which are computing devices that control production in many safety-critical systems. Finally, [27] proposes a different notion of equivalence, which on top of the usual functional equivalence also considers runtime equivalence.

Another approach towards efficiently verifying all program revisions, which is the one we follow in this paper, is to use during each revision verification partial results obtained from previous revisions, in order to limit necessary analysis. Work in this field vary based on the underlying non-incremental verification technique used, which determines what information can be reused and how efficiently so. The work we find most closely related to ours is that of Beyer et al [4], which suggests to reuse the abstraction precision in predicate abstraction.

Other techniques for reuse of verification results include reuse of function summaries for bounded model checking [7], contextual assumptions for assume-guarantee reasoning [15], parts of a proof or counter-example obtained through ic3 [9] and inductive invariants [12]. Also, incremental techniques for runtime verification of probabilistic systems modeled as Markov decision processes are developed in [14]. For the special case of component-based systems, [19] uses algebraic representations to minimize the number of individual components that need to be reverified. Last, the tool Green [28] facilitates reuse of SMT solver results for general purposes, and authors demonstrate how this could be beneficial for incremental program analysis.

7 Conclusion

We have presented a novel automata-based approach for incremental verification. Our approach relies on the method of [16,17] which uses a trace abstraction as a proof of correctness. Our idea is to reuse a trace abstraction by first translating it to the alphabet of the program under inspection, and then subtracting its automata from the control-flow automaton. We have defined a procedure, TranslateAutomaton, for automata translation, and two algorithms for reuse of trace abstraction that differ in their strategy for automata subtraction. We have evaluated our approach on a set of previously established benchmarks on which we get significant speedups, thus demonstrating the usefulness of trace abstraction reuse.

References

1. J. Adler, R. Berryhill, and A. Veneris. Revision debug with non-linear version history in regression verification. In *Verification and Security Workshop (IVSW), IEEE International*, pages 1–6. IEEE, 2016.
2. J. Backes, S. Person, N. Rungta, and O. Tkachuk. Regression verification using impact summaries. In *International SPIN Workshop on Model Checking of Software*, volume 7976 LNCS, pages 99–116. Springer, 2013.
3. B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl. Regression verification for programmable logic controller software. In *International Conference on Formal Engineering Methods*, pages 234–251. Springer, 2015.
4. D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 389–399, 2013.
5. M. Bohme, B. C. d. S. Oliveira, A. Roychoudhury, M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 302–311. IEEE Press, 2013.
6. B. Brandin, R. Malik, and P. Dietrich. Incremental system verification and synthesis of minimally restrictive behaviours. In *American Control Conference, 2000. Proceedings of the 2000*, volume 6, pages 4056–4061. IEEE, 2000.
7. S. Chaki, A. Gurfinkel, and O. Strichman. Regression Verification for Multi-threaded Programs. In *VMCAI*, volume 7148, pages 119–135. Springer, 2012.
8. K.-h. Chang, D. A. Papa, I. L. Markov, and V. Bertacco. InVerS: an incremental verification system with circuit similarity metrics and error visualization. In *Quality Electronic Design, 2007. ISQED'07. 8th International Symposium on*, pages 487–494. IEEE, 2007.
9. H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 135–143. FMCAD Inc, 2011.
10. H. Chockler, D. Kroening, L. Mariani, and N. Sharygina. *Validation of evolving software*. Springer, 2015.
11. D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski. Craig vs. newton in software model checking. In *ESEC/FSE 2017*, pages 487–497, 2017.
12. G. Fedyukovich, A. Gurfinkel, and N. Sharygina. Incremental Verification of Compiler Optimizations. In *NASA Formal Methods*, pages 300–306, 2014.
13. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating Regression Verification. *29th IEEE/ACM International Conference on Automated Software Engineering (ASE 2014)*, pages 349–360, 2014.
14. V. Forejt, M. Z. Kwiatkowska, D. Parker, H. Qu, and M. Ujma. Incremental Runtime Verification of Probabilistic Systems. In *RV*, pages 314–319. Springer, 2012.
15. F. He, S. Mao, and B.-Y. Y. Wang. Learning-based assume-guarantee regression verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9779:310–328, 2016.
16. M. Heizmann, J. Hoenicke, and A. Podelski. Refinement of trace abstraction. In *International Static Analysis Symposium*, pages 69–85. Springer, 2009.
17. M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *International Conference on Computer Aided Verification*, pages 36–52. Springer, 2013.

18. M. Heizmann, C. Yu-Fang, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski. Automizer and the Search for Perfect Interpolants. In *TACAS 2018*, 2018. To Appear.
19. K. Johnson, R. Calinescu, and S. Kikuchi. An incremental verification framework for component-based software systems. In *Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering*, pages 33–42. ACM, 2013.
20. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *TACAS*, volume 1, pages 98–112. Springer, 2001.
21. D. Maksimovic, A. Veneris, and Z. Poulos. Clustering-based revision debug in regression verification. *Proceedings of the 33rd IEEE International Conference on Computer Design, ICCD 2015*, pages 32–37, 2015.
22. P. Meseguer. Incremental verification of rule-based expert systems. In *Proceedings of the 10th European conference on Artificial intelligence*, pages 840–844. John Wiley & Sons, Inc., 1992.
23. O. Sery, G. Fedyukovich, and N. Sharygina. Incremental upgrade checking by means of interpolation-based function summaries. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 114–121, 2012.
24. O. Strichman and B. Godlin. Regression verification-a practical way to verify programs. *Verified Software: Theories, Tools, Experiments*, pages 496–501, 2008.
25. O. Strichman, M. Veitsman, O. S. B, and M. Veitsman. Regression Verification for unbalanced recursive functions. In *FM 2016: Formal Methods: 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings 21*, volume 9995, pages 645–658. Springer, 2016.
26. A. Trostanetski, O. Grumberg, and D. Kroening. Modular demand-driven analysis of semantic difference for program versions. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*, pages 405–427, 2017.
27. M. B. Venkatesh. A case study in Non-Functional Regression Verification. 2016.
28. W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 58:1–58:11, 2012.