Termination Analysis for Evolving Programs

An Incremental Approach by Reusing Certified Modules

FEI HE, Tsinghua University, China, Key Laboratory for Information System Security, MoE, China, and Beijing National Research Center for Information Science and Technology, China
JITAO HAN, Tsinghua University, China, Key Laboratory for Information System Security, MoE, China, and Beijing National Research Center for Information Science and Technology, China

Research on program termination has a long tradition. However, most of the existing techniques target a single program only. We propose in this paper an incremental termination analysis approach by reusing certified modules across different program versions. A transformation-based procedure is further developed to increase the reusability of certified modules. The proposed approach has wide applicability, applicable to various program changes. The proposed technique, to the best of our knowledge, represents a novel attempt to the termination analysis of evolving programs. We implemented the approach on top of Ultimate Automizer. Experimental results show dramatic improvement of our approach over the state-of-the-art tool.

CCS Concepts: • **Software and its engineering → Correctness**; **Software verification**.

Additional Key Words and Phrases: Termination analysis, incremental analysis, Büchi automaton, ranking function

ACM Reference Format:
Fei He and Jitao Han. 2020. Termination Analysis for Evolving Programs: An Incremental Approach by Reusing Certified Modules. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 199 (November 2020), 27 pages. https://doi.org/10.1145/3428267

1 INTRODUCTION

Termination is a fundamental program property. There is a wide variety of work on proving program termination from different angles [Brockschmidt et al. 2013; Cook et al. 2006a,b, 2013; Gulwani et al. 2008; Harris et al. 2010; Heizmann et al. 2014; Kroening et al. 2008, 2010; Larraz et al. 2013; Podelski and Rybalchenko 2004; Podelski and Rybalchenko 2005, 2011; Ströder et al. 2017]. However, most of these techniques target a single program version only. Note that programs evolve throughout their life cycles. *How to efficiently prove termination of evolving programs* is the main problem addressed in this paper.

Incremental analysis is a methodology that attempts to reuse the intermediate results, computed in the previous round of analysis, in the current analysis. The incremental analysis has been successfully applied to model checking [Beyer et al. 2012; Conway et al. 2005; He et al. 2016; Henzinger et al. 2003; Lauterburg et al. 2008; Yang et al. 2009], program verification [Beyer et al. 2013; Fedyukovich et al. 2013; Rothenberg et al. 2018; Sery et al. 2012; Yang et al. 2014], etc. However,

Authors' addresses: Fei He, School of Software, Tsinghua University, China, Key Laboratory for Information System Security, MoE, China, Beijing National Research Center for Information Science and Technology, Beijing, China, hefei@tsinghua.edu.cn; Jitao Han, School of Software, Tsinghua University, China, Key Laboratory for Information System Security, MoE, China, Beijing National Research Center for Information Science and Technology, Beijing, China, hanjt18@mails.tsinghua.edu.cn.

2475-1421/2020/11-ART199
https://doi.org/10.1145/3428267

to the best of our knowledge, we are not aware of any incremental technique for termination analysis of evolving programs.

Certified module, proposed by Heizmann, Hoenicke and Podelski [Heizmann et al. 2014], is a special case of Büchi automaton. A certified module guarantees that: 1) all accepted ω-traces are terminating, and 2) they share the same termination argument. To prove the termination of a program, it suffices to find a set of certified modules such that all ω-traces of the program are covered by these certified modules. The Ultimate Automizer tool that implements this approach won the 1st place in the Termination category of SV-COMP from 2017 – 2019.

We found that certified modules are *reusable*. Construction of a certified module is not guided by the program structure (see Section 2). Instead, a certified module may recognize traces that do not belong to the program, and can also reject traces of the program (i.e., left to other certified modules to cover). This freedom in representing the ω-language gives them more possibilities to be reused across different program versions. Recall that a certified module represents a set of *terminating* ω-traces. These traces can be subtracted from the program without proving their termination again. The whole verification efforts can thus be considerably reduced.

We propose an *incremental termination analysis* approach by reusing certified modules. Each certified module is decomposed into a number of *Hoare triples*. A Hoare triple $\{\varphi\}\ b\ \{\psi\}$ is reusable as long as the block b does not change. The certified module can guarantee that all decomposed Hoare triples are *valid*. So these triples can be reused without proving their validity again. Moreover, the precondition φ and the postcondition ψ convey information about the *ranking function*. This information is also reusable when we consider to add new Hoare triples. In this way, we realize as much as possible reusing of certified modules.

Regarding the program as an automaton over the alphabet of blocks (or statements), the program changes can be categorized into two types: the block changes, that add or delete blocks to the program, or the control-flow changes, that add or delete control-flow edges to the program. We show that our approach is *applicable* to both types of changes. As long as the two program versions share some blocks (or statements), our incremental approach is beneficial.

We implemented our approach on top of Ultimate Automizer. We evaluated our approach on two sets of programs, where the first contains 90 manually-crafted program revisions, and the second contains 611 real-world program revisions. Compared to the state-of-the-art Ultimate Automizer, our approach is x4.16 faster for manually-crafted revisions, and x3.08 faster for real-world revisions.

The main technical contributions of this paper are summarized as follows:

- We proposed, to the best of our knowledge, the first approach for termination analysis of evolving programs.
- We developed a transformation-based procedure for increasing the reusability of certified modules. We showed that our approach are applicable to various program changes.
- We implemented the proposed approach. Experimental results showed dramatic improvement of our approach.

The remainder of this paper is organized as follows. Section 2 introduces the necessary preliminaries. Section 3 motivates our approach using a simple example. Section 2.5 reviews the existing termination analysis approach for a single program version. Section 4 presents our incremental termination analysis approach for evolving programs. Section 5 discusses the applicability of our approach. Section 6 reports evaluation results on our approach. Section 7 discusses related work and Section 8 concludes this paper.

2 PRELIMINARIES

2.1 Program, Alphabet and Trace

Let P be a program. A *basic block* (or shortly, *block*) b of P is a straight-line sequence of program statements in P without any program jumps. Let Σ be the set of blocks in P. A program is a *control-flow automaton* $P = (Loc, \delta, l_\circ)$, where Loc is the set of program locations, $\delta \subseteq Loc \times \Sigma \times Loc$ is the set of control-flow edges labeled with blocks, and $l_\circ$ is the initial location.

We may treat Σ as the alphabet and each $b \in \Sigma$ as a letter [Heizmann et al. 2013a]. Then a finite *trace* over Σ is a *finite* sequence of letters in Σ, and an ω-*trace* over Σ is an *infinite* sequence of letters in Σ. Note that Σ is a finite set, the ω-trace σ over Σ is always in the *lasso shape*, i.e., $\sigma = b_0 \dots b_{k-1} (b_k \dots b_n)^\omega$, where the prefix $b_0 \dots b_{k-1}$ is called the *stem* of the lasso, and the periodic part $b_k \dots b_n$ is called the *loop* of the lasso. Denote Σ^* the set of all traces over Σ, and Σ^ω the set of all ω-traces over Σ.

A program is a special case of a Büchi automaton $(\Sigma, Loc, \delta, l_\circ, Loc)$, where the alphabet is the set of program blocks, the set of nodes is the set of the program locations, and the set of accepting nodes contains all program locations. A *path* π of P is an alternating sequence of locations and blocks, i.e.,

$$\pi = l_\circ \xrightarrow{b_0} l_1 \xrightarrow{b_1} \dots,$$

such that $(l_i, b_i, l_{i+1}) \in \delta$ for each $i \geq 0$. A trace (or an ω-trace) over Σ is called a trace (or an ω-trace) of P if it labels a path of P. Denote $\mathcal{L}(P)$ (or $\mathcal{L}^\omega(P)$) the set of all traces (or ω-traces) recognized by P. Obviously, $\mathcal{L}(P) \subseteq \Sigma^*$ and $\mathcal{L}^\omega(P) \subseteq \Sigma^\omega$.

2.2 Infeasible Trace

In the following, we use *Hoare triple* to define the feasibility of traces.

Let φ and ψ be two first-order predicates, and b be a program block. The Hoare-triple $\{\phi\}\ b\ \{\psi\}$ is *valid* iff $\phi \Rightarrow wp(b, \psi)$, where wp stands for the *weakest precondition transformer* [Bradley and Manna 2007]. Moreover, let $b_0 b_1 \dots b_n$ be a *finite* sequence of blocks, the Hoare triple $\{\phi\}\ b_0 b_1 \dots b_n\ \{\psi\}$ is *valid* iff $\phi \Rightarrow wp(b_0, wp(b_1, \dots, wp(b_n, \psi)))$.

Let V be the set of variables in the program. A *state* of P is a pair (l, s), where l is a program location, and s is a valuation to V. Denote ϕ_s the formula represention of s. For instance, the valuation $\{x_0 \mapsto 0, x_1 \mapsto 1\}$ can also be represented as $x_0 = 0 \wedge x_1 = 1$. Let (l, s) and (l', s') be two states of the program, we say (l', s') is reachable from (l, s), iff there is a trace σ such that $l \xrightarrow{\sigma} l'$, and the Hoare triple $\{\phi_s\}\ \sigma\ \{\phi_{s'}\}$ is valid.

A finite trace τ of P is *infeasible* if $\{true\}\ \tau\ \{false\}$ is valid, i.e., there exists no possible execution of P for τ. An ω-trace τ is *infeasible* if either: 1) any of its finite prefix is infeasible, or 2) there exists no initial valuation to enable any infinite execution of τ.

2.3 Module

Termination analysis needs only to consider ω-traces of the program. An ω-trace is terminating if and only if it is infeasible. In the following, we introduce *module* [Heizmann et al. 2014] as a representation of a set of ω-traces.

Definition 1. *A* module *is a 5-tuple $M = (\Sigma, Q, \delta, q_\circ, q_\bullet)$, where Σ is the alphabet, Q is a finite set of nodes, $\delta \subseteq Q \times \Sigma \times Q$ is a set of edges labeled with letters in Σ, $q_\circ \in Q$ is the initial node, $q_\bullet \in Q$ is the accepting node, and Q can be partitioned into two sets Q^0 and Q^1, such that $q_\circ \in Q^0$, $q_\bullet \in Q^1$ and no node in Q^1 has a successor in Q^0.*

Note that a node in a module is not necessarily a program location, and an edge of a module may not correspond to any control-flow edge of the program.

A module is a special case of a Büchi automaton where the set of states is the set of nodes and the set of accepting states contains $q_\bullet$ only. An *infinite* path of M is *fair* (also said, *accepted* by M) if it visits the accepting node $q_\bullet$ infinitely often. An ω-*trace* of M is *fair* (also said, *accepted* by M) if it labels a fair path of M. Denote $\mathcal{L}^\omega(M)$ the set of all fair ω-traces accepted by M. A module M is *terminating* if all traces in $\mathcal{L}^\omega(M)$ are terminating.

2.4 Ranking Function and Certified Module

Let f be a function over the program variables V that returns an element in a well-ordered set $(S, \prec)$. The function f is called a *ranking function* for M [Heizmann et al. 2014] if the values of $f(V)$ decrease every time the accepting node is visited. Note that this definition is a little different from the ranking function of a program, where the values of $f(V)$ are required to decrease on every transition of the program.

In the following, we use an auxiliary variable $oldrnk$ to refer to the value of $f(V)$ at the previous visit of the accepting node $q_\bullet$, and ∞ to represent a big value that is greater than all other values from the well-ordered S. The value of $oldrnk$ needs to be updated every time $q_\bullet$ is visited. To certify that $f(V)$ is indeed a ranking function, we need to guarantee $f(V) < oldrnk$ and $oldrnk \geq 0$ at each visit of $q_\bullet$.

Definition 2. *Given a module $M = (\Sigma, Q, \delta, q_\circ, q_\bullet)$ and a function $f(V)$ that returns an element in a well-ordered set $(S, \prec)$, a* rank certificate *$\mathcal{I}$ for f and M is an annotation that maps each node of M to a predicate, such that*

- *the initial node $q_\circ$ is mapped to a predicate where $oldrnk$ has the value ∞, i.e.,*

$$\mathcal{I}(q_\circ) \Leftrightarrow oldrnk = \infty;$$

- *the accepting node $q_\bullet$ is mapped to a predicate stating that the value of $f(V)$ decreases since the last visit of $q_\bullet$, i.e.,*

$$\mathcal{I}(q_\bullet) \Rightarrow (f(V) < oldrnk \wedge oldrnk \geq 0);$$

- *for any edge $(q, b, q') \in \delta$ where $q \neq q_\bullet$,*

$$\{\mathcal{I}(q)\}\ b\ \{\mathcal{I}(q')\} \text{ is valid; and}$$

- *for any edge $(q_\bullet, b, q') \in \delta$,*

$$\{\mathcal{I}(q_\bullet)\}\ \texttt{oldrnk:=f(V);}\ b\ \{\mathcal{I}(q')\} \text{ is valid.}$$

The triple $(M, f, \mathcal{I})$ is called a certified module.

Throughout the paper, we often write a certified module as M when f and $\mathcal{I}$ are clear from the context.

Lemma 1. *[Heizmann et al. 2014] A certified module is always terminating.*

A certified module can be used as a representation of a set of fair ω-traces that share the same reason (i.e., the ranking function and the rank certificate) for termination.

2.5 Termination Analysis Using Certified Modules

Theorem 1. *[Heizmann et al. 2014] A program P is terminating if there exists a set of terminating modules $M_0, M_1, \cdots, M_n$, such that*

$$\mathcal{L}^\omega(P) \subseteq \mathcal{L}^\omega(M_0) \cup \mathcal{L}^\omega(M_1) \cup \cdots \cup \mathcal{L}^\omega(M_n).$$

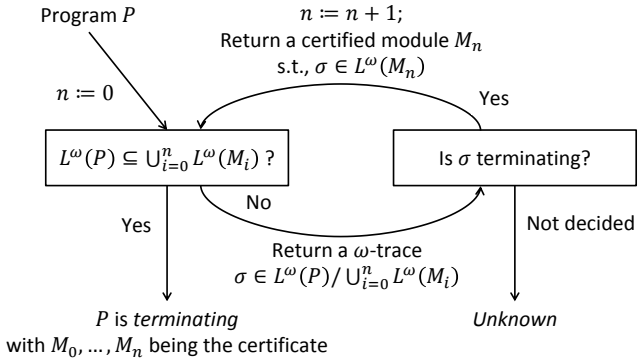


Fig. 1. Termination analysis using certified modules

```
1    int x, y;
2    x = nondet_int();
3    while (x >= 0){
4        y = 1;
5        while (y < x) {
6            y = 2*y;     // y = y + 1;
7        }
8        x = x - 1;
9    }
```

Fig. 2. A program example $loop_0$

Heizmann et al. [2014] proposed a decomposition-based approach for proving termination of a single program version. Figure 1 shows an overview of this approach. The basic idea is to iteratively construct a set of certified modules $M_0, M_1, \ldots, M_n$, such that $\mathcal{L}^\omega(P) \subseteq \bigcup_{i=0}^{n} \mathcal{L}^\omega(M_i)$.

Each iteration of this approach consists of two phases: the *language inclusion checking* and the *certified module construction*. During the language inclusion checking, we check if $\mathcal{L}^\omega(P) \subseteq \bigcup_{i=0}^{n} \mathcal{L}^\omega(M_i)$ or not. If the check succeeds, by Theorem 1, we immediately conclude that P is terminating. Otherwise, the program termination can not be proved with the current set of certified modules, and the checker returns a counterexample $\sigma \in \mathcal{L}^\omega(P) \setminus \bigcup_{i=0}^{n} \mathcal{L}^\omega(M_i)$.

During the certified module construction phase, the ω-trace σ returned by the former phase is semantically analyzed to decide if it is terminating or not. Termination analysis for a single ω-trace (or equivalently, for a lasso program that contains a single ω-trace) has long been studied. Many specialized methods was developed for lasso programs [Ben-Amram and Genaim 2013, 2014, 2015, 2017; Bradley et al. 2005a; Cook et al. 2010; Heizmann et al. 2013b; Kroening et al. 2008; Podelski and Rybalchenko 2004]. Even so, termination of this kind of simple programs is undecidable [Ben-Amram and Genaim 2014]. If the termination of σ cannot be decided, we report "unknown" [1]. Otherwise, with the termination argument (usually, a ranking function) generated by the existing methods, we proceed to construct a certified module.

3 A MOTIVATING EXAMPLE

Figure 2 shows a simple program $loop_0$ that contains a double nested `while` loop. Blocks in this program are: b_0 = `x=*` , b_1 = `assume x>=0; y=1` , b_2 = `assume y<x; y=2*y` , and b_3 = `assume y>=x; x=x-1` .

3.1 Termination Analysis of $loop_0$

To prove the termination of $loop_0$, we need to show that all ω-traces of $loop_0$ are terminating.

Let us first take an ω-trace of $loop_0$:

$$b_0 b_1 (b_2)^\omega$$

that enters the outer `while` loop and then visits the inner `while` loop infinitely often. Termination of this trace can be certified by the module M_0 in Figure 3a together with the linear ranking function $f(x, y) = x - y$ and the following rank certificate:

$$\mathcal{I}(q_1) \Leftrightarrow oldrnk = \infty, \text{ and}$$
$$\mathcal{I}(q_2) \Leftrightarrow (oldrnk > x - y \wedge oldrnk \geq 0 \wedge y > 0).$$

Note that the module M_0 accepts not only the input trace but also all traces that eventually always take the inner loop, i.e., $(\Sigma_0)^* b_1 (b_2)^\omega$, where $\Sigma_0 = \{b_0, b_1, b_2, b_3\}$. The rank certificate for M_0 further guarantees that all ω-traces accepted by M_0 are terminating.

Then, we check if $\mathcal{L}^\omega(loop_0) \subseteq \mathcal{L}^\omega(M_0)$ or not. This check returns a counterexample. Let us take the following counterexample trace:

$$b_0 (b_1 b_3)^\omega$$

that visits the outer `while` loop infinitely often. The termination of this trace can be certified by the module M_1 in Figure 3b together with the ranking function $f(x) = 2x + 1$ and the following rank certificate:

$$\mathcal{I}(q_0) \Leftrightarrow oldrnk = \infty,$$
$$\mathcal{I}(q_1) \Leftrightarrow (oldrnk > 2x + 1 \wedge oldrnk \geq 0), \text{ and}$$
$$\mathcal{I}(q_2) \Leftrightarrow (oldrnk \geq 2x + 1 \wedge x \geq 0).$$

Again, all ω-traces accepted by M_1 are terminating.

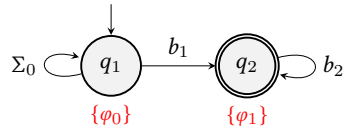
Finally, we check whether $\mathcal{L}^\omega(loop_0) \subseteq \mathcal{L}^\omega(M_0) \cup \mathcal{L}^\omega(M_1)$. We see that this check succeeds, i.e., all ω-traces of $loop_0$ are covered by M_0 and M_1. We thus conclude that $loop_0$ is terminating (by Theorem 1).

3.2 Termination Analysis of $loop_1$

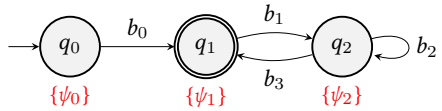
Now, let us assume that the program $loop_0$ evolves into a new version $loop_1$, where the statement `y=2*y` (at line 6) is replaced by the statement `y=y+1` . In terms of alphabet, $loop_1$ differs $loop_0$ only in block b_2. Denote b_2' = `assume y<x; y=y+1` as the altered version of b_2. Note that this change has side effects to the ranking function $f(x, y) = x - y$ and also the inner loop condition.

Observe that the certified modules M_0 and M_1 are intermediate results generated in the previous analysis, conveying important information about the termination of $loop_0$. Note that these two modules are defined over the alphabet Σ_0, but not Σ_1. They cannot directly be applied to the termination analysis of $loop_1$. However, if we look deep into the nodes and the edges of these two modules, reusable information can be identified.

[1]Before that, we will try to apply the non-termination analysis [Leike and Heizmann 2018] to get a conclusive "non-terminating" result.



(a) The certified module M_0, where $\varphi_0 \Leftrightarrow oldrnk = \infty$ and $\varphi_1 \Leftrightarrow (oldrnk > x - y \wedge oldrnk \geq 0 \wedge y > 0)$



(b) The certified module M_1, where $\psi_0 \Leftrightarrow oldrnk = \infty$, $\psi_1 \Leftrightarrow (oldrnk > 2x + 1 \wedge oldrnk \geq 0)$ and $\psi_2 \Leftrightarrow (oldrnk \geq 2x + 1 \wedge x \geq 0)$

Fig. 3. Certified modules for $loop_0$

Let us consider the edge (q_1, b_1, q_2) of M_0. According to the definition of a certified module (Definition 2), the Hoare triple $\{\mathcal{I}(q_1)\}\ b_1\ \{\mathcal{I}(q_2)\}$ is valid, where

$$\mathcal{I}(q_1) \Leftrightarrow oldrnk = \infty$$
$$\mathcal{I}(q_2) \Leftrightarrow (oldrnk > x - y \wedge oldrnk \geq 0 \wedge y > 0)$$

are two predicates mapped by the rank certificate $\mathcal{I}$ to q_1 and q_2, respectively.

The above Hoare triple is reusable, because b_1 is not changed in $loop_1$. With this valid Hoare triple, we know that if b_1 is executed from a state satisfying $\mathcal{I}(q_1)$, its final state must satisfy $\mathcal{I}(q_2)$. The validity of this Hoare triple just tells a fact about the semantics of b_1, which always hold as long as b_1 does not change. By reusing this Hoare triple, we need not to check its validity again. Given that the validity checking of Hoare triples needs to invoke an SMT solver, and in many cases is quite time-consuming. Reusing this information significantly reduces redundant computation.

Moreover, let us look at the postcondition $\mathcal{I}(q_2)$ of this Hoare triple. The ranking function $f(x, y) = x - y$ is embraced in this predicate. In other words, reusing this Hoare tripe can save us the efforts for re-computing the ranking function (if this ranking function still takes effect). Recall that the synthesis of ranking function is among the most time-consuming operations for termination analysis, reusing this information can save a great deal of effort.

4 INCREMENTAL TERMINATION ANALYSIS

This section proposes our incremental termination analysis approach. The basic idea is to reuse the previously-computed information in the current analysis.

In the sequel of this section, we first present the framework of our approach, then explain our *Reuse* technique, and our *on-demand* construction technique, and finally prove the correctness of our approach.

4.1 The Whole Procedure

Let P^- and P be two versions of a program, and Σ^- and Σ be their alphabets, respectively. Assume the termination of P^- has already been analyzed, generating a set of certified modules, say $M_0^-, M_1^-, \ldots, M_m^-$. Figure 4 shows an overview of our incremental termination analysis using certified modules. Before the new round of analysis starts, an initialization procedure is performed, which attempts to reuse the information stored in the previously-constructed certified modules.

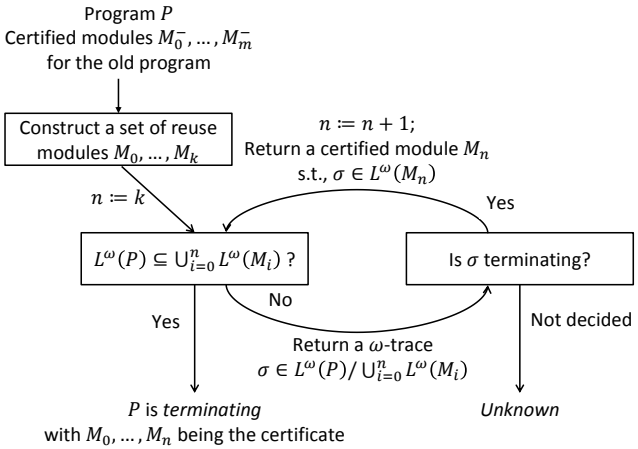


Fig. 4. Incremental termination analysis using certified modules

The *reuse* procedure is applied to each of the input certified modules, and outputs a reuse module. A reuse module is a certified module over Σ, and can be used for proving termination of P. Due to the diversity of program changes, the constructed reuse module may recognize an empty ω-language. For this case, this module covers no ω-traces of P, and is directly abandoned. Therefore, the number k of constructed modules is less than or equal to the number m of the input certified modules, i.e., $k \leq m$.

In each iteration, the language inclusion checking

$$\mathcal{L}^\omega(P) \subseteq \bigcup_{i=0}^{n} \mathcal{L}^\omega(M_i)$$

is equivalent to check if $P \cap \overline{M_0} \cap \cdots \cap \overline{M_n}$ recognizes an empty language or not. The latter check can be implemented in an incremental way, i.e., $P^{(0)} = P$, $P^{(1)} = P^{(0)} \cap \overline{M_0}, \ldots, P^{(n+1)} = P^{(n)} \cap \overline{M_n}$. Each automaton $P^{(l)}$ $(0 \leq l \leq n+1)$ is called a *remainder automaton*. We iteratively check if $P^{(l)}$ recognizes an empty ω-language or not.

4.2 The *Reuse* Procedure

Given a certified module M^- of the previous program version, we construct a new module M, called the *reuse module*, in the following steps.

Step 1. Extract predicates and Hoare triples. Recall that the previously-constructed certified module M^- and the current program version P are over different alphabets. The M^- may not be entirely reusable for the current analysis. However, we may decompose the certified module into a set of Hoare triples, many of which are reusable for the current analysis.

According to the certified module definition (Definition 2), on each edge of M^-, there is a valid Hoare triple. Particularly, for any edge (q, b, q') of M^-, where $q \neq q_\bullet^-$, we get a Hoare triple

$$\{\mathcal{I}(q)\}\ b\ \{\mathcal{I}(q')\};$$

and for any edge $(q_\bullet^-, b, q')$, we get a Hoare triple

$$\{\mathcal{I}(q_\bullet^-)\}\ \texttt{oldrnk:=f(V);}\ b\ \{\mathcal{I}(q')\}.$$

Denote $\mathcal{H}_1$ as the set of Hoare triples obtained in this step.

We also collect the set of predicates used in the rank certificate of M^-, i.e.

$$\mathcal{U}_1 = \bigcup_{\forall q \in Q^-} \{\mathcal{I}(q)\}.$$

Especially, we use $u_\circ = \mathcal{I}(q^-_\circ)$ and $u_\bullet = \mathcal{I}(q^-_\bullet)$ to mark the predicates that are mapped to the initial and accepting nodes of M^-, respectively. Note that $\mathcal{U}_1$ can also be considered as the set of predicates (precondition or postcondition) of all Hoare triples in $\mathcal{H}_1$.

Example 1. *Consider the certified module M_0 in Figure 3a, the predicates mapped to q_1 and q_2 by the rank certificate are respectively*

$$\varphi_0 \Leftrightarrow oldrnk = \infty, \text{ and}$$
$$\varphi_1 \Leftrightarrow (oldrnk > x - y \wedge oldrnk \geq 0 \wedge y > 0).$$

The set of predicates of M_0 is thus $\{\varphi_0, \varphi_1\}$. The Hoare triple for edges (q_1, b_1, q_2) and (q_2, b_2, q_2) are respectively

$$\{\varphi_0\}\, b_1\, \{\varphi_1\}, \text{ and}$$
$$\{\varphi_1\}\ \texttt{oldrnk:=f(V);}\ b_2\, \{\varphi_1\}.$$

The set of all Hoare triples of M_0 are listed in Figure 5a.

Step 2. Delete inapplicable predicates and Hoare triples. The extracted predicates and Hoare triples record information about the previous program version. They may not all be applicable to the current analysis.

Let V be the set of program variables in P. For any predicate $\varphi \in \mathcal{U}$, define $FV(\varphi)$ as its set of free variables. Basically, φ can only use program variables in V and a special variable $oldrnk$ as its free variables. The predicate φ is *inapplicable* to P if

$$FV(\varphi) \setminus (V \cup \{oldrnk\}) \neq \emptyset.$$

A Hoare triple $\{\varphi\}\, b\, \{\varphi'\}$ is *inapplicable* to P if either (1) φ or φ' is inapplicable to P, or (2) b does not exist in P.

The inapplicable predicates and Hoare triples need be removed from $\mathcal{U}$ and H, respectively. Note that $u_\circ$ and $u_\bullet$ are necessary for constructing the reuse module. If $u_\bullet$ is deleted from $\mathcal{U}$ in this step, we stop the *reuse* procedure for M^-, and proceed to process the next certified module. Note that $u_\circ$ will never be deleted, since it always represents $oldrnk = \infty$, containing a single free variable $oldrnk$. Denote the resulting sets of predicates and Hoare triples after elimination as $\mathcal{U}_2$, $\mathcal{H}_2$, respectively.

Example 2. *Consider the predicate set $\mathcal{U}$ obtained in Example 1, except of $oldrnk$, other two variables x and y are declared in $loop_1$, thus no predicate needs to be eliminated from $\mathcal{U}$. Consider the Hoare triple set $\mathcal{H}_1$ obtained in Example 1 (in Figure 5a). Note that the only changed block in $loop_1$ against $loop_0$ is b_2. Therefore, the third and sixth Hoare triples in $\mathcal{H}_1$ are inapplicable to $loop_1$ and need to be removed from this set.*

Step 3. Add new Hoare triples. The new program version may contain new blocks, which should be tested for addition of new Hoare triples.

Let $b \in \Sigma \setminus \Sigma^-$ be a new block. In this step, we limit the precondition and postcondition of the candidate Hoare triple to among the predicates in $\mathcal{U}_2$. For any two predicates $\varphi, \varphi' \in \mathcal{U}_2$, if $\varphi \neq u_\bullet$, we test the validity of $\{\varphi\}\, b\, \{\varphi'\}$; if $\varphi = u_\bullet$, we test the validity of $\{\varphi\}$ `oldrnk:=f(v);` $b\, \{\varphi'\}$; we then add the passed Hoare triples to $\mathcal{U}_2$. In this way, the previously-computed rank certificate are adapted to the new program version. Let $\mathcal{H}_3$ be the triple set after addition of new Hoare triples.

Example 3. *Consider the Hoare triple set $\mathcal{H}_2$ obtained in Example 2. Note that b_2' is the only new block in $loop_1$ compared to $loop_0$. After validity testing, two new Hoare triples (colored red in Figure 5b) are added into this triple set.*

Step 4. Construct the reuse module. Given $\mathcal{U}_2$ and $\mathcal{H}_3$, we now construct a module M, called a *reuse module*, and a mapping $\mathcal{I}$, in the following steps:

- create a node q for each predicate φ in $\mathcal{U}_2$, and set $\mathcal{I}(q) = \varphi$;
- add an edge (q, b, q'), if $q \neq q_\bullet$ and there is a Hoare triple
$$\{\mathcal{I}(q)\}\ b\ \{\mathcal{I}(q')\} \in \mathcal{H};$$
- add an edge (q, b, q'), if $q = q_\bullet$ and there is a Hoare triple
$$\{\mathcal{I}(q)\}\ \texttt{oldrnk := f(V);}\ b\ \{\mathcal{I}(q')\} \in \mathcal{H};\ \text{and}$$
- the node q is made the initial node (or the accepting node) if $\mathcal{I}(q) = u_\circ$ (or $\mathcal{I}(q) = u_\bullet$).

Recall that the extraction procedure in Step 1 extracts a set of valid Hoare triples from a certified module. The above construction procedure works in the reverse direction, i.e., constructs a certified module from a set of valid Hoare triples.

Example 4. *Consider the predicate set $\mathcal{U}_2 = \{\varphi_0, \varphi_1\}$ and the Hoare triple set $\mathcal{H}_3$ in Figure 5b. According to the above module construction rules, two nodes are created for the reuse module, named q_0 and q_1, and mapped to φ_0 and φ_1, respectively. Additionally, six edges are connected in the reuse module, each standing for one Hoare triple in Figure 5b. The reuse module after construction is shown in Figure 5c.*

Theorem 2. *The reuse module constructed in the* reuse *procedure is terminating.*

Proof. Let $M = (Q, \delta, q_\circ, q_\bullet)$ be a reuse module constructed from a certified module $M^- = (Q^-, \delta^-, q_\circ^-, q_\bullet^-)$ by the *reuse* procedure. By *Step 1* and *Step 4*, $\mathcal{I}(q_\circ) = \mathcal{I}(q_\circ^-)$ and $\mathcal{I}(q_\bullet) = \mathcal{I}(q_\bullet^-)$. Moreover, M^- is a certified module, the first two conditions of Definition 2 are thus satisfied. For any edge $(q, b, q') \in \delta$, it is either inherited from M^-, or added by *Step 3* of the *reuse* procedure. For the former case, the validity of the corresponding Hoare triple is ensured by M^- (a certified module); for the latter case, the processing of *Step 3* guarantees the validity of the corresponding Hoare triple. Therefore, the last two conditions in Definition 2 are also satisfied. Therefore, M is a certified module (by Definition 2), and M is terminating (by Lemma 1). □

4.3 On-Demand Construction

The *reuse* procedure indeed enforces an *eager construction* of the reuse module – the whole module is constructed before it participates in the verification. In practice, the eager construction can be very *expensive*. Note that in *Step 3* of the *reuse* procedure, we need to check the validity of all candidate Hoare triples. In the worst case, the number of validity checking is $O(\delta \cdot |\mathcal{U}|^2)$, where δ is the number of new blocks in P, and $|\mathcal{U}|$ is the number of predicates in $\mathcal{U}$. The validity checking (in *Step 3*) is the most computationally expensive step in the *reuse* procedure.

Recall that the constructed certified modules are devoted to cover ω-traces of P. Not all edges of the reuse module are related to the automaton of P. Validation checking of Hoare triples for these irrelative edges are redundant. To avoid such redundant validation checking, we apply an *on-demand approach* to construct the reuse module on-the-fly. The basic idea is to defer the validity checking to the verification phase, and add the corresponding edges only on demand.

More specifically, let $P^{(l)}$ be the current remainder automaton, and M^- the certified module to be processed, in the language inclusion checking, we compute a new remainder automaton $P^{(l+1)}$, and

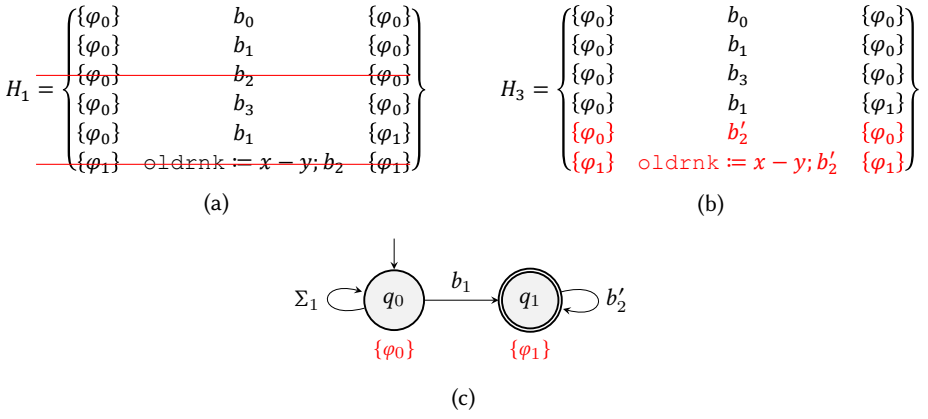


Fig. 5. Construction of the reuse module, where $\varphi_0 \Leftrightarrow oldrnk = \infty$, $\varphi_1 \Leftrightarrow (oldrnk > x - y \wedge oldrnk \geq 0 \wedge y > 0)$

in the meanwhile transform M^- to M. Algorithm 1 depicts our *on-demand construction* procedure. This algorithm consists of two phases. In the first phase, the algorithm transforms the inputted certified module to an (incomplete) reuse module M by applying *Step 1*, *Step 2* and *Step 4* of the *reuse* procedure in turn (lines 1 – 3). Note that the most time-consuming step (i.e. *Step 3*) is skipped, and the edges with respect to the new blocks have not been added to the reuse module.

In the second phase, the algorithm checks the language inclusion by constructing the next remainder automaton $P^{(l+1)} = P^{(l)} \cap \overline{M}$. Denote Q, $Q^{(l)}$ and $Q^{(l+1)}$ the node sets of M, $P^{(l)}$ and $P^{(l+1)}$, respectively. Each node in $Q^{(l+1)}$ is a composition of a node in $Q^{(l)}$ and a node in Q. In the beginning, $Q^{(l+1)}$ contains only the initial node $(q_\circ^{(l)}, q_\circ)$ (line 5). Then the algorithm tries to traverse and add all reached nodes to $Q^{(l+1)}$. Let $(q_1^{(l)}, q_1)$ be the current node to be explored (line 7). We first take an edge of $P^{(l)}$, say $(q_1^{(l)}, b, q_2^{(l)})$ (line 8), then try to simulate this edge in M from q_1 (lines 9–10). If M has an edge, say (q_1, b, q_2), that leads from q_1 on label b to q_2, we reach a state $(q_2^{(l)}, q_2)$ of $Q^{(l+1)}$. Otherwise, we check if such an edge should be added or not, by checking the validity of the corresponding Hoare triple. The returned Hoare triple by $Triple(q_1, b, q_2)$ (line 11) is $\{\mathcal{I}(q_1)\}\ b\ \{\mathcal{I}(q_2)\}$ if $q_1 \neq q_\bullet$, and $\{\mathcal{I}(q_1)\}$ `oldrnk:=f(V);` $b\ \{\mathcal{I}(q_2)\}$ if $q_1 = q_\bullet$. If the reached state $(q_2^{(l)}, q_2)$ is a new state (line 17), we add it to $Q^{(l+1)}$ (line 18) and *waitlist* (line 19).

Theorem 3. *The reuse module constructed in the* on-demand *procedure is terminating.*

Proof. Let M be a reuse module constructed from a certified module M^- by the *on-demand* procedure. Similar to the proof for Theorem 2, $\mathcal{I}(q_\circ)$ and $\mathcal{I}(q_\bullet)$ satisfy the first two conditions of Definition 2. For any edge (q, b, q') of the M, if it is inherited from M^-, the corresponding Hoare triple must be valid (M^- is a certified module); if it is added by the *on-demand* procedure (at line 12), the checking at line 11 guarantees that the corresponding Hoare triple is valid. Therefore, M is a certificate module (by Definition 2) and M is terminating (by Lemma 1). □

4.4 Correctness

We prove the correctness of our incremental termination analysis in this subsection. Let us start with the soundness.

Theorem 4. *If our procedure in Figure 4 returns "terminating", the input program is terminating.*

Algorithm 1: On-demand construction

Input: A remainder automaton $P^{(l)} = (\Sigma, Q^{(l)}, \delta^{(l)}, q_\circ^{(l)}, q_\bullet^{(l)})$ and a certified module M^-.
Output: A new remainder automaton $P^{(l+1)} = (\Sigma, Q^{(l+1)}, \delta^{(l+1)}, q_\circ^{(l+1)}, q_\bullet^{(l+1)})$ and a reuse module $M = (\Sigma, Q, \delta, q_\circ, q_\bullet)$.

```
/* Certified module transformation                                  */
(𝒰, ℋ) ← Extract(M⁻) ;                    /* step 1: extract Hoare triples */
(𝒰, ℋ) ← Filter(𝒰, ℋ) ;                 /* step 2: delete inapplicable triples */
M ← Construct(𝒰, ℋ) ;                     /* step 4: construct reuse module */
/* Language inclusion checking                                      */
Initialize P^(l+1) to be an empty automaton ;
Q^(l+1) ← {(q_∘^(l), q_∘)}, waitlist ← {(q_∘^(l), q_∘)};
while waitlist ≠ ∅ do
    Pop (q_1^(l), q_1) from waitlist;
    foreach (q_1^(l), b, q_2^(l)) ∈ δ^(l) do          /* following an edge of P^(l) */
        foreach q_2 ∈ Q do
            if (q_1, b, q_2) ∉ δ then                  /* edge with same label */
                if Triple(ℐ(q_1), b, ℐ(q_2)) is valid then
                    δ ← δ ∪ {(q_1, b, q_2)} ;          /* add the edge to M */
                else
                    Continue;
                end
            end
            if (q_2^(l), q_2) ∉ Q^(l+1) then           /* a new state of P^(l+1) */
                Q^(l+1) ← Q^(l+1) ∪ {(q_2^(l), q_2)} ;
                waitlist ← waitlist ∪ {(q_2^(l), q_2)};
            end
            δ^(l+1) ← δ^(l+1) ∪ {((q_1^(l), q_1), b, (q_2^(l), q_2))} ;
        end
    end
end
return (P^(l+1), M);
```

Proof. This procedure returns "terminating" only when the language inclusion checking succeeds. In other words, there exists a set of modules $M_0, M_1, \ldots, M_n$, each of which is either constructed from an ω-trace, or transformed from a previously-constructed certified module, such that $\mathcal{L}^\omega(P) \subseteq \mathcal{L}^\omega(M_0) \cup \mathcal{L}^\omega(M_1) \cup \cdots \cup \mathcal{L}^\omega(M_n)$. The modules constructed from ω-traces [Heizmann et al. 2014] are terminating. By Theorem 2 and Theorem 3, the reuse modules are terminating. Thus all modules are terminating. According to Theorem 1, the program is terminating. □

The completeness of our approach cannot be directly guaranteed, since our approach requires to synthesize a ranking function for any lasso program, which in some cases maybe impossible [Ben-Amram and Genaim 2014]. Note that a terminating program can always be decomposed into a

finite set of certified modules $M_0, M_1, \ldots, M_n$ [Heizmann et al. 2014], such that

$$\mathcal{L}^\omega(P) \subseteq \mathcal{L}^\omega(M_0) \cup \mathcal{L}^\omega(M_1) \cup \cdots \cup \mathcal{L}^\omega(M_n).$$

Then, under the assumption of an oracle that synthesizes a ranking function for any lasso program, we prove the *relative completeness* of our approach – if the input program is terminating, the procedure in Figure 4 always deduces this result and returns "terminating".

5 APPLICABILITY TO VARIOUS PROGRAM CHANGES

This section discusses the applicability of our approach to various kinds of program changes.

A program is a control-flow automaton over the alphabet of blocks. In this respect, the program changes can be categorized into two types:

- **Type I (Block changes)**: the changes that add or delete basic blocks to the program.
- **Type II (Structure changes)**: the changes that add or delete control-flow edges to the program.

5.1 *Type I* Changes

This type of changes may alter the alphabet of the program. Let Σ^- and Σ be the alphabet before and after the program changes, respectively.

Type I changes can be efficiently handled by our approach. For any block $b \in \Sigma \cap \Sigma^-$, all previously-established Hoare triples on b are reusable. For any new block $b' \in \Sigma \setminus \Sigma^-$, we process as the Step 3 in Section 4 to prove the validity of new Hoare triples on b'. Finally, a certified module is constructed from the set of valid Hoare triples.

Example 5. *The change of $loop_1$ over $loop_0$ (in Figure 2) is a* Type I *change. It modifies the value of y, and has side effects to the inner loop condition. As discussed before (see Section 4), our incremental approach can efficiently handle this change.*

5.2 *Type II* Changes

This type of changes does not alter the program alphabet. As a result, all previously-constructed certified modules can be directly reused in the current analysis. There is even no need to apply the *Reuse* procedure. The certified module M^- can be directly used as the reuse module. The only consequence of this type of changes is that some traces of $\mathcal{L}^\omega(M^-)$ may no longer belong to $\mathcal{L}^\omega(P)$.

Example 6. *The following two versions of a program show a* Type II *change.*

```
1      x = nondet_int();
2      y = nondet_int();
3      while (y < x) {
4          y = 2 * y;
5      }
6      x = x - 1;
```

```
1      x = nondet_int();
2      y = nondet_int();
3      while (y >= x) {
4          x = x - 1;
5      }
6      y = 2 * y;
```

All certified modules of the first program can be directly applied to the analysis of the second program.

5.3 Mixed Changes

A real-world program revision often relates to both types of program changes.

Example 7. *The following program can also be considered as a changed version of $loop_0$ (in Figure 2), although they looks very different:*

```
1     x = nondet_int();
2     while (x >= 0){
3         x = x - 1;
4     }
5     x = nondet_int();
6     y = 1;
7     while (y < x) {
8         y = 2 * y;
9     }
```

We find two common blocks between the above program and $loop_0$, i.e., `assume x>=0; x=x-1` *and* `x=*` *. The previously-established results on these two blocks are reusable. We still benefit from the incremental approach.*

In conclusion, our incremental approach has a wide applicability. No matter what types of program changes, it is applicable, and can lead a significant saving in computational efforts.

5.4 More Reuse Possibilities

Throughout the paper, programs are assumed to be composed of blocks. Our approach can be naturally adapted to the statement-based program model.

If we consider the termination analysis under the statement-based program model, more information is reusable. Let $b = st_1 st_2 ... st_l$ be a block, where each st_i $(1 \leq i \leq l)$ is a statement. With the block-based program model, any statement change leads the previously-established results on the whole block be abandoned. On the contrary, under the statement-based program model, only Hoare triples on the changed statements need to be abandoned, previously-established results on other unchanged statements can be kept for further analysis.

6 EVALUATION

We implemented our incremental termination analysis algorithm on top of Ultimate Automizer, a verification tool that implemented the decomposition-based approach [Heizmann et al. 2014]. In the following, we refer to the underlying Ultimate Automizer and our enhanced implementation as *Baseline* and *Reuse*, respectively.

We used two benchmarks to evaluate our approach. The first benchmark contains 90 artificial programs and the second benchmark contains 611 real-world programs. In our experiments, a *run set* consists of a series of revisions of the same program. The first revision is analyzed from scratch, and the subsequent revisions are verified with/without reuse. In the following, we call a run of termination analysis on any program revision an *analysis task*, and a run of termination analysis on not-first revision an *regression analysis task*. We compare the performance of *Baseline* and *Reuse* on regression analysis tasks.

All experiments were conducted on a machine with a Intel(R) Core(TM) i7-8700 CPU of 3.2GHz and 16GB RAM. We used Ultimate Automizer of version 0.1.24-57e1ae7 with the default configuration. We took *Z3* as the default SMT solver and set 500 seconds as the time limit for all analysis tasks.

6.1 Experiment on Artificial Benchmark

Benchmark. The first benchmark consists of 90 revisions of 15 distinct programs. The 15 original programs were obtained from [Heizmann et al. 2014]. We then asked five students (2 undergraduate students and 3 first-year graduated students, none participated in this project) to manually create more revisions for these programs. Every student was assigned three programs and was required to

craft five new revisions for each of them. They can alter the program as they want, disregarding the original meanings of the program; especially, they are encouraged to make *loop-relevant* changes; the only requirement is that the program after modification must be compilable.

In total, this benchmark contains 90 analysis tasks, among which 75 are regression analysis tasks.

Results. Experimental results on artificial benchmark are listed in Table 1. The first column assembly ("Run set") lists information about the run set, including the program name ("Programs"), the number of terminating/non-terminating ("T/N") cases in this set, the total number of lines of code ("LoC") for all revisions in this set, and the consumed CPU time ("T_{r_0}") for verifying the first revision of this set. The following two column assemblies report the experimental results of *Baseline* and *Reuse*, respectively. For each approach, we report the total consumed CPU time ("Time") and the total number ("#Iter.") of iterations for this run set. The "#CertModule" column assembly lists information about the certified modules, including the total number ("Avai.") of available certified modules that were generated in the analysis of the previous revision, and the total number ("Reuse") of (non-empty) reuse modules that were transformed from the previous certified modules. The last column ("Speedup") shows the speedup of *Reuse* over *Baseline*, computed by $Baseline.time/Reuse.time$. Note that only statistics for regression analysis tasks are counted. The table is sorted by the "speedup" column. The last two rows ("Sum" and "Average") report the total and average amounts of all rows in the table, respectively.

From Table 1, we observed that our approach outperforms *baseline* in all run sets, with a x4.16 average speedup. Looking at the `edn.t2` run set, the maximum speedup is x9.99. Even in the worst case `ud.t2`, *reuse* outperforms *Baseline* by x1.95 of speedup.

The "#Iter." columns can reveal the main reason of the efficiency of our technique. Observing these columns, we found that each run set has a dramatic reduction of iterations. On average, our approach decreases the number of iterations by 89.8%. From the "#CertModule" column assembly, we conclude that our algorithm is practical on these artificial revisions. The average proportion of the reuse modules among all previously-constructed certified modules ("Reuse" / "Avai.") is 89.6%.

Results for a single run set. To address the detailed mechanism of our approach and to demonstrate the acceleration of our approach on three types of program changes, we present the results for `bubbleSort.t2` in Table 2.

The first column ("Rev.") indicates the number of revisions, where the first row with *Rev. = 0* represents the original revision of `bubbleSort.t2`. The second column ("Type") presents the type of program changes applied to this revision (see Section 5). The column ("Rst.") presents the verification result (T for terminating and N for non-terminating). All other columns have the same meanings as in Table 1.

During the preparation of this run set, we deleted an irrelevant variable and the corresponding statements (*Type I* changes) from *Rev. 0* to *Rev. 1*. Among the 8 certified modules, 7 can be reused with the iterations decreased from 7 to 2, achieving a speedup of x12.86. In *Rev. 2*, we modify several variables via adding some statements (*Type I* changes). By reusing all 8 previously-constructed certified modules successfully, *reuse* reduces iterations from 5 to 0 and gets a x4.42 speedup. *Rev. 3* changes control-flow edges (*Type II* changes) by inserting `goto` statements into the program causing the program to be non-terminating. *Baseline* spends some time on the last iteration for finding a non-terminating arguments. By reusing 5 out of 8 certified modules, our approach reduces 80% of iterations on *Rev. 3*. Although the last iteration is a little bit time-consuming, we still get a x1.83 speedup. *Rev. 4* deletes some basic blocks directly and removes two `goto` statements (*Mixed* changes). The program turns to be much simpler, and *Baseline* only needs 4 iterations to get the result. Only 2 certified modules are successfully reused from 8 previously-constructed certified modules. *Reuse* decreases 2 iterations and gets a x1.37 speedup. The last revision changes

Table 1. Experimental results on artificial benchmark

Run Set				Baseline		Reuse		#CertModule		
Programs	T/N	Loc	T_{r_0}	Time	#Iter.	Time	#Iter.	Avai.	Reuse	Speedup
edn.t2	5/0	1471	223.73	1059.02	666	106.01	14	725	713	9.99
firewire.t2	5/0	897	14.02	66.97	77	10.89	4	100	97	6.15
eric.t2	2/3	266	3.71	25.23	28	4.91	5	40	35	5.14
a.10.c.t2	2/3	922	87.80	320.86	39	72.61	9	45	33	4.42
traverse_twice	5/0	7132	4.74	19.54	28	6.01	8	30	26	3.25
sas2.t2	2/3	1364	17.02	53.55	42	17.44	8	85	74	3.07
reverse.t2	4/1	9683	15.71	54.62	21	19.92	7	20	16	2.74
b.f20.t2	5/0	772	3.00	13.60	31	4.98	7	55	44	2.73
consts1.t2.c	4/1	204	1.31	6.54	18	2.40	3	15	12	2.72
s3-work.t2	4/1	23137	11.90	68.71	46	27.20	1	55	53	2.53
sumit.t2	3/2	420	4.64	15.99	29	6.86	8	50	39	2.33
spiral.t2	5/0	333	4.70	26.09	81	11.60	16	80	69	2.25
bubbleSort.t2	4/1	547	9.78	22.93	28	10.66	8	40	28	2.15
mc91.t2	4/1	237	4.49	23.54	53	11.50	14	70	45	2.05
ud.t2	5/0	1402	158.92	421.27	206	215.72	30	360	302	1.95
Sum	59/16	48787	565.48	2198.45	1393	528.74	142	1770	1586	-
Average	-	650	7.54	29.31	18.57	7.05	1.89	23.60	21.15	4.16

Table 2. Results for run set `bubbleSort.t2`

			Baseline		Reuse		#CertModule		
Rev.	Type	Rst.	Time	#Iter.	Time	#Iter.	Avai.	Reuse	Speedup
0	-	T	9.78	7	-	-	-	-	-
1	*Type I*	T	9.87	7	0.77	2	8	7	12.86
2	*Type I*	T	1.60	5	0.36	0	8	8	4.42
3	*Type II*	N	1.48	5	0.81	1	8	5	1.83
4	*Mixed*	T	0.98	4	0.72	2	8	2	1.37
5	*Type I*	T	9.00	7	8.01	3	8	6	1.12
Sum	-	-	22.93	28	10.66	8	40	28	-
Average	-	-	4.59	5.60	2.13	1.60	8.00	5.60	2.15

a loop condition (*Type I* changes) which affects the corresponding ranking function. To prove the termination of *Rev. 5*, our algorithm needs to recompute new ranking functions. Even so, *Reuse* can get a fair acceleration (x1.12) against *Baseline* by reducing 4 iterations.

On average, *Reuse* gets x2.15 speedup on run set `bubbleSort.t2` by reusing 70% of certified modules. This experiment reveals the relevance of the performance of our algorithm to the adopted program changes (refer to Section 5): no matter which type of program changes, our algorithm is always applicable and can often lead to significant computational saving.

6.2 Experiment on Real-World Benchmark

Benchmark. This set of programs were collected from `Codeforces` [2], a website that hosts competitive programming contests. The program collection was limited to the first 100 contests hosted on this website, and filtered by the following restrictions:

- C programs;
- compilable by `gcc`;
- contains at least 2 loops; and
- Ultimate Automizer can prove termination/non-termination in 500 seconds.

The initial collection contains some trivial, identical or isolated revisions. We took the following steps to remove them:

- We grouped the programs submitted by some contestant for a certain problem into a run set;
- We removed trivially non-terminating programs that contains statements like `while(scanf("%d",&n))` ;
- We compared each pair of adjacent revisions using the following command:
  ```
  $ git diff --ignore-cr-at-eol --ignore-all-space
      --ignore-blank-lines --ignore-space-change --no-index -U0
  ```
  and removed one if they are identical;
- We removed the run sets with only one program revision.

Finally, we get 276 run sets of 611 program revisions. Therefore, this benchmark contains 611 analysis tasks, among which 335 are regression analysis tasks.

Results. Experimental results on real-world benchmark are listed in Table 3. Due to page limitation, we restrict this table to the 20 best and 20 worst run sets, sorted by the "speedup" column.

Looking at the 20 best run sets, our approach gets noteworthy acceleration. The reason is that for almost of these run sets, the previously-constructed certified modules can all be reused and the transformed reuse modules can cover all ω-traces of the program. Thus, the termination is proved immediately.

For the worst several run sets, our approach gets deceleration. The reason is multiple-folds. One is that the program changes make the subsequent revisions much simpler and their termination becomes easy to determine (even for *Baseline*), such as `59A-usr206`, `6C-usr199` and `59A-usr047`. These cases are not friendly to our approach, since there are a lot of previously-constructed certified modules, while most of them are useless for proving termination of the subsequent revisions, leading worthless computation of our approach. For example, when analyzing the second revision of `59A-usr206`, our approach only reduces 5 iterations by reusing 16 out of 54 certified modules generated by the first version.

Another reason is that the program changes may have impact to the previous termination arguments, causing the previously-constructed certified modules invalid for the current analysis, such as `15C-usr072`, `4A-usr082`.

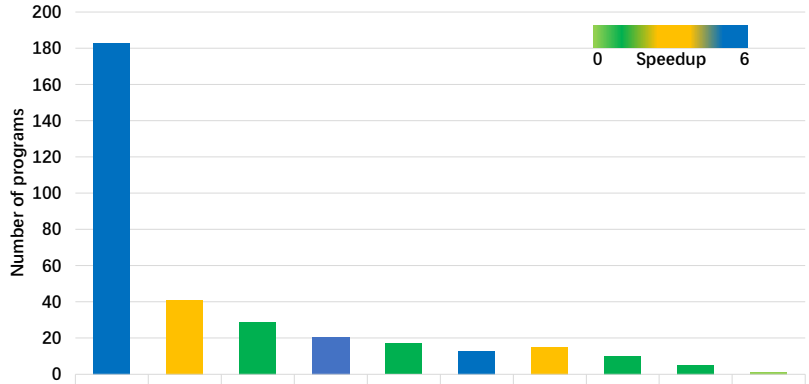
The third reason is that the number of the previously-constructed certificate modules is too small (e.g. ≤ 2), such as `1B-usr035`, `14D-usr103` and `58A-usr154`. As a result, there is little chance to get a non-empty reuse module, so as to accelerate the current analysis. This can be avoided by setting a minimal number of certified modules. Only when the available number of certified modules exceeds that number, can we apply the incremental approach.

Example 8. *We present two revisions of 4A-usr082 in the following. The right version heavily changes the nested loop (lines 4 – 5) of the left version, leading the termination arguments for the left*

[2] https://codeforces.com

Table 3. Experimental results on real-world benchmark

Run set				Baseline		Reuse		#CertModule		
Programs	T/N	Loc	T_{r_0}	Time	#Iter.	Time	#Iter.	Avai.	Reuse	Speedup
96A-usr146	2/0	121	39.99	39.04	6	0.35	0	4	4	112.84
94A-usr112	2/0	122	178.52	178.63	37	2.09	0	36	36	85.27
66A-usr023	3/0	80	12.07	24.35	6	0.37	0	4	4	65.20
36A-usr085	3/0	120	22.38	44.51	16	0.93	0	14	14	47.69
71A-usr056	2/0	84	22.38	22.08	13	0.52	0	11	11	42.36
58A-usr237	2/0	98	16.66	16.64	6	0.44	0	5	5	37.99
94A-usr167	5/0	203	26.19	137.34	104	4.72	4	101	100	29.11
58A-usr234	2/0	114	8.22	8.23	5	0.28	0	4	4	29.00
16A-usr055	2/0	69	41.44	40.88	18	1.50	0	17	17	27.18
58A-usr184	2/0	82	41.86	52.37	35	1.96	0	23	23	26.75
2A-usr133	0/5	904	15.33	598.35	18	24.35	8	12	8	24.57
71A-usr041	2/0	58	19.85	19.41	17	0.83	0	16	16	23.34
58A-usr150	2/0	72	3.11	3.30	2	0.14	0	1	1	22.75
58A-usr002	2/0	68	3.55	7.42	7	0.34	0	4	4	21.71
71A-usr058	2/0	48	5.66	5.56	4	0.26	0	3	3	21.63
9A-usr068	2/0	55	9.87	6.96	10	0.33	0	9	9	21.17
78A-usr217	2/0	68	5.36	15.67	15	0.75	0	14	14	21.02
25A-usr227	4/0	142	6.52	19.53	18	0.97	0	15	15	20.17
11A-usr091	2/0	60	95.97	41.79	5	2.17	2	8	3	19.29
38A-usr049	2/0	33	3.79	3.91	3	0.21	0	2	2	18.64
⋮				⋮				⋮		
94A-usr099	2/0	48	1.36	1.28	7	1.36	6	8	1	0.94
92A-usr207	0/2	68	0.45	0.45	2	0.49	1	1	1	0.93
58A-usr079	2/0	68	0.96	0.97	2	1.04	2	1	1	0.93
46C-usr003	2/0	76	0.36	0.52	4	0.56	4	2	0	0.93
58A-usr138	2/0	115	3.52	0.91	2	0.98	2	4	1	0.93
58A-usr017	2/0	56	3.57	0.86	2	0.94	2	4	1	0.91
71A-usr001	2/0	85	0.61	0.44	4	0.48	4	4	1	0.91
61A-usr046	2/0	57	1.62	1.70	4	1.88	4	3	1	0.91
58A-usr000	2/0	129	1.20	1.09	2	1.26	2	1	1	0.86
59A-usr047	2/0	88	14.88	2.63	7	3.22	3	37	12	0.82
1B-usr035	2/0	148	0.33	0.26	2	0.32	2	2	1	0.81
14D-usr103	2/0	132	0.52	0.42	3	0.52	3	2	1	0.81
81A-usr118	0/2	36	3.62	2.74	7	3.46	4	7	4	0.79
49A-usr192	2/0	54	1.90	1.43	3	2.18	4	4	3	0.66
4A-usr082	2/0	88	7.95	63.24	10	96.62	9	10	1	0.65
15C-usr072	2/0	37	32.86	20.52	8	32.58	6	8	2	0.63
58A-usr188	2/0	156	6.86	5.63	10	9.02	8	10	7	0.62
58A-usr154	2/0	86	0.93	18.80	28	35.83	42	1	0	0.52
6C-usr199	2/0	54	130.02	3.71	4	8.55	3	13	2	0.43
59A-usr206	2/0	79	99.34	4.47	7	14.00	2	54	16	0.32
Sum	586/25	24627	4054.47	4555.25	2962	1476.76	984	2677	1902	-
Average	-	40.31	14.69	13.60	8.84	4.41	2.94	7.99	5.68	3.08



DiffRate	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
#Prgrams	183	41	29	21	17	13	15	10	5	1
Speedup	4.52	1.98	1.18	3.90	1.28	5.48	1.84	0.98	1.11	0.93

Fig. 6. Experimental results divided by *DiffRate*

version inapplicable to the right version. As a result, the right version can only reuse 1 out of the 10 previously-constructed certified modules. In the end, Reuse *takes more time than* Baseline.

```
1  int w,i,j,a;
2  scanf("%d", &w);
3  if(w%2==1)
4      for(i=2;i<w;i=i+2)
5          for(j=w-1;j>0;j=j-2){
6            a=i+j;
7            if(w==a)
8                return 0;
9          }
10 ...
```

```
1  int w,i,j,a;
2  scanf("%d", &w);
3  if(w%2==1)
4      for(i=2;i<=w/2;i=i+2)
5          for(j=w-1;j>=w/2;j=j-2){
6            a=i+j;
7            if(w==a)
8                return 0;
9          }
10 ...
```

On average, *Baseline* requires 8.84 iterations and 13.60 seconds to complete an analysis task. In contrast, *Reuse* needs 2.94 iterations and 4.41 seconds. Our approach achieves an overall x3.08 speedup over *Baseline*, reusing 5.68 out of 7.99 certified modules.

6.2.1 Degree of Program Changes. To further analyze the results in Table 3, we use *DiffRate* to measure the degree of program changes, and then analyze the impact of *DiffRate* to the efficiency of our approach. The *DiffRate*, i.e., the rate of different lines of codes, is calculated by

$$DiffRate = \frac{\#deleted\_lines + \#added\_lines}{\#LoC_1 + \#LoC_2},$$

where $\#LoC_1$ and $\#LoC_2$ are lines of codes of the two comparative programs, respectively.

We first divided *DiffRate* into 10 ranges: 0%–10%, 10%–20%, ..., 90%–100%. Then, we counted the number of programs in each range, and summarized the average speedup of our approach among the programs in each region. The statistics results are depicted in Figure 6. The first two histograms indicate there are 183 programs with *DiffRate* less than 10%, and 41 programs with *DiffRate* between 10% and 20%. These programs have relatively small changes; the speedup of our approach on these programs is significant. It is worth noting that the cases with *DiffRate* between 50% and 60% also show significant speedup, which suggest the capability of our approach on large changes. However,

Table 4. Comparison on *loop-relevant* and *loop-irrelevant* changes

		Baseline		Reuse		#CertModule		
Relevant?	#Prog.	Time	#Iter.	Time	#Iter.	Avai.	Reuse	Speedup
No	66	449.64	558	95.77	64	495	459	4.69
Yes	269	4105.61	2404	1380.99	920	2182	1443	2.97
Sum	335	4555.25	2962	1476.76	984	2677	1902	3.08

from Figure 6, we did not observe a clear relation between the degree of program changes and the efficiency of our approach. This explains that our approach is not very *sensitive* to the degree of program changes (also witnessed in Section 5).

Example 9. *Consider the following two revisions of 94A-usr005. Although the right version changes a lot against the left version (even altering the number of loops), our approach can still reuse 25 out of 26 certified modules, reduces 24 iterations and achieves x14.33 acceleration.*

```
int i,j,t[8];
char x[8][11],y[10][11];
for(i=0;i<8;i++){
    for(j=0;j<10;j++)
        scanf("%c ",&x[i][j]);
    x[i][10]='\0';
}
for(i=0;i<10;i++)
    gets(y[i]);
for(i=0;i<8;i++)
    for(j=0;j<10;j++)
        if(strcmp(x[i],y[j])==0)
            t[i]=j;

for(i=0;i<8;i++){
    printf("%d",t[i]);
}
```

```
int i,j,o;
char str[8][11],str2[10][11];
for(j=0;j<8;j++)
    for(i=0;i<10;i++)
        scanf("%c",&str[j][i]);

gets(str2[0]);
for(i=0;i<10;i++)
    gets(str2[i]);
for(i=0;i<8;i++)
   for(j=0;j<10;j++){
      for(o=0;o<10;o++)
        if(str[i][o]!=str2[j][o])
            break;
      if(o==10)
         printf("%d",j);
   }
```

6.2.2 Loop-Relevant *vs.* Loop-Irrelevant *Changes.* To further analyze the results in Table 3, we applied Frama-C [Cuoq et al. 2012] (more specifically, its engine of *change impact analysis*) to distinguish *loop-relevant* and *loop-irrelevant* changes. A *loop-relevant* change has impact to loops (either loop conditions or loop bodies) of the program. We then compared the efficiency of our approach with these two kinds of changes.

The statistics results are listed in Table 4, where the first column ("Relevant?") indicates if the change is *loop-relevant* or not. The second column ("# Prog.") presents the number of programs in each type of changes. All other columns have the same meanings as in Table 1.

From Table 4, we observed that our approach get a significant acceleration (x4.69) for 66 *loop-irrelevant* changes. This is consistent with our intuition. Program termination is closely related to the loops. If a change has no impact to loops, we have much bigger chance (93% in the table) to keep all previously-constructed certified modules. We also observed that our approach gets considerable acceleration (x2.97) for 269 *loop-relevant* changes. This result is more meaningful, witnessing the wide applicability of our approach for various program changes.

Example 10. *The following two revisions of 12A-usr172 show a* Type II *change. The left version is obviously non-terminating. The right version deletes the innermost* `while` *loop and becomes terminating. Although the terminating results of these two programs are different, our approach still benefits from the certified modules generated by the left program, and gets a 4.91x speedup.*

```
1  char x[3][3];
2  int i,j;
3  for(i=0;i<3;i++){
4      for(j=0;j<3;j++){
5          do{
6              scanf("%c",&x[i][j]);
7          }while(x[i][j]=='\n');
8      }
9  }
```

```
1  char x[3][3];
2  int i,j;
3  for(i=0;i<3;i++){
4      for(j=0;j<3;j++){
5          scanf("%c",&x[i][j]);
6          if(x[i][j]=='\n')
7              scanf("%c",&x[i][j]);
8      }
9  }
```

Example 11. *The following two revisions of 96A-usr208 show a* Mixed *change. The right version deletes a statement at line 4 and adds a statement at line 9. Among the 7 certified modules generated by the left version, only 2 can be reused, due to the impact of program changes to the termination arguments. Nevertheless, our method still achieves a non-trivial acceleration of x1.79.*

```
1      char a[150];
2      int i,j,flag=0;
3      scanf("%s",a);
4      for(i=0;a[i]!='\0';i++){
5          for(j=0;j<7;j++){
6              if(a[i] != a[i+j])
7                  break;
8          }
9
10         if(j==7){
11             flag=1;break;
12         }
13     }
```

```
1      char a[150];
2      int i,j,flag=0;
3      scanf("%s",a);
4      for(i=0;a[i]!='\0';){
5          for(j=0;j<7;j++){
6              if(a[i] != a[i+j])
7                  break;
8          }
9          i+=j;
10         if(j==7){
11             flag=1;break;
12         }
13     }
```

In summary, by reusing the previously-constructed certified modules, our incremental approach can significantly reduce the time consumption for termination verification, and has good applicability to various types of real-world program changes.

7 RELATED WORK

7.1 Termination Analysis

Termination analysis was investigated mainly in two directions, proving termination and proving non-termination. Our paper was focused on termination proving only. But For the sake of completeness, we give a brief overview on both directions.

Proving Termination. A wide range of methods utilize iterative reasoning and counterexample driven method for proving termination [Brockschmidt et al. 2013; Cook et al. 2006a,b, 2013; Gulwani et al. 2008; Harris et al. 2010; Heizmann et al. 2014; Kroening et al. 2008, 2010; Larraz et al. 2013; Podelski and Rybalchenko 2004; Podelski and Rybalchenko 2005, 2011; Ströder et al. 2017]. These methods usually rely on termination arguments (e.g., ranking function) to show that their transition models cannot be executed forever. Most methods find termination arguments using constraint-based synthesis, among which techniques [Bagnara et al. 2012; Ben-Amram and Genaim 2013; Colóon and Sipma 2001; Cook et al. 2010; Podelski and Rybalchenko 2004] are based on linear

arithmetic constraint solving, and techniques [Bradley et al. 2005a,b] are based on non-linear arithmetic constraint solving. In some methods [Brockschmidt et al. 2013; Cook et al. 2006a; Urban et al. 2016], the termination proving is reduced to a safety verification problem. For example, Urban et al. [Urban et al. 2016] used a safety verifier to build the termination arguments. Bound analysis [Gulwani et al. 2009; Nori and Sharma 2013] goes another direction by computing a symbolic upper bound on the number of iterations for each loop. It often works in the infer-and-validate framework – first infers a candidate bound and then validate it. If we get a valid bound for each loop, we prove the termination of the program.

Termination arguments are usually harder to solve for the general programs with branches and nesting. Consequently, many methods were focused on the lasso programs, that is a simpler type of programs containing only one infinite path. The termination argument for this type of programs can be solved rather efficiently [Ben-Amram and Genaim 2013, 2014, 2015, 2017; Bradley et al. 2005a; Cook et al. 2010; Heizmann et al. 2013b; Kroening et al. 2008; Podelski and Rybalchenko 2004]. In particular, Leike et al. [Leike and Heizmann 2014] use linear ranking templates for the constraint-based synthesis of termination arguments for linear loop programs. For the general program, some methods[Borralleras et al. 2017; Heizmann et al. 2014; Le et al. 2015; Urban et al. 2016] proposed to decompose the original program into components such that termination arguments for each component are quite simple. For example, HipTNT [Le et al. 2015] and SeaHorn [Urban et al. 2016] decomposed the state space of the program and infer ranking functions for each component separately. Ultimate Automizer found lasso traces in the control flow automaton iteratively, and converts them to lasso programs and finds termination arguments only for lasso-shaped programs.

All these research works were focused on a single program version. In this paper, we address the termination proving for evolving programs. We proposed an incremental termination proving approach. Experimental results show very promising performance of our approach.

Proving Non-termination. Most techniques for proving non-termination [Bakhirkin and Piterman 2016; Brockschmidt et al. 2012; Cook et al. 2014; Gupta et al. 2008; Leike and Heizmann 2018] are based on the searching for lasso-shaped traces or the discovering of recurrence sets. For example, TNT [Gupta et al. 2008] first enumerates lasso-shaped candidate paths for counterexamples dynamically, and then proves their feasibility statically. Leike et al. [Leike and Heizmann 2018] proved the non-terminating linear lasso programs by deciding the existence of a geometric non-termination argument, using a nonlinear algebraic ∃-constraint. Ultimate Automizer proved non-termination of programs by deciding if the selected lasso trace is non-terminating or not. If it is non-terminating, this lasso trace is returned as a counterexample. A minor range of tools reduce termination analysis to safety attracts significant attention. In particular, Velroyen et al. [Velroyen and Rümmer 2008] showed that terminating states of a program are unreachable from certain initial states via invariant generation. Gulwani et al. [Gulwani et al. 2008] used weakest precondition inference to discover the most-general characterization of the inputs under which the original program is non-terminating. Chen et al. [Chen et al. 2014] iteratively eliminated terminating traces through a loop by adding extra assumptions. Ultimate Automizer eliminates terminating lasso traces via the language-theoretic difference (complementation and intersection) of Büchi automata.

7.2 Incremental Verification

Incremental verification was also investigated mainly in two directions, the verification of differences and the reuse of intermediate results.

Verification of Differences. In this line of research, one attempts to establish the correctness of the new program by proving its (conditional) equivalence to an old and verified program. Many techniques [Backes et al. 2013; Beyer et al. 2012; Böhme et al. 2013; Chaki et al. 2012; Fedyukovich

et al. 2016; Felsing et al. 2014; Godlin and Strichman 2009; Mora et al. 2018; Rungta et al. 2012; Trostanetski et al. 2017; Yang et al. 2014] have been proposed in this area of research. For instance, Golden et al. [Godlin and Strichman 2009] proposed a technique for proving conditional equivalence of two programs by abstraction and decomposition of procedures. Backes et al. [Backes et al. 2013] proposed to distinguish the program behaviors that are impacted by the changes. Only the impacted program behaviors needed to be considered during the regression verification. Beyer et al. [Beyer et al. 2012] proposed the conditional model checking, which outputs a condition such that the program satisfies the specification under this condition. Felsing et al. [Felsing et al. 2014] reduced the equivalence proving of two related imperative integer programs to Horn constraints over uninterpreted predicates, and then solved the constraints using an SMT solver. Moreover, Rungta et al. [Rungta et al. 2012] presented a technique for interprocedural change impact analysis. Yang et al. [Yang et al. 2014] introduced an incremental approach for checking the conformance of code against different properties. Trostanetski et al. [Trostanetski et al. 2017] analyzed the semantic difference between successive revisions. Fedyukovich et al. [Fedyukovich et al. 2016] established the property directed equivalence of two programs by discovering a fresh property that actually holds using simulation.

Reuse of Intermediate Results. This area studies the reuse of previously-generated results to the current verification. A variety of information has been proposed for reuse.

Some researchers[Alt et al. 2017; Conway et al. 2005; Henzinger et al. 2003; Lauterburg et al. 2008; Sery et al. 2012; Yang et al. 2009] proposed to keep the reached state space and reuse them in the further verification runs. The rationale of these techniques is that state spaces of consecutive versions tend to be similar. For instance, Visser et al.[Visser et al. 2012] noticed the importance of constraint solving for symbolic execution. They proposed to cache and reuse the results of constraint solving. This approach was further improved in [Aquino et al. 2015; Jia et al. 2015] from different aspects. Beyer et al.[Beyer et al. 2013] proposed to use abstract precision as the intermediate result. [Fedyukovich et al. 2013; Sery et al. 2012] proposed a regression verification technique by means of interpolation-based procedure summaries. Pastore et al. [Pastore et al. 2014] proposed a method to validate that an already tested code has not been broken by an upgrade, It maintains a test suite that can be used to revalidate the software as it evolves.

Rothenberg et al. [Rothenberg et al. 2018] proposed to reuse the sequence of Floyd-Hoare automata computed in the trace abstraction. Their approach differs from ours in the following aspects. Firstly, the targeted problems are different. Their approach mainly considers the safety verification, while our paper is focused on the termination proving. To the best of our knowledge, our approach is the first attempt to incremental termination analysis. Secondly, the reused intermediate results are different. The Floyd-Hoare automaton was applied in [Rothenberg et al. 2018] to recognize a set of *finite* traces, while in our paper, we use certificate module to recognize a set of *infinite* traces. Last but not least, a so-called *lazy reuse* was proposed in [Rothenberg et al. 2018], which looks similar but is indeed different to our *on-demand construction* – the former is a scheme of module reuse, while the latter is a scheme of module construction. More specifically, considering the basic scheme of our approach in Figure 4, the on-demand construction will lead the full construction of reuse modules be deferred to the language inclusion checking phase, while the lazy reuse in [Rothenberg et al. 2018] will add a new "reuse" phase after the language inclusion checking and defer the subtraction of reuse modules to this new phase.

8 CONCLUSION

We proposed in this paper an incremental technique for termination analysis of evolving programs. A transformation procedure was developed to reuse certified modules across different program

versions. The proposed approach is applicable to various types of program changes. Its soundness and relative completeness were formally proved. We implemented the approach in Ultimate Automizer. Experimental results show dramatic improvement of our approach.

ACKNOWLEDGMENTS

This work was partially funded by the National Key R&D Program of China (No. 2018YFB1308601), the NSF of China (No. 61672310 and No. 62072267), the CDZ project CAP (No. GZ 1023) and the Guangdong Science and Technology Department (No. 2018B010107004).

REFERENCES

Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. 2017. HiFrog: SMT-based Function Summarization for Software Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–213. https://doi.org/10.1007/978-3-662-54580-5_12

Andrea Aquino, Francesco A. Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. 2015. Reusing Constraint Proofs in Program Analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 305–315. https://doi.org/10.1145/2771783.2771802

John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. 2013. Regression Verification Using Impact Summaries. In *Model Checking Software*, Ezio Bartocci and C. R. Ramakrishnan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–116. https://doi.org/10.1007/978-3-642-39176-7_7

Roberto Bagnara, Fred Mesnard, Andrea Pescetti, and Enea Zaffanella. 2012. A new look at the automatic synthesis of linear ranking functions. *Information and Computation* 215 (2012), 47 – 67. https://doi.org/10.1016/j.ic.2012.03.003

Alexey Bakhirkin and Nir Piterman. 2016. Finding Recurrent Sets with Backward Analysis and Trace Partitioning. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–35. https://doi.org/10.1007/978-3-662-49674-9_2

Amir M. Ben-Amram and Samir Genaim. 2013. On the Linear Ranking Problem for Integer Linear-Constraint Loops. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/2429069.2429078

Amir M. Ben-Amram and Samir Genaim. 2014. Ranking Functions for Linear-Constraint Loops. *J. ACM* 61, 4, Article 26 (July 2014), 55 pages. https://doi.org/10.1145/2629488

Amir M. Ben-Amram and Samir Genaim. 2015. Complexity of Bradley-Manna-Sipma Lexicographic Ranking Functions. In *Computer Aided Verification*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 304–321. https://doi.org/10.1007/978-3-319-21668-3_18

Amir M. Ben-Amram and Samir Genaim. 2017. On Multiphase-Linear Ranking Functions. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 601–620. https://doi.org/10.1007/978-3-319-63390-9_32

Dirk Beyer, Thomas A Henzinger, M Erkan Keremoglu, and Philipp Wendler. 2012. Conditional model checking: a technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Austin, TX, USA). ACM, 57.

Dirk Beyer, Stefan Löwe, Evgeny Novikov, Andreas Stahlbauer, and Philipp Wendler. 2013. Precision Reuse for Efficient Regression Verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 389–399. https://doi.org/10.1145/2491411.2491429

Marcel Böhme, Bruno C d S Oliveira, and Abhik Roychoudhury. 2013. Partition-based regression verification. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA). IEEE Press, 302–311. https://doi.org/10.1109/ICSE.2013.6606576

Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. 2017. Proving Termination Through Conditional Termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–117. https://doi.org/10.1007/978-3-662-54577-5_6

Aaron R Bradley and Zohar Manna. 2007. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005a. Linear Ranking with Reachability. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 491–504.

https://doi.org/10.1007/11513988_48

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2005b. Termination Analysis of Integer Linear Loops. In *CONCUR 2005 – Concurrency Theory*, Martín Abadi and Luca de Alfaro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 488–502. https://doi.org/10.1007/11539452_37

Marc Brockschmidt, Byron Cook, and Carsten Fuhs. 2013. Better Termination Proving through Cooperation. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 413–429. https://doi.org/10.1007/978-3-642-39799-8_28

Marc Brockschmidt, Thomas Ströder, Carsten Otto, and Jürgen Giesl. 2012. Automated Detection of Non-termination and NullPointerExceptions for Java Bytecode. In *Formal Verification of Object-Oriented Software*, Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 123–141. https://doi.org/10.1007/978-3-642-31762-0_9

Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. 2012. Regression Verification for Multi-threaded Programs. In *Verification, Model Checking, and Abstract Interpretation*, Viktor Kuncak and Andrey Rybalchenko (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 119–135. https://doi.org/10.1007/978-3-642-27940-9_9

Hong-Yi Chen, Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Proving Nontermination via Safety. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–171. https://doi.org/10.1007/978-3-642-54862-8_11

Michael A. Colóon and Henny B. Sipma. 2001. Synthesis of Linear Ranking Functions. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tiziana Margaria and Wang Yi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 67–81. https://doi.org/10.1007/3-540-45319-9_6

Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. 2005. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 449–461. https://doi.org/10.1007/11513988_45

Byron Cook, Carsten Fuhs, Kaustubh Nimkar, and Peter O'Hearn. 2014. Disproving termination with overapproximation. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 67–74. https://doi.org/10.1109/FMCAD.2014.6987597

Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. 2010. Ranking Function Synthesis for Bit-Vector Relations. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 236–250. https://doi.org/10.1007/978-3-642-12002-2_19

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006a. Termination Proofs for Systems Code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. Association for Computing Machinery, New York, NY, USA, 415–426. https://doi.org/10.1145/1133981.1134029

Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006b. Terminator: Beyond Safety. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 415–418. https://doi.org/10.1007/11817963_37

Byron Cook, Abigail See, and Florian Zuleger. 2013. Ramsey vs. Lexicographic Termination Proving. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 47–61. https://doi.org/10.1007/978-3-642-36742-7_4

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C. In *Software Engineering and Formal Methods*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2016. Property Directed Equivalence via Abstract Simulation. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 433–453. https://doi.org/10.1007/978-3-319-41540-6_24

Grigory Fedyukovich, Ondrej Sery, and Natasha Sharygina. 2013. eVolCheck: Incremental Upgrade Checker for C. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 292–307. https://doi.org/10.1007/978-3-642-36742-7_21

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 349–360. https://doi.org/10.1145/2642937.2642987

Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proceedings of the 46th Annual Design Automation Conference* (San Francisco, California) *(DAC '09)*. Association for Computing Machinery, New York, NY, USA, 466–471. https://doi.org/10.1145/1629911.1630034

Sumit Gulwani, Krishna K. Mehra, and Trishul Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 127–139. https://doi.org/10.1145/1480881.1480898

Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. 2008. Program Analysis as Constraint Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 281–292. https://doi.org/10.1145/1375581.1375616

Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 2008. Proving Non-Termination. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 147–158. https://doi.org/10.1145/1328438.1328459

William R. Harris, Akash Lal, Aditya V. Nori, and Sriram K. Rajamani. 2010. Alternation for Termination. In *Static Analysis*, Radhia Cousot and Matthieu Martel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–319. https://doi.org/10.1007/978-3-642-15769-1_19

Fei He, Shu Mao, and Bow-Yaw Wang. 2016. Learning-Based Assume-Guarantee Regression Verification. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 310–328. https://doi.org/10.1007/978-3-319-41528-4_17

Matthias Heizmann, Jochen Hoenicke, Jan Leike, and Andreas Podelski. 2013b. Linear Ranking for Linear Lasso Programs. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Mizuhito Ogawa (Eds.). Springer International Publishing, Cham, 365–380. https://doi.org/10.1007/978-3-319-02444-8_26

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2013a. Software Model Checking for People Who Love Automata. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2014. Termination Analysis by Learning Terminating Programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 797–813. https://doi.org/10.1007/978-3-319-08867-9_53

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. 2003. *Extreme Model Checking*. Springer Berlin Heidelberg, Berlin, Heidelberg, 332–358. https://doi.org/10.1007/978-3-540-39910-0_16

Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/2771783.2771806

Daniel Kroening, Natasha Sharygina, Stefano Tonetta, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2008. Loop Summarization Using Abstract Transformers. In *Automated Technology for Verification and Analysis*, Sungdeok (Steve) Cha, Jin-Young Choi, Moonzoo Kim, Insup Lee, and Mahesh Viswanathan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 111–125. https://doi.org/10.1007/978-3-540-88387-6_10

Daniel Kroening, Natasha Sharygina, Aliaksei Tsitovich, and Christoph M. Wintersteiger. 2010. Termination Analysis with Compositional Transition Invariants. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 89–103. https://doi.org/10.1007/978-3-642-14295-6_9

D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. 2013. Proving termination of imperative programs using Max-SMT. In *2013 Formal Methods in Computer-Aided Design* (Portland, OR, USA). IEEE, 218–225. https://doi.org/10.1109/FMCAD.2013.6679413

Steven Lauterburg, Ahmed Sobeih, Darko Marinov, and Mahesh Viswanathan. 2008. Incremental state-space exploration for programs with dynamically allocated data. In *Proceedings of the 30th international conference on Software engineering* (Leipzig, Germany). ACM, 291–300. https://doi.org/10.1145/1368088.1368128

Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. 2015. Termination and Non-Termination Specification Inference. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 489–498. https://doi.org/10.1145/2737924.2737993

Jan Leike and Matthias Heizmann. 2014. Ranking Templates for Linear Loops. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 172–186. https://doi.org/10.1007/978-3-642-54862-8_12

Jan Leike and Matthias Heizmann. 2018. Geometric Nontermination Arguments. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 266–283. https://doi.org/10.1007/978-3-319-89963-3_16

Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific equivalence checking. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France, France). ACM, 441–451. https://doi.org/10.1145/3238147.3238178

Aditya V. Nori and Rahul Sharma. 2013. Termination Proofs from Tests. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/2491411.2491413

Fabrizio Pastore, Leonardo Mariani, Antti E. J. Hyvärinen, Grigory Fedyukovich, Natasha Sharygina, Stephan Sehestedt, and Ali Muhammad. 2014. Verification-Aided Regression Testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2610384.2610387

Andreas Podelski and Andrey Rybalchenko. 2004. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*, Bernhard Steffen and Giorgio Levi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 239–251. https://doi.org/10.1007/978-3-540-24622-0_20

A. Podelski and A. Rybalchenko. 2004. Transition invariants. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.* (Turku, Finland, Finland). IEEE, 32–41. https://doi.org/10.1109/LICS.2004.1319598

Andreas Podelski and Andrey Rybalchenko. 2005. Transition Predicate Abstraction and Fair Termination. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 132–144. https://doi.org/10.1145/1040305.1040317

Andreas Podelski and Andrey Rybalchenko. 2011. Transition Invariants and Transition Predicate Abstraction for Program Termination. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–10. https://doi.org/10.1007/978-3-642-19835-9_2

Bat-Chen Rothenberg, Daniel Dietsch, and Matthias Heizmann. 2018. Incremental Verification Using Trace Abstraction. In *Static Analysis*, Andreas Podelski (Ed.). Springer International Publishing, Cham, 364–382. https://doi.org/10.1007/978-3-319-99725-4_22

Neha Rungta, Suzette Person, and Joshua Branchaud. 2012. A change impact analysis to characterize evolving program behaviors. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (Trento, Italy). IEEE, 109–118. https://doi.org/10.1109/ICSM.2012.6405261

Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2012. Incremental upgrade checking by means of interpolation-based function summaries. In *Formal Methods in Computer-Aided Design (FMCAD), 2012* (Cambridge, UK). IEEE, 114–121.

Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann. 2017. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning* 58, 1 (2017), 33–65. https://doi.org/10.1007/s10817-016-9389-x

Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-Driven Analysis of Semantic Difference for Program Versions. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 405–427. https://doi.org/10.1007/978-3-319-66706-5_20

Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. 2016. Synthesizing Ranking Functions from Bits and Pieces. In *Tools and Algorithms for the Construction and Analysis of Systems*, Marsha Chechik and Jean-François Raskin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–70. https://doi.org/10.1007/978-3-662-49674-9_4

Helga Velroyen and Philipp Rümmer. 2008. Non-termination Checking for Imperative Programs. In *Tests and Proofs*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–170. https://doi.org/10.1007/978-3-540-79124-9_11

Willem Visser, Jaco Geldenhuys, and Matthew B. Dwyer. 2012. Green: Reducing, Reusing and Recycling Constraints in Program Analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 58, 11 pages. https://doi.org/10.1145/2393596.2393665

Guowei Yang, Matthew B Dwyer, and Gregg Rothermel. 2009. Regression model checking. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on* (Edmonton, AB, Canada). IEEE, 115–124. https://doi.org/10.1109/ICSM.2009.5306334

Guowei Yang, Sarfraz Khurshid, Suzette Person, and Neha Rungta. 2014. Property Differencing for Incremental Checking. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 1059–1070. https://doi.org/10.1145/2568225.2568319