Strongest Postcondition of Unstructured Programs

Radu Grigore
University College Dublin

Julien Charles
University College Dublin
julien.charles@gmail.com

Fintan Fairmichael
University College Dublin
fintan.fairmichael@ucd.ie

Joseph Kiniry
University College Dublin
joseph.kiniry@ucd.ie

ABSTRACT

To avoid exponential explosion, program verifiers turn the program into a passive form before generating verification conditions. A little known fact is that the passive form makes it easy to use a strongest postcondition calculus to derive the verification condition. In the first part of this paper, the passivation phase is defined precisely enough to allow a study of its algorithmic properties. In the second part, the weakest precondition and strongest postcondition methods are presented in a unified way and then compared empirically.

Categories and Subject Descriptors

F3.1 [**Theory of Computation**]: Logics and Meanings of Programs—*Mechanical verification*

Keywords

verification condition generation, algorithms, efficiency

1. INTRODUCTION

Program verifiers are not widely used because annotations are hard to write, but also because verifiers are perceived as slow compared to other tools that developers use regularly, such as compilers. This paper analyzes from the point of view of efficiency some of the problems a program verifier must solve.

Flanagan and Saxe [7] noticed that a verification condition (VC) built from a passive program is much smaller than one built from a program with assignments, so they gave an algorithm that obtains a passive form and they implemented it in ESC/Java [5, 4]. Barnett and Leino [2] informally describe an improved algorithm that is implemented in Spec$^\sharp$ [3, 2]. Surprisingly, neither give a formal definition of what a passive form is.

Once the program is passive the actual VC generation is performed, using either a weakest precondition or a strongest postcondition calculus. The VC is then sent to a theorem prover like Z3 [18]. To our knowledge, no empirical study relates the chosen calculus to proving time. Contributions:

- a precise definition for *passive form* that suggests two notions of optimality (Section 3)
- a fast algorithm for deriving one optimum (Section 3.1) and evidence that there is no fast algorithm for deriving the other optimum (Section 3.2); an implementation of the fast algorithm in FreeBoogie [8] and an experimental comparison with the algorithm of Flanagan and Saxe [7] (Section 3.3)
- a unified theoretical treatment of the weakest precondition and of the strongest postcondition method that makes it easy to compare them (Section 4); an implementation of both methods in FreeBoogie (Section 4.1) and an experimental comparison on the effect they have on the theorem prover time (Section 4.2)

program →	statement$^+$
statement →	*label* **:** command (goto | **return**) **;**
goto →	(**goto** *label* (**,** **goto** *label*)*
command →	assignment | assumption | assertion
assignment →	*id* **:=** *expression* **;**
assumption →	**assume** *expression* **;**
assertion →	**assert** *expression* **;**

Figure 1: A subset of Boogie.

2. BACKGROUND

Program verifiers, like ESC/Java and Spec$^\sharp$, check if code and annotations agree. The code and annotations undergo a sequence of transformations into simpler and simpler languages until eventually a first order logic (FOL) formula is obtained, which is valid ideally if and only if there are no disagreements in the high-level code. A theorem prover then tries to establish whether the formula is indeed a theorem or not.

This paper discusses the last two of these transformations, getting rid of assignments and building a FOL formula. The input of the first of these two phases is given in a subset of Boogie [16] that appears in Fig. 1. An *expression* is a FOL formula. The **goto** is nondeterministic. Control-flow cycles are removed in a previous phase, which is not discussed in this paper.

In this paper we will be sloppy and identify programs with their flowgraphs. A flowgraph $P = (V, E, n_0)$ has a set of nodes V, each of them being a command, a set of edges E,

```
a: x := fresh; goto b, c;
b: assume ¬even(x); goto d;
c: assume even(x); goto e;
d: x := x + 1; goto e;
e: assert even(x);
```

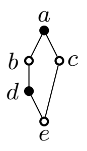

Figure 2: An example of a Boogie program

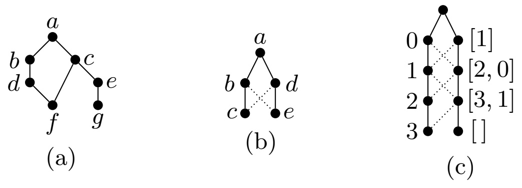

Figure 3: Various interesting special cases

each of them representing a **goto**, and an initial node n_0, which is the command in the first statement. All flowgraphs are acyclic and all nodes are reachable from the initial node. We say that a program is *structured* when its flowgraph is series–parallel. We say that a node is a *write node* with respect to variable x if it is an assignment whose left-hand side is variable x. We say that a node is a *read node* with respect to variable x if variable x appears in its *expression*. Most of the time the variable is clear from the context, so we omit the phrase "with respect to variable x." We say that a node is a *copy node* if it is an assignment with the form $x_i := x_j$. See Table 1 for other notations.

As a reminder, a graph is a *(two terminal) series–parallel* graph [20, 10] if it can be obtained from one edge between two nodes by operations of subdividing and doubling edges. Note that programs composed using sequential composition and nondeterministic choice have series–parallel flowgraphs.

The running example in Fig. 2 illustrates the basic notations used throughout the paper. The Boogie code corresponds to the high-level statement "**if** $(\neg even(x))$ $++x$;". The flowgraph tells us that nodes a and d write to variable x and may also read it, while nodes b, c, and e may only read it. The special cases in Fig. 3 are used to illustrate various points later on. Figure 3a shows a flowgraph whose nodes all write variable x and may also read it. Figure 3b shows a flowgraph with 7 nodes: The nodes a, b, c, d, and e all write to variable x and they may also read it; there are also two unnamed nodes that read the variable x, one has nodes b and e as predecessors and the other has nodes c and d as predecessors. The use of dotted edges to represent unnamed nodes may seem confusing at first, but imagine how Fig. 3c, which has 14 nodes, would look without this convention. The numbers on the left in Fig. 3c identify nodes in the top–down order; the lists on the right give the dotted neighbors and each list is decreasing.

3. OPTIMAL PASSIVE FORM

The main purpose of this section is to give a precise formulation for the problem of finding a good passive form.

Example 1. A passive form of the program in Fig. 2 appears in Fig. 4. It is obtained by introducing *versions* 1 and 2 of the variable x and by inserting the *copy command* f.

Example 1 contains only one variable and this is the case we shall analyze in detail. Multiple variables do not in-

```
a: x₁ := fresh; goto b, c;
b: assume ¬even(x₁); goto d;
c: assume even(x₁); goto f;
d: x₂ := x₁ + 1; goto e;
e: assert even(x₁);
f: x₂ := x₁; goto e;
```

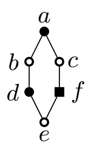

Figure 4: A passive form for the program in Fig. 2

troduce any new complications: A program can be made passive with respect to each of its variables in turn, while considering the others to be constants.

Definition 1. A program is *passive* when on all execution paths each variable is written to at most once.

Remark 1. Obtaining a passive form is similar to automatically deriving a functional equivalent of a loop-less imperative program. The remaining assignments, which are eventually transformed into assumptions, can be seen as **let** bindings.

The passive form of a program P' is an equivalent program P that is passive. If there are execution paths in program P' that write to variable x multiple times then those writes must be changed to write to distinct variables in program P. We denote the variables of program P by x_i where i is some integer and say that x_i is version i (of variable x). To maintain the semantics, each read from variable x must be replaced by a read from the latest written version. As Example 1 illustrates, it is necessary sometimes to alter the structure of the program, and in current approaches [7, 2] this is always done by inserting copy commands of the form $x_i := x_j$. The following definition makes these notions precise.

Definition 2. A passive program $P = (V, E, n_0)$ is a *passive form* of the program $P' = (V', E', n'_0)$ when there exists a mapping $c : V' \to V$, a write-version function $w : V \to \mathbb{N}$, and a read-version function $r : V \to \mathbb{N}$ such that

- command structure is preserved: the command $c(n)$ is obtained from command n by replacing each occurrence of variable x by some variable x_i
- the new nodes are copy commands: all commands in $C = V - c(V')$ have the form $x_i := x_j$
- the flow structure is preserved: there is an edge $m \to n$ in program P' if and only if there is a path $c(m) \rightsquigarrow_C c(n)$ in program P
- the initial node is preserved: $c(n'_0) = n_0$
- writes and reads are confined: each command n in program P may only read version $r(n)$ and may only write version $w(n)$
- the read version is always the latest written version: $w(m) = r(n)$ for all edges $m \to n$ in program P

The requirement that program P is passive can be expressed as a constraint on the write-version function w.

Table 1: Notations used throughout the paper

notation	description
$\top, \bot$	true, false, respectively
$m \to n$	directed edge from node m to node n
$m \rightsquigarrow_C n$	path from m to n whose intermediary nodes are from set C
●, ○, ■, □	read&write, read-only, copy, and non flowgraph node, respectively
m ●····● n	read node with incoming edges from the write nodes m and n
$[\psi]$	1 if ψ is true, 0 otherwise

PROPOSITION 1. *If the write-version function w is a witness that program P is a passive form then the existence of a path $m \rightsquigarrow n$ where both m and n are write nodes implies that $w(m) \neq w(n)$.*

Proposition 1 suggests that a "passive form" may be called more explicitelly "distinct-version passive form". Definition 2 is very general. The passive forms obtained by previous approaches [7, 2] all satisfy a stronger definition that corresponds to the intuition that versions increase in time.

Definition 3. An *increasing-version passive form* is a passive form such that there is a witness write-version function w with the property that $w(m) < w(n)$ whenever there is a path $m \rightsquigarrow n$ from a write node m to another write node n.

Programs have multiple passive forms, some better than others. There are two obvious notions of optimality.

Definition 4. A passive form is *version-optimal* when it uses as few variable versions as possible. A passive form is *copy-optimal* when it uses as few copy commands as possible.

Remark 2. An increasing-version passive form that is both version-optimal and copy-optimal does not always exist (see Fig. 3a). Also, if the increasing-version restriction is dropped it is sometimes possible to obtain passive forms with fewer copy nodes (see Fig. 3b).

3.1 The Version-Optimal Passive Form

This section presents a simple algorithm for obtaining a version-optimal passive form. Since it is simpler than the algorithm of Barnett and Leino [2] and guarantees some form of optimality (unlike the algorithm of Flanagan and Saxe [7]) it should be the algorithm of choice for implementations.

LEMMA 1. *The number of versions used by a passive form program P is greater or equal to the number of write nodes on any execution path in the original program P'.*

Consider now the following labeling of the original flowgraph.

$$r'(n) = \max_{m \to n} w'(m) \tag{1}$$

$$w'(n) = r'(n) + [\text{the node } n \text{ writes the variable } x] \tag{2}$$

These values can be used to derive a version-optimal passive form. The maximum value k taken by the function w' corresponds to the number of writes on some execution path and therefore to the lower bound given by Lemma 1. If we take $w(c(n)) = w'(n)$ then the passive form will write versions $1, \ldots, k$. Similarly, if we take $r(c(n)) = r'(n)$ then the passive form may read versions $0, 1, \ldots, k$. Version 0 is never written. If it is read then it provides a nondeterministic value of a certain type. Instead of having a version 0 for each variable x we can introduce one global uninitialized variable. This leads to a passive form that uses only k versions and is therefore optimal.

Whenever the original flowgraph has an edge $m \to n$ such that $w'(m) = r'(n)$ we keep an edge $c(m) \to c(n)$ in the passive form. Whenever the original flowgraph has edges $m \to n$ such that $w'(m) \neq r'(n)$ we introduce a copy node p in the passive form. We ensure that the passive form has paths $c(m_i) \rightsquigarrow c(n)$ by adding edges $c(m) \to p$ and $p \to c(n)$. We take $r(p) = w'(m)$ and $w(p) = r'(n)$.

We have therefore proved the following theorem.

THEOREM 1. *The functions r' and w' from (1) and (2) determine a version-optimal passive form, which contains $O(|E|)$ copy commands.*

The proof of Theorem 1 can be turned into a program that mutates the flowgraph into a passive form. The values $r'(n)$ and $w'(n)$ are computed by a pair of mutually recursive memoized functions that follow (1) and (2). (See Fig. 5.)

THEOREM 2. *The algorithm in Fig 5 constructs a version-optimal passive form in $\Theta(|V|+|E|)$ time and $\Theta(|V|)$ space. The number of copy commands is $O(|E|)$.*

Remark 3. It is easy to slightly decrease the number of copy commands inserted by the algorithm in Fig. 5. A possibility is to introduce only one copy command for each family of edges $m_1 \to n, \cdots, m_k \to n$ that have $w'(m_i) = w_m \neq r'(n)$ for $0 \leq i < k$ and some constant w_m.

3.2 The Copy-Optimal Passive Form

We saw that finding a version-optimal passive form is rather easy. In contrast, finding a copy-optimal passive form seems rather hard. This is more evidence that the algorithm in Section. 3.1 should be the algorithm of choice in practice.

THEOREM 3. *The problem of finding a copy-optimal passive form among the increasing-version passive forms is NP-hard.*

PROOF. We will prove that if we can find a copy-optimal passive form efficiently then we can also find a maximum independent node set efficiently. As a reminder, a set of nodes is independent when its nodes are pairwise non-adjacent. Finding such a set with maximum cardinality is the same as finding a maximum clique in the complement graph, and

```
READ($n$)
    ▷ memoized (cache results)
1   $r := 0$
2   for $m \in$ PREDECESSORS($n$)
3       do $r := \max(r, \text{WRITE}(m))$
4   return $r$

WRITE($n$)
    ▷ memoized (cache results)
1   $r :=$ READ($n$)
2   if IS-WRITE($n$)
3       do $r := r + 1$
4   return $r$

PASSIVATE($n$)
1   for $n \in V$
2       do for reads: $x \to x_{\text{READ}(n)}$
3          for writes: $x \to x_{\text{WRITE}(n)}$
4   for $(m \to n) \in E$
5       do if WRITE($m$) $\neq$ READ($n$)
6           then $p := (x_{\text{READ}(n)} := x_{\text{WRITE}(m)})$
7                replace $m \to n$ by $m \to p$ and $p \to n$
```

Figure 5: Algorithm for finding a version-optimal passive form

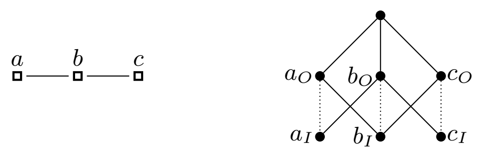

Figure 6: Reduction from Maximum Independent Node Set.

the latter is known to be NP-hard since the dawn of NP-completeness [14].

Figure 6 illustrates the transformation. Each node n in the original graph corresponds to the write nodes n_I, n_O, the read node n_R, and the edges $n_I \to n_R$ and $n_O \to n_R$. (In Fig. 6 the node n_R is represented by a dotted line joining n_I and n_O.) Each non-flowgraph edge $m — n$ corresponds to the edges $m_O \to n_I$ and $n_O \to m_I$. Consider an arbitrary write-function w on the constructed graph. The number of necessary and sufficient copy operations is $\sum_n [w(n_I) \neq w(n_O)]$. Therefore, finding a copy-optimal passive form is equivalent to finding a function w that minimizes the previous sum. This suggests how to transform a passive form into an independent set: Include the original node n in the set when $w(n_I) = w(n_O)$.

A node set constructed this way is always independent because whenever there was an original edge $m — n$ we have $w(m_O) < w(n_I)$ and $w(n_O) < w(m_I)$, which together imply that at least one of $w(m_O) \neq w(m_I)$ and $w(n_I) \neq w(n_O)$ must be true. Furthermore, if there is no non-flowgraph edge $m — n$, then there is no path between the corresponding nodes and hence no restriction on the write-version function. □

CONJECTURE 1. *The problem of finding a copy-optimal passive form is NP-hard.*

This conjecture is supported by the observation that for flowgraphs in the family illustrated by Fig. 3c it is harder to find a copy-optimal passive form if we restrict the search to increasing-version passive forms. The flowgraphs consist of write nodes aligned in two chains and read nodes that have one parent from each chain. In the rest of this section N stands for the number of write nodes and M stands for the number of read nodes in such a graph. The task is to select as many dotted edges (read nodes) as possible that can simultaneously have the same version at their endpoints. In the increasing-version case two dotted edges can be selected simultaneously if they don't intersect; in the distinct-version case two dotted edges can be selected if they don't intersect at one endpoint.

The increasing-version case can be solved, as Fig. 3c suggests, by sorting the adjacency lists in decreasing order, concatenating them, and finding a *longest increasing subsequence*. The best known algorithm for the last step [13] works in $O(M \lg\lg M)$ time. The distinct-version case is exactly the problem of finding a *maximum bipartite matching*, for which the best known algorithms [12, 11] work in $O(\min(M\sqrt{N}, N^{2.38}))$ time. Therefore, the best algorithms known to solve this particular family of instances are faster for the increasing-version case than for the distinct-version case.

3.3 Experiments

Figure 7 compares the results of the passivation algorithm given by Flanagan and Saxe algorithm with the version-optimal passive form (Section 3.1). If for N tests the algorithm of Flanagan and Saxe generated i versions and the FreeBoogie algorithm generated j versions, then this is represented by a circle centered at (i, j) with a radius proportional to $\lg N$. The experiments use 84% of the implementations in the Boogie benchmark [17], because we identified the others as unstructured [20]. The Flanagan–Saxe algorithm is defined on the structure of the ESC/Java intermediate language. This structure can be reconstructed as a side-effect of recognizing series–parallel flowgraphs. The ESC/Java intermediate language may occasionally lead to flowgraphs that are not series–parallel, because it uses exceptions as a control flow mechanism.

For 70% of the variables in the Boogie benchmark both algorithms say that one version is enough. For the other 30% variables the Flanagan–Saxe algorithm introduces on average 46% more versions than needed.

It is interesting to note that for randomly generated flowgraphs the difference between the two algorithms is much bigger: Both algorithms say that only one version is needed in less that 1% of situations, while for the others the algorithm of Flanagan and Saxe introduces on average 160% more versions than needed.

Full experimental results are available on the Internet [1].

4. WEAKEST PRECONDITION VERSUS STRONGEST POSTCONDITION

This section uses Hoare triples to define correctness of a program and proceeds by presenting two methods for deriving annotations, based on weakest preconditions and, respectively, strongest postconditions. The two methods lead to equivalent but structurally different VCs.

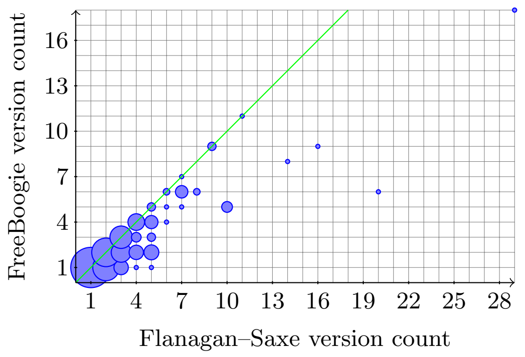

Figure 7: Version count experimental comparison

After the passive form is obtained all assignments $x := e$ are transformed into assumptions of the form **assume** $x{=}e$. Now, a command is either **assert** ψ or **assume** ψ, where ψ is some FOL formula. The semantics of these commands can be defined using Hoare triples as follows.

$$\frac{(\alpha \wedge \psi) \Rightarrow \beta}{\{\alpha\}\ \textbf{assume}\ \psi\ \{\beta\}} \qquad \frac{\alpha \Rightarrow (\psi \wedge \beta)}{\{\alpha\}\ \textbf{assert}\ \psi\ \{\beta\}} \tag{3}$$

This tells us how to compute the weakest precondition given the postcondition and how to compute the strongest postcondition given the precondition.

$$\text{wp}\ (\textbf{assume}\ \psi)\ \beta = \psi \Rightarrow \beta \tag{4}$$
$$\text{wp}\ (\textbf{assert}\ \psi)\ \beta = \psi \wedge \beta \tag{5}$$
$$\text{sp}\ (\_\ \psi)\ \alpha = \alpha \wedge \psi \tag{6}$$

Note that the simpler form of the sp predicate transformer comes at a price. The triple $\{\text{wp}\ n\ \beta\}\ n\ \{\beta\}$ holds for all commands n, while the triple

$$\{\alpha\}\ \textbf{assert}\ \psi\ \{\text{sp}\ (\textbf{assert}\ \psi)\ \alpha\}$$

leaves us the proof obligation $\alpha \Rightarrow \psi$. An edge $m \rightarrow n$ from the node m to the node n in the flowgraph imposes the proof obligation $\beta_m \Rightarrow \alpha_n$, where β_m is the postcondition of the node m and α_n is the precondition of the node n.

Definition 5. A program is *correct* when it is possible to attach to each command n a precondition α_n and a postcondition β_n such that

- the precondition α_0 of the initial node is valid,
- $\{\alpha_n\}\ n\ \{\beta_n\}$ is valid, for all commands n, and
- $\beta_m \Rightarrow \alpha_n$ is valid, for all edges $m \rightarrow n$ in the flowgraph.

When the wp predicate transformer is used, the preconditions and the postconditions are computed as follows:

$$\beta_m = \bigwedge_{m \rightarrow n} \alpha_n \tag{7}$$
$$\alpha_m = \text{wp}\ m\ \beta_m \tag{8}$$

These equations make sure that proof obligations imposed by commands and proof obligations imposed by flowgraph edges are valid. It remains to be checked whether the precondition of the initial node is valid. Hence, the VC is simply $vc_{\text{sp}} = \alpha_0$.

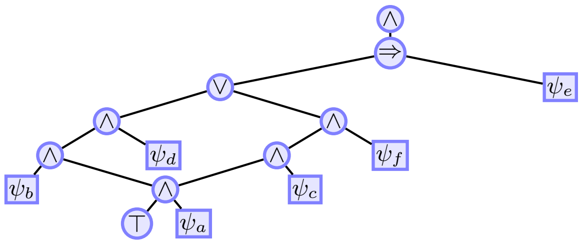

Figure 8: Data structure for the VC in (13)

When the sp predicate transformer is used, the preconditions and the postconditions are computed as follows:

$$\alpha_n = \begin{cases} \top & \text{if the node } n \text{ is initial} \\ \bigvee_{m \rightarrow n} \beta_m & \text{otherwise} \end{cases} \tag{9}$$
$$\beta_n = \text{sp}\ n\ \alpha_n \tag{10}$$

These equations make sure that α_0, proof obligations imposed by flowgraph edges, and proof obligations imposed by assumptions are valid. Proof obligations imposed by assertions remain to be checked. Hence, the VC is

$$vc_{\text{sp}} = \bigwedge_{\substack{n \text{ is an} \\ \text{assertion}}} (\alpha_n \Rightarrow \psi_n) \tag{11}$$

Remark 4. Both methods for obtaining a VC are complete, meaning that if the program is correct then the VC is valid. The proof is standard.

Example 2. For the code in Fig. 4 the weakest precondition method and the strongest postcondition method yield equivalent but different VCs:

$$\begin{aligned} vc_{\text{wp}} =& (x_0 = \mathit{fresh}) \Rightarrow \\ & \big((\neg \mathit{even}(x_0) \Rightarrow (x_1 = x_0 + 1) \Rightarrow (\mathit{even}(x_1) \wedge \top)) \\ & \wedge (\mathit{even}(x_0) \Rightarrow (x_1 = x_0) \Rightarrow (\mathit{even}(x_1) \wedge \top))\big) \end{aligned} \tag{12}$$

$$\begin{aligned} vc_{\text{sp}} =& \big((\top \wedge (x_0 = \mathit{fresh}) \wedge \neg \mathit{even}(x_0) \wedge (x_1 = x_0 + 1)) \\ & \vee (\top \wedge (x_0 = \mathit{fresh}) \wedge \mathit{even}(x_0) \wedge (x_1 = x_0))\big) \\ & \Rightarrow \mathit{even}(x_1) \end{aligned} \tag{13}$$

4.1 Implementation and Verification Condition Size

The implementation uses three functions, Vc, Pre, and Post, that follow (11), (9), and (10), respectively. To avoid unnecessary work, the last two are memoized. (See Fig. 9.)

Example 3. The data structure built by such an implementation for the VC given in (13) appears in Fig. 8. Note that $\beta_a = \top \wedge \psi_a$ is shared because of memoization.

```
PRE($n$)
    ▷ memoized (cache results)
1   $p := \emptyset$
2   for $m \in$ PARENTS($n$)
3         do $p := p \cup \{$POST($m$)$\}$
4   return OR($p$)

POST($n$)
    ▷ memoized (cache results)
1   return AND(PRE($n$), FORMULA($n$))

VC()
1   $r := \emptyset$
2   for $n \in V$
3         do if IS-ASSERTION($n$)
4                 then $r := r \cup \{$IMPLIES(PRE($n$), $\psi_n$)$\}$
5   return AND($r$)
```

Figure 9: VC computation via SP

THEOREM 4. *The algorithm inf Fig. 9 computes the VC in $O(|V| + |E|)$ time. If the program contains no assertions then the lower bound $\Omega(|V|)$ is attained.*

PROOF. Because of memoization the functions PRE and POST are called at most once for each node. The function PRE analyzes all the incoming edges of the node it was called for, so in the worst case it looks once at each edge in the flowgraph. If there are no assertions then the runtime is dominated by the loop in VC that checks there are no assertions. □

THEOREM 5. *The algorithm in Fig. 9 computes a VC with size $O(|V|+|E|+|\Psi|)$, where $|\Psi|$ is the space needed to represent all the expressions that appear in the program. If the program contains no assertions then the lower bound $\Omega(1)$ is attained.*

PROOF. Each execution of the function PRE creates one new node that contains as many links to children as there are predecessors in the flowgraph. Each execution of the function POST creates one new node with two children, one of which is an expression from the program. If there are no assertion then $vc_{\text{sp}} = \bot$. □

Remark 5. Results similar to Theorem 4 and Theorem 5 hold for the analogous implementation of the weakest precondition method.

4.2 Experimental Comparison

Figure 10 shows the ratios between the proving times required for weakest precondition and strongest postcondition. The tests were ran using the Z3 v1.2 prover on a single core of a dual-core AMD Opteron 2218 (2 X 2.6GHz) with 16GiB of DDR2-667 RAM. Both methods perform similarly, as can be seen by the relative symmetry of the graph. Weakest precondition does mildly better however, as evidenced by the slight shift to the right. It is interesting to note that in over 40% of the test cases the time ratio between the worse method and the better method is > 2. These are the cases outside of the shaded area.

Full experimental results are available on the Internet [1].

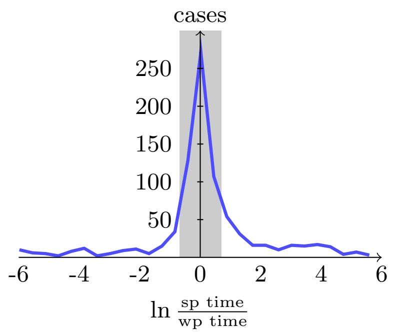

Figure 10: Proving time experimental comparison

5. RELATED WORK

The passive form used for program verification resembles the dynamic single assignment (DSA) form used in the compiler community to facilitate optimizations. Two main concerns in deriving a DSA form are how to deal with arrays and how to deal with loops. In the program verification setting arrays are naturally not a problem since the *update* and *select* operations are axiomatized anyway for the purpose of verification; in the program verification setting loops are naturally not a problem since they are replaced by invariants anyway for the purpose of verification. Recent work on computing the DSA form [21] handles these two problems and identifies the trick of inserting copy operations (used earlier by Flanagan and Saxe [7]) as the key to obtaining a scalable algorithm for deriving the DSA form. There are no theoretical guarantees on achieving a minimum number of versions or a minimum number of copy operations. The number of copy operation is brought down by a second, optimizing step, the result of which is analyzed empirically. To the best of our knowledge the problem of obtaining a DSA form is also not formally defined. (Informal definitions give more leeway for out-of-the-box thinking that can lead to interesting new results, while formal definitions facilitate a more precise analysis of what is going on. In our opinion, the latter is important when studying program verifiers, and that is why we give Definition 2.)

ESC/Java can use both the strongest postcondition [7] and the weakest precondition method [15], but not for arbitrary acyclic flowgraphs. Note that [7] describes a method analogous to our strongest postcondition method but only uses the term "outcome predicate". Boogie uses only the weakest precondition method and can treat unstructured programs [2].

Strongest postcondition was discussed before in the context of a simple language similar to ours in relation to predicate abstraction [6] (but only for structured programs) and in relation to proof reuse [9] (but not shown sound). It was also formalized in the context of structured Java bytecode [19].

6. CONCLUSIONS AND PROBLEMS

The precise definition of *passive form* led to a simple, fast, and slightly better algorithm for passivation. The weakest precondition method yields slightly faster prover response times.

The following problems remain open:

- Is the problem of finding a copy-optimal passive form NP-hard?
- Is it always possible to find a passive form that is both version-optimal and copy-optimal?
- Find an approximation algorithm for the problem of finding a copy-optimal passive form.
- Quickly decide based on the structure of the program if the weakest precondition method or the strongest post-condition method should be used to minimize prover time.
- Exploit the structure of vc_{sp} to better handle big VCs.

Acknowledgments.

The authors thank Mikoláš Janota for discussions, Mike Barnett and Rustan Leino for promptly replying to queries about [2], Joseph Cheriyan for pointing out reference [11], and the reviewers for contributing to the quality of the presentation. This work is funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. The article contains only the authors' views and the Community is not liable for any use that may be made of the information therein. This work is partially supported by an EMBARK Scholarship from the Irish Research Council in Science, Engineering and Technology.

7. REFERENCES

[1] Experimental results. http://groups.google.com/group/freeboogie/web/spup-experimental-data.

[2] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *Workshop on Program Analysis for Software Tools and Engineering*, pages 82–87, 2005.

[3] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^{\#}$ programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 3362, 2004.

[4] D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, 3362:108–128, 2005.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, 2002.

[6] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 191–202, New York, NY, USA, 2002. ACM.

[7] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *Symposium on Principles of Programming Languages*, pages 193–205, 2001.

[8] R. Grigore. The web home of FreeBoogie. http://kind.ucd.ie/products/opensource/FreeBoogie/.

[9] R. Grigore and M. Moskal. Edit and verify. In S. Ranise, editor, *Proceedings of the 6th International Workshop on First-Order Theorem Proving*, pages 101–113. University of Liverpool, Sept. 2007.

[10] J. L. Gross and J. Yellen. *Handbook of graph theory.* CRC Press, 2004.

[11] N. J. A. Harvey. Algebraic structures and algorithms for matching and matroid problems. In *Foundations of Computer Science*, pages 531–542, 2006.

[12] J. E. Hopcroft and R. M. Karp. An $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225, 1973.

[13] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[14] R. M. Karp. Reducibility among combinatorial problems. *Complexity of computer computations*, pages 85–104, 1972.

[15] K. R. M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6), 2005.

[16] K. R. M. Leino. This is Boogie 2. 2008.

[17] Microsoft Research. The Boogie benchmark. http://research.microsoft.com/en-us/projects/specsharp/.

[18] L. D. Moura and N. Bjorner. Z3: An efficient SMT solver. 4963:337, 2008.

[19] M. Pavlova. *Java Bytecode verification and its applications.* PhD thesis, PhD thesis, University of Nice Sophia-Antipolis, 2007.

[20] J. Valdes, R. E. Tarjan, and E. L. Lawler. The recognition of Series–Parallel digraphs. *ACM symposium on Theory of computing*, pages 1–12, 1979.

[21] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A practical dynamic single assignment transformation. *ACM Transactions on Design and Automation of Electronic Systems*, 12(4), 2007.