Typestate Verification: Abstraction Techniques and Complexity Results

J. Field[1], D. Goyal[1], G. Ramalingam[1], and E. Yahav[2]

[1] IBM T.J. Watson Research Center {jfield,dgoyal,rama}@watson.ibm.com
[2] Tel Aviv University yahave@post.tau.ac.il

Abstract. We consider the problem of *typestate verification* for *shallow* programs; i.e., programs where pointers from program variables to heap-allocated objects are allowed, but where heap-allocated objects may not themselves contain pointers. We prove a number of results relating the complexity of verification to the nature of the finite state machine used to specify the property. Some properties are shown to be intractable, but others which appear to be quite similar admit polynomial-time verification algorithms. While there has been much progress on many aspects of automated program verification, we are not aware of any previous work relating the difficulty of typestate verification to properties of the finite state automaton. Our results serve to provide insight into the inherent complexity of important classes of verification problems. In addition, the program abstractions used for the polynomial-time verification algorithms may be of independent interest.

1 Introduction

The desire for more reliable software has led to increasing interest in extended static checking: statically verifying whether a program satisfies certain desirable properties. A technique that has received particular attention is that of finite state or *typestate* verification (e.g., see [24, 23, 19, 5, 7, 3, 8, 12, 11, 16, 1]). In this model, objects of a given type may exist in one of several states; the operations permitted on an object depend on the state of the object, and the operations may potentially alter the state of the object. The goal of typestate verification is to statically determine if the execution of a given program may cause an operation to be performed on an object in a state where the operation is not permitted.

Typestate verification can thus be used to check that objects satisfy certain kinds of temporal properties; for example, that an object is not used before it is initialized, or that a file is not used after it is closed. The properties we will consider in this paper will be specified using regular expressions or finite state automata which define the set of *valid* sequences of operations that can be performed on a single object.

Our goal in this paper is to develop an initial understanding of how the difficulty of performing typestate verification relates to the *nature of the property being verified*. While there has been much progress on many aspects of automated program verification, we are not aware of any previous work relating the difficulty of typestate verification to properties of the finite state automaton. This work is part of a broader effort to develop efficient program verification techniques that are tailored to the property being verified [21].

Typestate Verification and Shallow Programs

Complexity results of the kind presented in this paper require that we talk about analyses with a specific degree of precision (i.e., analyses that are *complete* with respect to a specific abstraction). We will use the term *verification* to mean verification that is *precise* modulo the widely used assumption that all paths in the program are feasible. Given a property, a path in a program is said to be an *error path*, if execution along that path would cause an invalid sequence of operations to be performed on at least one *object*, and the goal of verification is to determine if a given program has any error path.

Typestate verification can be done in polynomial time in the absence of aliasing. We can show that in the presence of two or more levels of pointers, typestate verification is PSPACE-hard

and that in the presence of recursive data structures typestate verification is undecidable. In this paper, we will be concerned with typestate verification of *shallow* programs. We use the term *shallow program* to denote a procedure-free program where all variables are pointers to objects of a given type T, and whose statements are either allocations (creation of a new object of type T), copy assignments (copying the value of a variable to another), or invocations of an operation on a variable. Note that shallow programs allow multiple pointers to the same allocated object, but allocated objects may not themselves contain pointers (i.e., pointers in shallow programs are *single-level* [18]). (We focus on programs with a single type because any typestate property concerns objects of a specific type T. Objects of other types are irrelevant to verification of the given property. Our results apply even to programs that contain other complex or recursive data structures of other types, as long as they do not contain any pointers to objects of type T. Programs that are shallow with respect to a given type, e.g. `File`, are not uncommon in practice.)

Consider $\texttt{read}^*;\texttt{close}$, the problem of checking that a file is never read after it is closed. The principal difficulty in doing a precise analysis arises from determining how aliasing interacts with operations on objects. Some prior work on typestate verification (e.g. [6]) has employed a two-step approach to the problem, in which an initial phase performs a conservative heap analysis of the program, and a subsequent phase uses the information from the heap analysis to do typestate analysis. In some cases, such an approach can lead to imprecise results, as is illustrated by the program fragments in Figure 1.

One can manually verify that the read statements in both Figures 1(a) and 1(b) are safe. However, consider a two-phase analysis in which the heap analysis is separate from the typestate analysis. In Fig. 1(a), any conservative heap analysis will determine that program variable `z` may be aliased to program variables `x` and `y` at program point `s0`. Furthermore, a typestate analysis would determine that the object being pointed to by `y` could be in a *closed* state at `s0`. Such an analysis would therefore erroneously determine that the read could be performed on a closed file. Similarly, in Fig. 1(b), any conservative heap analysis would determine that objects created at program points `s1` and `s3` could reach the read statement at `s2`. In addition, a typestate analysis would also determine that the objects created at program points `s1` and `s3` could be in a closed state at `s2`. The analysis would, however, not be able to discover that `f` can never point to a closed object at `s2`, and would incorrectly indicate a possible error. In this paper we show that for a certain class of problems (including $\texttt{read}^*;\texttt{close}$), it is possible to formulate a precise polynomial time verification algorithm for shallow programs.

```
x := new (); y := new ();
z := y;
if (?) {
  y.close();
  z := x;
}
s0 : z.read();
```

(a)

```
s1 : f := new ();
while (?) {
  s2 : f.read();
  if (?) {
    f.close();
    s3 : f := new ();
}}
```

(b)

Fig. 1. Program fragments illustrating the effect of aliasing on typestate verification.

Main Results

Every finite state property determines a corresponding verification (decision) problem: Given a shallow program, determine if there exists an error path in the program. We show in this paper that not all finite state properties are equally hard to verify. For instance, ***polynomial time verification is possible for the problem of determining whether a file is read after it is closed*** (we will denote this problem by the regular expression $\texttt{read}^*;\texttt{close}$), ***while verification is PSPACE-Complete for the problem of determining whether a file is read before it is opened*** ($\texttt{open}^+;\texttt{read}$). In

this paper, we will usually use regular expressions to represent properties, where a sequence of operations is considered to be valid if and only if it is a *prefix* of some string matching the regular expression.

The main results established in this paper are as follows (in all cases, we assume that programs are shallow):

- Verification is in P for omission-closed properties: a property is said to be omission-closed if every subsequence of a valid sequence is also a valid sequence. (Example: $\texttt{read}^*;\texttt{close}$.)
- Verification is NP-Complete for acyclic programs (i.e., programs without loops) and PSPACE-complete for arbitrary programs for properties with a repeatable enabling sequence: a property is said to have a repeatable enabling sequence if there is a state where a particular sequence γ of operations is invalid, but $\beta^+\gamma$ is valid. Example: $\texttt{open}^+;\texttt{read}$.
- An integer-valued function f is said to be a bound on the shortest error path length for a finite state property if every erroneous program of size n is guaranteed to have an error path of length $f(n)$ or less. If PSPACE is not equal to NP, then no polynomial bound exists for the shortest error path length for properties with a repeatable enabling sequence. (In other words, it may not be possible to find short (i.e. of polynomial size) error paths in the worst case.)
- Verification is in P for acyclic programs for almost omission-closed properties: a property is said to be almost omission-closed if there is an integer k such that every subsequence of a valid sequence of length greater than k is also valid. Example: $\texttt{open};\texttt{read}$. Note that any property with only finitely many valid sequences is trivially almost omission-closed.
- Verification is in P for almost omission-closed properties that have a polynomial bound on the shortest error path length.
- A program is said to have a maximum aliasing factor of k if there is no path in the program that will produce an object pointed to by more than k different variables. Arbitrary finite state properties for programs of size n with a maximum aliasing factor of k may be verified in time $O(n^{k+1})$ for programs of size n.

The results above are summarized in Fig. 2 in terms of the properties of regular expressions which define the properties to be verified (the notation used there will be defined in Section 2).

	Omission-Closed	Almost Omission-Closed	Repeatable Enabling Seq	Other
E.g.	$\texttt{read}^*;\texttt{close}$	$\texttt{open};\texttt{read}$	$\texttt{open}^+;\texttt{read}$	$(\texttt{lock};\texttt{unlock})^*$
Defn.	$\forall\alpha\beta\gamma.\mathit{Valid}(\alpha\beta\gamma) \implies \mathit{Valid}(\alpha\gamma)$	$\exists k\forall\alpha\beta\gamma.\lvert\alpha\beta\gamma\rvert \geq k \wedge \mathit{Valid}(\alpha\beta\gamma) \implies \mathit{Valid}(\alpha\gamma)$	$\exists\alpha\beta\gamma.\mathit{Valid}(\alpha\beta^+\gamma)\wedge \neg\mathit{Valid}(\alpha\gamma)$	
Acyclic Pgms	P	P	NP-complete	?
Cyclic Pgms	P	Poly. Error Path $\implies$ P General: ?	PSPACE-complete	?
Bounded Aliasing	P			

Fig. 2. Overview.

The polynomial-time verification results summarized above use program abstractions that may be of independent interest—in particular, they may prove useful as the starting point for developing more general abstractions for non-shallow programs (e.g., in a manner similar to [21]).

Another interesting contribution of this paper is a novel abstract interpretation we use, in Section 5, which is based on *counting* the number of program paths along which a simple property holds true as an indirect way of inferring if a more complex property holds true.

Related Work

There has been a significant resurgence of interest in *partial* verification recently. Many, though not all, of the problems considered in the literature can be expressed as typestate verification problems.

While significant progress has been made in several dimensions, developing verification techniques that are precise enough and scale up to handle industrial-size applications is still a challenge and is the motivation for the complexity results of the form presented in this paper.

One of the challenges in typestate verification is dealing with aliasing. Some approaches avoid the issue: e.g., the original work on typestate verification [24, 23] did not allow any aliasing, and more recent work on typestate verification based on linear types [7] also restrict aliasing severely. Some other approaches (e.g. [6]) do the alias analysis and typestate verification separately: an initial phase performs a conservative alias analysis for the program, and a subsequent phase uses the information from the alias analysis to do typestate verification. This can lead to imprecise results, as illustrated by the examples in Fig. 1.

A second challenge in typestate verification is dealing with infeasible paths in the program. Das et al. [6] present efficient algorithms that are path-sensitive (i.e., eliminate certain infeasible paths from consideration during analysis) but do not track aliasing precisely (as explained above). In contrast, our algorithms track (one-level) aliasing precisely, but are not path-sensitive. It might be worthwhile trying to combine these techniques.

Several recent verification approaches [2, 15], based on *predicate abstraction* [14], avoid imprecision (e.g., due to aliasing or infeasible paths) by iteratively refining the abstractions as necessary, but are fundamentally exponential algorithms. These techniques use symbolic and theorem-proving techniques (during verification) to identify a set P of "relevant" predicates, and then use the powerset lattice $2^{P \to \{true, false\}}$ for abstraction, and then model check the resulting finite state system (and usually iterate with increasingly larger sets of predicates until a satisfactory result is obtained). The worst-case complexity of a single iteration is exponential in the number of predicates. In contrast, for our upper bounds, we describe the set of predicates Q to be used a priori (and do not rely on expensive predicate discovery at verification time). More importantly, our verification algorithms utilize the function-space lattice $Q \to \{false, maybe\}$ for abstraction, with a worst-case complexity that is linear in the number of predicates. Our selection of predicates ensures that there is no loss of precision in using the smaller function-space lattice, i.e., we ensure that our abstraction is *complete* (e.g., see [13]). Our approach yields polynomial time verification algorithms for various problems, while none of the predicate abstraction based techniques mentioned above can guarantee a polynomial time worst-case complexity for the same problems (even though they may perform verification with a smaller number of predicates than our algorithm).

Our lower bound results follow the tradition set by earlier complexity results due to Landi and Ryder [17] and Muth and Debray [18].

2 Terminology and Notation

In this section, we provide some basic definitions that we will use in the rest of the paper.

Definition 1 (Shallow Program). *A* shallow program *is a* `<Stmt>` *defined by the following context-free grammar, where the* `?` *denotes a nondeterministic branch (i.e., an uninterpreted conditional). All variables* `<Var>` *in the language are references to objects of type T. All operations* `<Op>` *in the language are methods supported by type T.*

```
<Stmt> ::= <Var> := <Var> | <Var> := new() | <Var>.<Op>()
         | <Stmt>;<Stmt> | if (?) <Stmt> [ else <Stmt> ]
         | Label: <Stmt> | goto Label
```

We will make the simplifying assumption that in the initial state of the program all program variables point to separate objects (i.e., initialized to non-aliased values), and all objects reside in their initial state, but other initial states can be easily handled. In other respects, the semantics

of shallow programs is completely standard, and we will not formalize it here. We will, however, appeal to the intuitive notion of a *path* ρ through a program P (or P-path): a valid sequence of statements starting at P's entry.

In this paper, we will study safety properties of shallow programs. Although safety properties could be specified via temporal logics (e.g., LTL [4]), we will use finite automata or regular expressions to simplify the presentation. Formally:

Definition 2 (Prefix-Closed Safety Automaton). *A prefix-closed safety property $\mathcal{F}$ is represented by a finite state automaton (FSA) $\mathcal{F} = \langle \Sigma, \mathcal{Q}, \delta, init, \mathcal{Q} \setminus \{err\} \rangle$ where Σ is the automaton alphabet which consists of observable operations, $\mathcal{Q}$ is the set of automaton states, δ is the transition function mapping a state and an operation to a successor state, $init \in \mathcal{Q}$ is a distinguished* initial state*, $err \in \mathcal{Q}$ is a distinguished* error state *for which for every $\sigma \in \Sigma$, $\delta(err, \sigma) = err$, and all states in $\mathcal{Q} \setminus \{err\}$ are accepting states. We say that q' is the successor of a state q on operation* `op` *when $\delta(q, \texttt{op}) = q'$. Given a sequence of operations $\alpha = \texttt{op}_1; \texttt{op}_2; \ldots; \texttt{op}_k$, we write $Valid_{\mathcal{F}}(\alpha)$ or $\alpha \in Valid_{\mathcal{F}}$ when α is accepted by $\mathcal{F}$, and we write $Invalid_{\mathcal{F}}(\alpha)$ when α is not accepted by $\mathcal{F}$.*

For brevity, we will refer to safety properties using a regular expression representing the language accepted by an automaton, rather than specifying the automaton itself. When specifying a safety property using a regular expression, we will adopt the convention that a regular expression α denotes the *prefix closure* of the set of sequences of operations defined by α. For example, when we write `read`*;`close` we also consider ϵ (the empty sequence) and `read` to be valid sequences.

Example 1. Consider the property `read`*;`close` stating that a file may be read an arbitrary number of times before it is closed (and should never be read after it was closed and never be closed twice). The alphabet for this problem consists of two operations $\Sigma = \{\texttt{read}, \texttt{close}\}$. The FSA for this property is shown in Fig. 3.

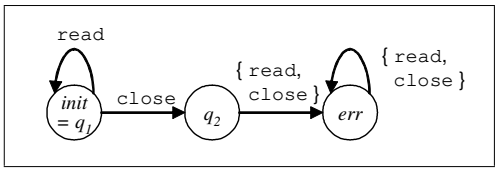

Fig. 3. A finite-state automaton for the property `read`*;`close`.

When verifying a safety property represented by an automaton $\langle \mathcal{Q}, init, err, \Sigma, \delta \rangle$ for a shallow program P, we will assume that each method name used in P is mapped to an element of Σ. Given this convention, we will use names of operations in Σ and methods in P interchangeably, i.e., we will say that a statement of the form `x.op()` invokes an operation $\texttt{op} \in \Sigma$. We can then relate method invocations to sequences of operations in Σ as follows:

Definition 3 (Operation Sequences for Objects). *Given a P-path ρ, $\mathcal{U}(\rho)$ denotes the set of object instances created during this execution, and for any object $o \in \mathcal{U}(\rho)$, $\rho[o]$ denotes the sequence of operations performed on o during execution of ρ.*

Given the definitions above, we can now formally describe the class of verification problems we wish to solve:

Definition 4 ($\mathrm{SV}_{\mathcal{F}}$). *Given a safety property $\mathcal{F}$, the* shallow verification problem *for $\mathcal{F}$, $\mathrm{SV}_{\mathcal{F}}$, determines for any shallow program P whether there exists a path P-path ρ such that $\rho[o] \in Invalid_{\mathcal{F}}$ for some $o \in \mathcal{U}(\rho)$.*

3 Omission-Closed Properties in Polynomial Time

In this section, we show that *omission-closed* properties can be verified in polynomial time.

Omission-Closed Properties

Informally, a property is omission-closed if the set of all valid sequences of operations is closed with respect to omissions: any sequence obtained by omitting one or more operations from a valid sequence of operations is also valid.

Definition 5. *A property represented by an automaton $\mathcal{F}$ is said to be* omission-closed *when for all sequences $\alpha, \beta, \gamma \in \Sigma^*$, $Valid_{\mathcal{F}}(\alpha\beta\gamma) \Rightarrow Valid_{\mathcal{F}}(\alpha\gamma)$.*

The following theorem presents alternative characterizations of omission-closed properties.

Theorem 1. *Given an automaton $\mathcal{F}$, the following are all equivalent, where all sequences are elements of Σ^*:*

(a) For all sequences α, β, γ, $Valid_{\mathcal{F}}(\alpha\beta\gamma) \Rightarrow Valid_{\mathcal{F}}(\alpha\gamma)$.
(b) If ω_1 is a subsequence of ω_2, then $Valid_{\mathcal{F}}(\omega_2) \Rightarrow Valid_{\mathcal{F}}(\omega_1)$.
(c) There exists a finite set of forbidden subsequences *$\xi_1, \xi_2, \ldots, \xi_k$ such that a sequence α is in $Invalid_{\mathcal{F}}$ iff α contains some ξ_i as a subsequence.*

Proof. The equivalence of (a) and (b) is straightforward. As for, (c), consider the forbidden subsequences ξ_i corresponding to the *acyclic* paths in the automaton $\mathcal{F}$ from the initial state to the error state. (For example, the forbidden subsequences for the automaton in Fig. 3 are ξ_1 = `close; read` and ξ_2 = `close; close`.) The result follows.

Background: Distributive Predicate Abstractions

The analysis we present will utilize a *predicate* abstraction that tracks the values of a set of predicates P (defined on the concrete state). For efficiency reasons, we will utilize an *independent attributes analysis* [20], an analysis that does not maintain the correlation between different predicate values. Specifically, the set of concrete states arising at a program point will be abstracted by a value in $P \rightarrow \{false, maybe\}$. We now summarize the conditions under which an *independent attributes analysis* can be used for a predicate abstraction without losing precision. Given a predicate φ and a statement `St`, we denote by WP$(\texttt{St}, \varphi)$ the weakest precondition of φ with respect to `St` [9].

Definition 6. *Given a finite set of predicates Base, we say that a finite set of predicates $\mathcal{P} = \{P_1, \ldots, P_k\}$ is a* distributive WP-closure *of Base when Base $\subseteq \mathcal{P}$ and for each predicate $P_i \in \mathcal{P}$, and for each statement `St`,* WP$(\texttt{St}, P_i) = P_{j_1} \vee \ldots \vee P_{j_m}$*, such that for all $1 \leq g \leq m, P_{j_g} \in \mathcal{P}$. We also say that the set of predicates $\mathcal{P}$ is* distributively WP-closed*.*

Theorem 2. *Given a distributively WP-closed set of predicates $\mathcal{P}$ for a program* Pgm*, precise analysis (i.e., determining for every program point and every predicate in $\mathcal{P}$ whether there exists a path to the program point causing the predicate to be true) is possible in time $O(|\mathcal{P}||\text{Pgm}|)$.*

Proof. Straightforward. E.g., the problem can be reduced to a reachability problem over a graph of size $O(|\mathcal{P}||Pgm|)$, as in the IFDS framework of [22].

A Polynomial Algorithm

We use a designated predicate *Error* that is *true* in a state if and only if the state contains an object in the error state *err*. We will now show that for omission-closed properties, a distributive WP closure of polynomial size can be constructed for $\{Error\}$. In general, the analysis needs to track aliasing relationships among variables as well as the state of the objects pointed to by the variables. This motivates the following definition of a family of predicates.

Definition 7. *We write $In_\sigma(\mathtt{x})$ to denote the fact that the object pointed to by the variable* x *is in state $\sigma \in \mathcal{Q}$. Given any $S \subseteq \mathcal{Q}$, we use the shorthand $In_S(\mathtt{x}) \triangleq \bigvee_{\sigma \in S} In_\sigma(\mathtt{x})$ to denote that the object pointed to by the variable* x *is in one of the states in S.*

Definition 8. *Let A be a non-empty set of variables (in a given program), $S \subseteq \mathcal{Q}$ a set of states in $\mathcal{F}$. We use the predicate $\langle A, S\rangle$ to mean that all variables in A have the same value (are aliases), and the object referred to by variables in A is in one of the states in S. Formally,*

$$\langle A, S\rangle \triangleq \bigwedge_{\mathtt{x} \in A, \mathtt{y} \in A} (\mathtt{y} = \mathtt{x}) \wedge \bigwedge_{\mathtt{x} \in A} In_S(\mathtt{x})$$

The number of predicates of the form $\langle A, S\rangle$ is exponential in the number of program variables. However, not all predicates of this form are needed for verification of omission-closed properties. The key to obtaining a polynomial time algorithm is to bound the size of the set A, for any predicate $\langle A, S\rangle$ *that we care about*, by a constant. We will do this in two steps. First, we will show that we care about predicate $\langle A, S\rangle$ only for certain $S \subseteq \mathcal{Q}$. Then, we will show that for every S we care about, we care about predicate $\langle A, S\rangle$ for only A of cardinality less than a specific constant.

We first present an algorithm for determining which $S \subseteq \mathcal{Q}$ are relevant for verification. The algorithm shown in Fig. 4 is based on a backward traversal of the finite state automaton. The algorithm constructs a graph $\overleftarrow{\mathcal{F}} = (V_{\overleftarrow{\mathcal{F}}}, E_{\overleftarrow{\mathcal{F}}})$, whose vertices are subsets of $\mathcal{Q}$, and an edge $P \rightarrow S$ denotes that P is a pre-image of S for the transition function δ (see below).

Definition 9. *We use $\overleftarrow{\delta}$ to denote the* reverse transition relation *of $\mathcal{F}$, that is, given a state $q \in \mathcal{Q}$, an operation $a \in \Sigma$, and a set of states $S \subseteq \mathcal{Q}$, $\overleftarrow{\delta}(q, a) \triangleq \{q' \in \mathcal{Q} | \delta(q', a) = q\}$, and $\overleftarrow{\delta}(S, a) \triangleq \bigcup_{q \in S} \overleftarrow{\delta}(q, a)$. For $S_1, S_2 \subseteq \mathcal{Q}$, we say that S_2 is a* pre-image *of S_1 if $\overleftarrow{\delta}(S_1, a) = S_2$ for some $a \in \Sigma$.*

```
V_←F = ∅;  E_←F = ∅;  workSet = {{err}};
while workSet ≠ ∅ {
  select and remove S from workSet;
  for each operation op ∈ Σ {
    P = δ←(S, op);
    if P ∉ V_←F { V_←F = V_←F ∪ {P};  workSet = workSet ∪ {P}; }
    E_←F = E_←F ∪ {P → S};
} }
```

Fig. 4. Backwards exploration of the property automaton.

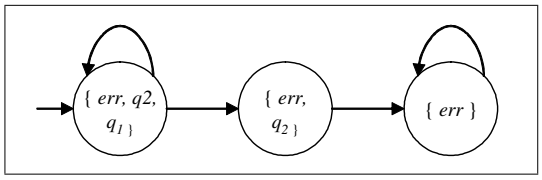

Fig. 5. The graph constructed by backward exploration of the automaton of Fig. 3.

Fig. 5 illustrates the graph constructed by backward exploration of the `read`*;`close` automaton shown in Fig. 3. We now establish a result that will enable us to show that it is sufficient to consider predicates of the form $\langle A, S\rangle$ for $S \in V_{\overleftarrow{\mathcal{F}}}$.

Theorem 3. *If $\mathcal{F}$ represents an omission-closed property, then for any $S \in V_{\overleftarrow{\mathcal{F}}}$, and any operation $a \in \Sigma$, $\overleftarrow{\delta}(S, a) \supseteq S$. Further, the graph $\overleftarrow{\mathcal{F}}$ is acyclic except for self-loops.*

Proof. For any $S \in V_{\overleftarrow{\mathcal{F}}}$ there exists a sequence of operations ξ such that S is the set of all states in which ξ is invalid (by construction). Now, $\overleftarrow{\delta}(S, a)$ is the set of all states in which $a\xi$ is invalid. Since $\mathcal{F}$ is omission-closed, $\overleftarrow{\delta}(S, a) \supseteq S$.

Since any predecessor P of S must be a superset of S, it follows immediately that any cycle in the graph $\overleftarrow{\mathcal{F}}$ must be a self-loop. □

Fig. 6 and Fig. 7 present weakest-precondition equations for predicates of the form $\langle A, S\rangle$ and the special predicate *Error*. From these weakest-precondition equations, we can determine which predicates are relevant for our verification algorithm. The weakest-precondition equations reveal two things. First, they show that it is sufficient if we restrict our attention to predicates of the form $\langle A, S\rangle$ where $S \in V_{\overleftarrow{\mathcal{F}}}$. Second, it shows that a predicate $\langle A, P\rangle$ is relevant only if there is a relevant predicate $\langle B, S\rangle$ where S is a proper successor of P in the graph $\overleftarrow{\mathcal{F}}$ and B has cardinality at least $|A| - 1$. In other words, it shows that we need to only consider predicates of the form $\langle A, P\rangle$ where the cardinality of A is less than or equal to the length of the longest acyclic path from P to $\{err\}$ in $\overleftarrow{\mathcal{F}}$.

`Stmt`	WP(`Stmt`, $\langle A, S\rangle$)	
`x := y`	$\langle A[\mathtt{x} \mapsto \mathtt{y}], S\rangle$	
`x := new ()`	$\langle A, S\rangle$	if $\mathtt{x} \notin A$
	false	if $\mathtt{x} \in A \wedge A \neq \{\mathtt{x}\}$
	true	if $A = \{\mathtt{x}\} \wedge$ *init* $\in S$
	false	if $A = \{\mathtt{x}\} \wedge$ *init* $\notin S$
`x.op()`	$\langle A, S\rangle$	if $\overleftarrow{\delta}(S, \mathtt{op}) = S$
	$\langle A \cup \{x\}, \overleftarrow{\delta}(S, op)\rangle \vee \langle A, S\rangle$	if $\overleftarrow{\delta}(S, op) \supset S$
At program entry	*true*	if $\lvert A\rvert = 1 \wedge$ *init* $\in S$
	false	if $\lvert A\rvert \neq 1 \vee$ *init* $\notin S$

Fig. 6. WP equations for predicates of the form $\langle A, S\rangle$. We denote by $A[x \mapsto y]$ the set obtained by replacing any occurrence of x in A by y.

Definition 10. *For any $S \in V_{\overleftarrow{\mathcal{F}}}$, define $dist(S)$ to be the number of edges in the longest acyclic path from S to $\{err\}$ in $\overleftarrow{\mathcal{F}}$. Given a program with a set of variables* Vars*, we define a set of predicates $\mathcal{P} = \{\langle A, S\rangle | S \in V_{\overleftarrow{\mathcal{F}}}, A \subseteq \text{Vars}, |A| \leq dist(S)\} \cup \{Error\}$.*

Theorem 4. *The set $\mathcal{P} \cup \{true, false\}$ is a distributively WP-closed set of predicates for the base $\{Error\}$.*

Proof. Follows from the above discussion.

Theorem 5. *If $\mathcal{F}$ is omission closed, then* $\text{SV}_{\mathcal{F}}$ *is in P.*

Proof. Immediate from Theorem 4 and Theorem 2. Note that the cardinality of $\mathcal{P}$ is $O(|Vars|^k)$, where *Vars* is the set of all variables in the program and k is the length of the longest acyclic path in $\overleftarrow{\mathcal{F}}$. (Note, from Theorem 3, that k is also bounded by the number of states in $\mathcal{F}$.) For example, `read`*`;close` verification can be done in time $O(|Vars|^2|Pgm|)$.

Stmt	WP(Stmt, *Error*)
`x := y`	*Error*
`x := new ()`	*Error*
`x.op()`	*Error* if $\overleftarrow{\delta}(\{err\}, op) = \{err\}$ $\langle\{x\}, \overleftarrow{\delta}(\{err\}, op)\rangle \vee$ *Error* if $\overleftarrow{\delta}(\{err\}, op) \supset \{err\}$
At program entry	*false*

Fig. 7. WP equations for the predicate *Error*.

Discussion

A logical formula can usually be simplified into a number of equivalent forms. Hence, a weakest-precondition can often be expressed in many ways. The form we chose to use in expressing weakest-preconditions above is critical to deriving a polynomial time verification algorithm. As an example, consider the `read`*; `close` example. The following is an alternative, correct, weakest-precondition equation, which says that an object in the *err* state is possible after `x.close()` iff either `x` points to an object in state q_2 or an object exists in the *err* state before the statement:

$$\text{WP}(\texttt{x.close()}, \mathit{Error}) = \langle\{x\}, \{q_2\}\rangle \vee \mathit{Error}. \tag{1}$$

The actual formulation we used

$$\text{WP}(\texttt{x.close()}, \mathit{Error}) = \langle\{x\}, \{\mathit{err}, q_2\}\rangle \vee \mathit{Error} \tag{2}$$

actually contains some redundancy. In particular, $\langle\{\texttt{x}\}, \{\mathit{err}, q_2\}\rangle$ is equivalent to $\langle\{\texttt{x}\}, \{\mathit{err}\}\rangle \vee \langle\{\texttt{x}\}, \{q_2\}\rangle$. But the disjunct $\langle\{\texttt{x}\}, \{\mathit{err}\}\rangle$ is redundant because it implies *Error*, another disjunct in our formula.

However, equation 2 is preferable to equation 1. In particular, we have seen that we can determine in polynomial time if $\langle\{\texttt{x}\}, \{\mathit{err}, q_2\}\rangle$ is possible at any program point. However, one can show that determining if $\langle\{\texttt{x}\}, \{q_2\}\rangle$ is possible at a program point is PSPACE-hard, adapting the proof we present in Section 4. Thus, unless PSPACE = P, a distributively WP-closed set containing $\langle\{\texttt{x}\}, \{q_2\}\rangle$ of polynomial size does not exist.

4 Repeatable Enabling Sequence Properties

In this section we show that verification of Repeatable Enabling Sequence properties (see Definition 11) is NP-complete for acyclic programs and PSPACE-complete in general.

Definition 11 (Repeatable Enabling Sequence Properties). *We say that a property represented by an automaton $\mathcal{F}$ is a repeatable enabling sequence property if there exist sequences of operations α, β and γ such that the set of sequences $\alpha\beta^+\gamma$ are all valid but the sequence $\alpha\gamma$ is invalid. (The sequence β may be thought of as a repeatable sequence that enables γ.)*

For example, the property `open`$^+$; `read` (see Figure 8) which requires that a `read` be preceded by one or more `open` operations is a repeatable enabling sequence property. We show that verification of repeatable enabling sequence properties is PSPACE-complete by reduction from the *simultaneously false* problem which is PSPACE-complete (see [18], [10]).

Definition 12. (Simultaneously False Problem) *Given a program P with an initial assignment of values (*0 *or* 1*) to a set $x_1, x_2, \ldots, x_n$ of boolean variables, where the program P contains only assignments (of constants or variables), conditionals or unconditional jumps, a* simultaneously false *problem for P is a problem of the form: is there an execution path from the entry point of P to a program point p such that $x_1 = 0, x_2 = 0, \ldots x_k = 0$ when control reaches p ?*

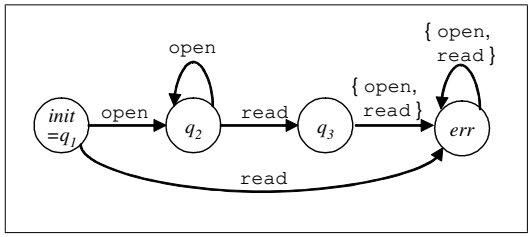

Fig. 8. An automaton for the property $\texttt{open}^+; \texttt{read}$.

Lemma 1. *(1) The simultaneously false problem for acyclic programs is NP-complete. (2) The simultaneously false problem for arbitrary programs is PSPACE-complete.*

Proof. See [18] and [10]. □

Let $\mathcal{F}$ be an automaton representing a repeatable enabling sequence property. We show that $\text{SV}_{\mathcal{F}}$ is PSPACE-hard by reduction from the simultaneously false problem. Note that if there exist sequences α, β, γ such that sequences $\alpha\beta^+\gamma$ are valid and the sequence $\alpha\gamma$ is invalid, then it must be the case that β and γ are non-empty (although α may be empty). This is because if β is empty, then $\alpha\beta^+\gamma = \alpha\gamma$, and if γ is empty, then it implies that $\alpha\beta^+$ is valid but α is invalid, which violates the prefix-closure property.

Given a binary program P and an instance of the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots, x_k = 0$ at program point p in program P, we construct a program P' as follows. First, we create two objects `Zero` and `One` which support methods corresponding to the sequences α, β, and γ. Next, we copy program P into P' replacing every assignment of the form $\texttt{x}_\texttt{i} = \texttt{0}$ by $\texttt{x}_\texttt{i} = \texttt{Zero}$ and $\texttt{x}_\texttt{i} = \texttt{1}$ by $\texttt{x}_\texttt{i} = \texttt{One}$ respectively. Furthermore, at program point p, we insert the statement $\texttt{if}\ (?)\ \texttt{goto}\ \texttt{p}_1$. Let the sequence of operations α be $a_1, a_2, \ldots, a_l$, β be $b_1, b_2, \ldots b_m$, and γ be $c_1, c_2, \ldots c_n$ respectively. Next, we insert the following sequence of operations at the end.

```
          goto exit;
p1 :      Zero.a1(); Zero.a2(); ...; Zero.a1();
          One.a1(); One.a2(); ...; One.a1();
          x1.b1(); x1.b2(); ...; x1.bm();
          x2.b1(); x2.b2(); ...; x2.bm();
          ...
          xk.b1(); xk.b2(); ...; xk.bm();
          One.c1(); One.c2(); ...; One.cn();
exit :
```

Note that control can reach program point $\texttt{p}_1$ only through the conditional branch statement $\texttt{if}\ (?)\ \texttt{goto}\ \texttt{p}_1$. This is ensured by adding the statement `goto exit;` just before $\texttt{p}_1$ which ensures that control cannot fall through into $\texttt{p}_1$.

Lemma 2. *Assuming that the sequences of operations β and γ are non-empty, the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots x_k = 0$ at program point p in P returns true if and only if program P' violates the property represented by $\mathcal{F}$.*

Proof. Program P' creates only two objects $Zero$ and One. Note that the only sequence of operations performed on $Zero$ is $\alpha\beta^i$ where i is the number of variables in $x_1, x_2, \ldots, x_k$ that are aliased to $Zero$ at program point p. Thus, no illegal operation is ever performed on $Zero$. Similarly, the only sequence of operations performed on One is $\alpha\beta^j\gamma$ where j is the number of variables in $x_1, x_2, \ldots, x_k$ that are aliased to One at program point p. This sequence is invalid iff

j can be 0. In other words, P' violates the property represented by $\mathcal{F}$ iff the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots x_k = 0$ at program point p in P returns true. □

Theorem 6. *Consider a repeatable enabling sequence property represented by an automaton $\mathcal{F}$. $\text{SV}_{\mathcal{F}}$ is NP-complete for acyclic programs and PSPACE-complete for arbitrary (cyclic) programs.*

Proof. The proofs of NP-hardness and PSPACE-hardness of acyclic and arbitrary programs resepectively follows from Lemmas 1 and 2 respectively. Lemma A1 in the Appendix shows that the problem of shallow verification for all safety properties represented by an automaton are in NP for acyclic programs and in PSPACE for arbitrary programs. □

Theorem 6 shows that verification of repeatable enabling sequence properties is difficult even for shallow programs. In fact, the situation is worse. We now show that even the shortest error paths may be of exponential size in the worst case.

Definition 13 (Error Path). *Let $\mathcal{F}$ be an automaton representing a property to be verified. We say that a path (possibly cyclic) in the control flow graph of P from the entry vertex to some vertex v is an* error path *if symbolic execution of the program along this path (ignoring the conditionals) exhibits a violation of the property associated with $\mathcal{F}$. The program P is said to be* erroneous *if there exists an error path in P. An integer-valued function f is said to be a bound on the shortest error path length if every erroneous program for size n is guaranteed to have an error path of length $f(n)$ or less.*

Theorem 7. *If NP $\neq$ PSPACE, then there does not exist a polynomial bound on the shortest error path length for repeatable enabling sequence properties.*

Proof. The proof is given in the appendix. □

Theorem 7 suggests that it may not be possible to find short counter-example paths exhibiting the violation of properties like $\texttt{open}^+;\texttt{read}$. This is important to know because many approaches to verification (e.g., [3]) are inherently associated with the generation of a counter-example path that exhibits the violation of the property of interest. Theorem 7 suggests the possibility that even the shortest error path in the program may be of size exponential in the size of the program.

5 Verification by counting

We have now seen that verification is intractable for repeatable enabling sequence properties and polynomial for omission-closed properties. Unfortunately, there are properties that fall into neither class. A simple example is the $\texttt{open};\texttt{read}$ property. Note that $\texttt{open};\texttt{read}$ is similar to $\texttt{open}^+;\texttt{read}$ in that it requires that an object be opened before it can be read, but it differs from it in that an object cannot be opened multiple times. Does this make verification any easier?

5.1 The Intuition

The requirement that an object cannot be opened multiple times is a forbidden subsequence problem (where $\texttt{open};\texttt{open}$ is the forbidden subsequence) (see Theorem 1(c)). It follows that we can verify if the given program may open an object multiple times in polynomial time. Thus, $\texttt{open};\texttt{read}$ verification is polynomial-time equivalent to $\texttt{open}^+;\texttt{read}$ verification of a program *guaranteed not to open any object more than once*. We will now show that, at least for acyclic programs, this added restriction (that an object can not be opened multiple times) does make polynomial time verification possible.

Let us begin by considering why $\texttt{read}^*;\texttt{close}$ verification is easy while $\texttt{open}^+;\texttt{read}$ verification is not. Consider the following code fragment:

```
...; p1.open(); ...; pk.open(); ...; q.read();
```

The open$^+$; read property will be violated if there is an execution path such that the value of q at the read statement is different from the values of *each* $\mathtt{p}_i$ at the corresponding open statements (assuming there are no open statements in the program other than those shown above). Determining if certain relationships can *simultaneously* exist among a potentially unbounded number of program variables is difficult.

In contrast, consider the following code fragment:

```
...; p1.close(); ...; pk.close(); ...; q.read();
```

The read*; close property will be violated here if there is an execution path such that the value of q at the read statement is equal to the value of *some* $\mathtt{p}_i$ at the corresponding close statement. In other words, this requires *independent* answers to k different questions, each about the value of only *two* program variables. This turns out to be easy.

Let us now turn back to the earlier example above.

```
...; p1.open(); ...; pk.open(); ...; q.read();
```

If we now know that no object is opened twice, how can we exploit this for open$^+$; read (i.e., open; read) verification? For any given i, we know that it is easy to determine if q.read() statement may read the same object that is opened by the $\mathtt{p}_i$.open() statement. Imagine that we can *count* the number of execution paths, n_i, along which this can happen, for each i. Adding up all the n_i would tell us how many times (i.e., along how many execution paths) the q.read() statement is a *valid* operation[3]. If this number does not equal the number of execution paths to the q.read() statement, then *there must be an execution path along which* q.read() *will read an unopened object*! Such indirect reasoning based on counting is the basis for the algorithm presented in this section.

Obviously, counting the number of paths is not feasible in the presence of cycles. In the rest of this section we will restrict our attention to acyclic, or loop-free, programs, and show how the above approach can be used for a class of verification problems.

5.2 Definitions

We start by formally defining the quantities we want to compute. Given some program P, consider a P-path ρ. Recall that $\mathcal{U}(\rho)$ denotes the set of object instances created in ρ, and for any $i \in \mathcal{U}(\rho)$, $\rho[i]$ denotes the sequence of operations performed on i. Let $\rho[\mathtt{p}]$ denote the value of variable p at the end of ρ. In the following examples, we will identify the i-th object instance created during any execution by the integer i. If s is a statement in the program, we will use s_{in} and s_{out} to denote the program points just before and just after the statement s.

Definition 14. *Let α denote a sequence of operations, π a program path, and Π_u the set of all paths from entry to a program point u. Then,*

$$ct(\alpha, \pi) \triangleq |\{\ i \in \mathcal{U}(\pi) \mid \pi[i] = \alpha\ \}|$$
$$ct(\alpha, u) \triangleq \sum_{\pi \in \Pi_u} ct(\alpha, \pi)$$

We now define auxiliary counts of the form $\widehat{ct}(\langle X, \alpha \rangle, u)$, which we will subsequently use to compute $ct(\alpha, u)$, where X is a set of program variables. Informally, the set X will constrain the counting to the object instance pointed to by all variables in X. Second, while $ct(\alpha, u)$ counts *exact* matches for α, $\widehat{ct}(\langle X, \alpha \rangle, u)$ will count *subsequence* matches for α.

[3] This is where we exploit the fact that no object is opened twice. Otherwise, adding up n_i will end up counting some paths multiple times.

Definition 15. *Given two sequences α and β, let $\widehat{ct}(\alpha, \beta)$ denote the number of times α occurs as a (not necessarily contiguous) subsequence of β.*

$$\widehat{ct}(a_1 \cdots a_k, b_1 \cdots b_m) \triangleq \mid \{(i_1, \cdots, i_k) \mid 1 \leq i_1 < \cdots < i_k \leq m \ \wedge a_1 \cdots a_k = b_{i_1} \cdots b_{i_k}\} \mid$$

In the special case where α is the empty sequence, $\widehat{ct}(\alpha, \beta)$ is defined to be 1.

Definition 16. *Given a set of variables X, we define $\mathcal{U}(\pi, X) \triangleq \{\ i \in \mathcal{U}(\pi) \mid \forall \mathtt{p} \in X.\pi[\mathtt{p}] = i\ \}$. Essentially, if X is empty, then $\mathcal{U}(\pi, X)$ is $\mathcal{U}(\pi)$. If X is non-empty and all variables in X point to the same object i then $\mathcal{U}(\pi, X)$ is $\{\ i\ \}$. If all variables in X do not point to the same object, then $\mathcal{U}(\pi, X)$ is empty.*

Definition 17. *Let α denote a sequence of operations, π a program path, and Π_u the set of all paths from the entry vertex to a program point u. Then,*

$$\widehat{ct}(\langle X, \alpha\rangle, \pi) \triangleq \sum_{i \in \mathcal{U}(\pi, X)} \widehat{ct}(\alpha, \pi[i])$$

$$\widehat{ct}(\langle X, \alpha\rangle, u) \triangleq \sum_{\pi \in \Pi_u} \widehat{ct}(\langle X, \alpha\rangle, \pi)$$

Example 2. Consider the program shown below on the left. Let u denote the program point after the last statement `y.read()`. Let ρ_1 denote the path to u where the false branch of the if-statement is taken, and let ρ_2 denote the other path to u. The table on the right shows the values of the various quantities defined above. The fact that $ct(\mathtt{read}, u)$ is non-zero indicates that the program contains a violation of the `open`; `read` property.

```
x := new ();
y := new ();
x.open();
if (?) {
  y.open();
}
x.read();
y.read();
```

X	α	$\widehat{ct}(\langle X, \alpha\rangle, \rho_1)$	$\widehat{ct}(\langle X, \alpha\rangle, \rho_2)$	$\widehat{ct}(\langle X, \alpha\rangle, u)$	$ct(\alpha, u)$
{x}	read	1	1	2	–
{x}	open; read	1	1	2	–
{y}	read	1	1	2	–
{y}	open; read	0	1	1	–
ϕ	read	2	2	4	1
ϕ	open; read	1	2	3	3

5.3 Counting Subsequences

We now show how the quantities defined above can be computed. Fig. 9 expresses the relationships that must hold between the $\widehat{ct}$ values at different program points.

Lemma 3. *For any sequence α and any acyclic program* Pgm *over a set of program variables* Vars*, $\widehat{ct}(\langle \phi, \alpha\rangle, u)$ can be computed for all program points u in polynomial time.*

Proof. We compute the values of $\widehat{ct}(\langle \phi, \alpha\rangle, u)$ using the equations presented in Fig. 9. Note that computing $\widehat{ct}(\langle \phi, \alpha\rangle, u)$ at a program point u may transitively require computing the value of $\widehat{ct}(\langle X, \beta\rangle, v)$ at some vertex v, where β is a prefix of α, and X is a set of variables of cardinality at most $|\alpha| - |\beta|$. Hence, the number of values (or equations) we need to compute at any program point is $O(|\mathit{Vars}|^{|\alpha|})$, where *Vars* is the set of all variables in the program. The result follows. □

5.4 Counting exact matches

Earlier we argued how we could compute the number of exact matches for `read` from the number of subsequence matches for `read` and the number of subsequence matches for `open`; `read`. We now present a generalization of this idea.

Statement u	Equations	
	$\widehat{ct}(\langle X, \alpha\rangle, \text{entry}_{in}) = \text{if } (\lvert X\rvert > 1 \text{ or } \lvert\alpha\rvert > 0) \text{ then 0 else 1}$ $\widehat{ct}(\langle X, \alpha\rangle, u_{in}) = \sum_{v \in \text{pred}(u)} \widehat{ct}(\langle X, \alpha\rangle, v_{out})$	
`x := y`	$\widehat{ct}(\langle X, \alpha\rangle, u_{out}) = \widehat{ct}(\langle X - \{\ \texttt{x}\ \} \cup \{\ \texttt{y}\ \}, \alpha\rangle, u_{in})$ $\widehat{ct}(\langle X, \alpha\rangle, u_{out}) = \widehat{ct}(\langle X, \alpha\rangle, u_{in})$	(if $\texttt{x} \in X$) (if $\texttt{x} \notin X$)
`x := new ()`	$\widehat{ct}(\langle\{\ \texttt{x}\ \}, \epsilon\rangle, u_{out}) = \widehat{ct}(\langle\{\ \texttt{x}\ \}, \epsilon\rangle, u_{in})$ $\widehat{ct}(\langle X, \alpha\rangle, u_{out}) = 0$ $\widehat{ct}(\langle X, \alpha\rangle, u_{out}) = \widehat{ct}(\langle X, \alpha\rangle, u_{in})$	 (if $\texttt{x} \in X$ and ($\lvert X\rvert > 1$ or $\lvert\alpha\rvert > 0$)) (if $\texttt{x} \notin X$ and $X \neq \phi$)
`x.f()`	$\widehat{ct}(\langle X, \alpha\rangle, u_{out}) = \widehat{ct}(\langle X, \alpha\rangle, u_{in})$ $\widehat{ct}(\langle X, \alpha\rangle, u_{out}) = \widehat{ct}(\langle X \cup \{\ x\ \}, \beta\rangle, u_{in}) + \widehat{ct}(\langle X, \beta f\rangle, u_{in})$	(when α is not of the form βf) (where $\alpha = \beta f$)

Fig. 9. Equations for computing the number of subsequence matches. Note that, in general, the set X may be empty, or the sequence α may be the empty sequence ϵ, but the equations assume that both X and α can not be simultaneously empty. (We are not interested in the value of $\widehat{ct}(\langle\phi, \epsilon\rangle, u)$.)

Lemma 4. *Let u denote any program point. We will use $\beta \succ \alpha$ to denote that β is a proper supersequence of α (i.e., that α is a proper subsequence of β). Then,*

$$ct(\alpha, u) = \widehat{ct}(\langle\phi, \alpha\rangle, u) - \sum_{\beta \succ \alpha} \widehat{ct}(\alpha, \beta) ct(\beta, u).$$

Proof. We will now show that $ct(\alpha, \pi) = \widehat{ct}(\langle\phi, \alpha\rangle, \pi) - \sum_{\beta \succ \alpha} \widehat{ct}(\alpha, \beta) ct(\beta, \pi)$ for any execution path π, from which the lemma follows immediately. Note that $ct(\alpha, \pi)$ counts exact matches for α in π, while $\widehat{ct}(\langle\phi, \alpha\rangle, \pi)$ counts occurrences of α as a subsequence in π. Now, consider any supersequence β of α. Every exact match for β in π will give us $\widehat{ct}(\alpha, \beta)$ subsequence matches for α. Hence, the above equality follows. □

A sequence α has infinitely many supersequences β. So, how can we make use of the above equation?

Definition 18. *A property represented by an automaton $\mathcal{F}$ is said to be* almost-omission-closed *if there exists an integer k such that for all sequences $\alpha, \beta, \gamma \in \Sigma^*$, if $\lvert\alpha\beta\gamma\rvert > k$ then $Valid_{\mathcal{F}}(\alpha\beta\gamma) \Rightarrow Valid_{\mathcal{F}}(\alpha\gamma)$.*

Let us refer to $(\alpha\gamma, \alpha\beta\gamma)$ as an omission-violation if $\alpha\beta\gamma$ is a valid sequence but $\alpha\gamma$ is not. An omission-closed property is one with no omission-violations. An almost-omission-closed property is one with only finitely many omission-violations. Note that `open; read` is an example of a verification problem where there is only one omission-violation, namely `read` is invalid but `open; read` is valid. We will now establish the following.

Theorem 8. *If $\mathcal{F}$ represents an almost-omission-closed property, then $\text{SV}_{\mathcal{F}}$ for acyclic programs is in P.*

Proof. Consider any α that is invalid. Then, any supersequence β of α of length $k+1$ must be a forbidden subsequence. Hence, we can check a program in polynomial time to see if it contains any such β. If it does, we can stop since the program does not satisfy the required property. Otherwise, we count the number of subsequence matches in the program for α and every supersequence β of α of size k or less. We can then compute the exact match count using Lemma 4. □

5.5 Verification of programs with loops?

How can we adapt the ideas described above to verify programs with loops? Given an almost-omission-closed property, if we can come up with a polynomial bound $p(n)$ on the length of the shortest error path, then we can "unroll" loops in a given program P sufficiently to generate a corresponding loop-free program P' that includes all paths of length $p(n)$ or less in P, and apply the preceding verification algorithm to P'. (Definition 20 in the appendix shows how such unrolling can be done.) This gives use the following theorem.

Theorem 9. *If $\mathcal{F}$ represents an almost-omission-closed property with a polynomial bound on the shortest error path length, then* $\text{SV}_{\mathcal{F}}$ *is in P.*

Unfortunately, we have not been able to identify polynomial bounds on the shortest error path length for almost-omission-closed properties. We conjecture that such polynomial bounds exist, at least for the `open; read` property.

6 Polynomial Time Verification for Programs with Limited Aliasing

In Section 4 we saw that, unless P = NP, verification of repeatable enabling sequence properties will require exponential time *in the worst-case*. Is it, however, possible to design verification algorithms that are efficient *in practice*, e.g., by exploiting properties of programs that arise in practice? For example, one seldom sees programs in which a very large number of variables point to the same object at a program point. In this section present a verification algorithm motivated by this observation. The algorithm runs in time $O(|Pgm|^{k+1})$, where $|Pgm|$ is the size of the program and k is the maximum aliasing factor of the program: a program is said to have a maximum aliasing factor of k if there is no path in the program that will produce an object pointed to by more than k different variables. Unlike the polynomial solutions of previous sections, the algorithm presented here works for any typestate property.

We note that naive verification algorithms do not achieve the above complexity, i.e. they may take exponential time even for programs with a maximum aliasing factor of 2. In particular, consider the obvious abstraction where the program state is represented by a partition of the program variables into equivalence classes (of variables that are aliased to each other), with a finite state associated with each equivalence class. The number of such states that can arise at a program point is exponential in the number of program variables even for programs with a maximum aliasing factor of 2.

Our algorithm uses predicates of the form $[A, S]$ defined below.

Definition 19. *Let $A \subseteq$ Vars be a non-empty set of program variables, and $S \subseteq \mathcal{Q}$ a set of states of $\mathcal{F}$.*

$$[A, S] = \bigwedge_{x \in A, y \in A} (y = x) \ \wedge \bigwedge_{x \in A, z \in \text{Vars} \setminus A} (z \neq x) \ \wedge \bigwedge_{x \in A} In_S(x) \ )$$

When S contains a single state $\sigma \in \mathcal{Q}$, we write $[A, \sigma]$, rather than $[A, \{\sigma\}]$.

Intuitively, a predicate $[A, S]$ means that all variables in A have the same value (are aliases), every variable not in A has a different value from the variables in A, and the object referred to by variables in A is in one of the state of S. The difference between $[A, S]$ and $\langle A, S \rangle$ (Definition 8) is noteworthy. The non-aliasing conditions are implicitly represented in $[A, S]$ by assuming that every variable not in A has a different value from the variables in A, whereas in $\langle A, S \rangle$, the variables not in A may or may not be aliased to the variables in A.

Fig. 10(b) presents our verification algorithm that computes, for all program points, the set of predicates of the form $[A, \sigma]$ that may-be-true at the program point. (A predicate p is said to be may-be-true at a program point u iff there exists a path to u such that execution along that path will

cause p to become true.) The algorithm is based on a standard iterative collecting interpretation algorithm. The function *flow*(`St`)(φ), defined in Fig. 10(a), identifies the set of predicates that may-be-true after statement `St` given a predicate φ that may-be-true before statement `St`. For any program point l, $Succ(l)$ denotes the successors of l.

`Statement`	*flow*(`Statement`)($[A, \sigma]$)	
`x := y`	$\{[A \cup \{x\}, \sigma]\}$	if $y \in A$
	$\{[A \setminus \{x\}, \sigma]\}$	if $y \notin A$
`x := new()`	$\{[\{x\}, \mathit{init}], [A \setminus \{x\}, \sigma]\}$	if $x \in A$
	$\{[A, \sigma]\}$	if $x \notin A$
`x.op()`	$\{[A, \delta(\sigma, op)]\}$	if $x \in A$
	$\{[A, \sigma]\}$	if $x \notin A$

(a)

```
workList = {}
for each program point l
  results(l) = {}
for each program variable x_i
  add (entry, [x_i, {init}]) to workList
while workList ≠ ∅ {
  remove (l, ψ) from workList
  for each ψ' ∈ flow(stmt_l)(ψ) {
    for l' ∈ Succ(l) {
      if ψ' ∉ results(l') {
        results(l') = results(l') ∪ {ψ'}
        add (l', ψ') to workList
} } } }
```

(b)

Fig. 10. An iterative algorithm using predicates of the form $[A, S]$.

Theorem 10. *The algorithm of Fig. 10* precisely *computes the set of predicates $[A, S]$ that may hold at any program point in time $O((\sum_{1 \le i \le k} \binom{n}{i}) * |\text{Pgm}|) = O(n^k * |\text{Pgm}|)$ where k is the maximum number of variables aliased to each other at any point in the program* Pgm*, and $n = |\text{Vars}|$ is the number of program variables.*

The proof of the theorem appears in the Appendix. Though the worst-case complexity of the algorithm is exponential, the exponential factor k is expected to be a small constant for typical programs, since the number of pointers simultaneously pointing to the same object is expected to be small (and significantly smaller than $|\mathit{Vars}|$).

Note that using the set of predicates defined in Definition 19 is not sufficient to achieve the desired complexity. The style of "forward propagation" used by our algorithm is also essential, as it ensures that the cost of analysis is proportional to the number of predicates that may-be-true (rather than the number of total predicates, as is the case with alternative analysis techniques).

7 Open Problems

In this paper we have shown that shallow verification for omission closed properties is in P and that verification for repeatable enabling sequence properties is NP-complete for acyclic programs and PSPACE-complete in general. We have also shown that verification for almost omission closed properties is in P for acyclic programs. However, many questions still remain open. For example, we do not know whether verification of almost omission closed properties is in P for arbitrary (cyclic) programs. Moreover there are other properties which do not lie in any of these classes. For example, consider the property `open`; `read`* which states that all (and any number of) `read` operations must be preceded by a single `open` operation. We can adapt the *counting* method presented in Section 5 to show that verification for `open`; `read`* is in P for acyclic programs. However, we have not been able to formulate such a result for a general class of properties that includes `open`; `read`*. Finally, there are also other properties such as (`lock`; `unlock`)* (any

number of alternating `lock` and `unlock` operations) for which we have neither been able to show a polynomial bound, nor show an NP-hardness result.

References

1. K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, May 2002.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 203–213, June 2001.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001: SPIN Workshop*, LNCS 2057, pages 103–122, 2001.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
5. J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proc. Intl. Conf. on Software Eng.*, pages 439–448, June 2000.
6. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 57–68, Berlin, June 2002.
7. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 59–69, June 2001.
8. R. DeLine and M. Fähndrich. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 13–24, Berlin, June 2002.
9. E. W. Dijkstra. *A Discipline of programing*. Prentice-Hall, 1976.
10. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Shallow finite state verification. Technical Report RC22673, IBM T.J. Watson Research Center, Dec. 2002.
11. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 234–245, Berlin, June 2002.
12. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. ACM Conf. on Programming Language Design and Implementation*, pages 1–12, Berlin, June 2002.
13. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.
14. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *In Proceedings of the 9th Conference on Computer-Aided Verification (CAV'97)*, pages 72–83, Haifa, Israel, June 1997.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
16. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. ACM Symp. on Principles of Programming Languages*, Portland, January 2002.
17. W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 93–103, New York, NY, 1991. ACM Press.
18. R. Muth and S. Debray. On the complexity of flow-sensitive dataflow analyses. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 67–80, New York, NY, 2000. ACM Press.
19. G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent sofware with FLAVERS. In *Proc. Intl. Conf. on Software Eng.*, pages 594–597, May 1997.
20. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 2001.

21. G. Ramalingam, A. Warshavsky, J. Field, D. Goyal, and M. Sagiv. Deriving specialized program analyses for certifying component-client conformance. In *Proc. ACM Conf. on Programming Language Design and Implementation*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 83–94, New York, June 17–19 2002. ACM Press.
22. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 49–61, 1995.
23. R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Software Eng.*, 19(5):478–485, May 1993.
24. R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

A Proofs

Lemma A1 *The problem of shallow verification for any safety property that is represented by a finite automaton is in NP for acyclic programs and in PSPACE for arbitrary programs*

Proof: The shallow finite state verification problem for acyclic programs can be trivially shown to be in NP by simply guessing a path through the program and checking to see if any of the objects reaches an error state.

In order to show that the verification of an arbitrary program P is in PSPACE, we can construct a non-deterministic multi-tape polynomial-space bound Turing Machine M_P to solve the problem. The input to M_P is the program P. M_P simulates program P and makes arbitrary decisions at branch points to select one of the many branches. As the simulation of P procees, M_P keeps track of all the objects that are being pointed to by the variables in P, which we call *relevant* objects. Since the number of variables in P is bounded by the size of P and we are interested only in shallow verification, the number of objects that M_P needs to keep track of is also bounded by the size of P. Note that M_P does not need to keep track of objects that are not being pointed to by any variable in P (i.e. objects that can be "garbage collected".) In addition to keeping track of which variables point to which objects, M_P also keeps track of the state of each of the relevant objects. The total amount of space needed to maintain this state is trivially polynomial bounded in the size of program P. The update to the state caused by the simulation of assignment statements can also be done in polynomial space. If any of the relevant objects goes into the error state during simulation, M_P halts and signals the possibility of an error. Conversely, if there is a path that causes one of the objects to go into the error state, then M_P can guess this path and will halt signalling the error.

Definition 20. *Consider the control-flow-graph $G_P = (V_P, E_P)$ of program P. Let $G'_P = (V_P, E'_P)$ denote the acyclic graph obtained from G_P by removing all back-edges. We define Unroll(G_P, n) to be the acyclic graph obtained by making $n+1$ copies of G'_P (called $G'_P(1), G'_P(2), \ldots G'_P(n+1)$ respectively), and for every back-edge (u, v) in G_P, adding an edge from vertex u in $G''_P(i)$ to vertex v in $G''_P(i+1)$ for all i from 1 to v. More formally Unroll$(G_P, n) = (V^*, E^*)$ where*

$$V^* = \{\ (v,i) \mid\ v \in V_P,\ 1 \le i \le n+1\ \}$$
$$E^* = \{\ [(u,i),(v,i)] \mid\ [u,v] \in E'_P,\ 1 \le i \le n+1\ \} \cup$$
$$\{[(u,i),(v,i+1)] \mid [u,v] \in E_P - E'_P, 1 \le i \le n\ \}$$

It is easy to verify that *Unroll*(G_P, v) is acyclic and that it contains every path of length v or less in G_P.

Proof of Theorem 7

Let $\mathcal{F}$ be the finite state automaton associated with the repeatable enabling sequence property. From Theorem 6 it follows that verification of $\mathcal{F}$ for acyclic programs is in NP and for arbitrary

(cyclic) programs is PSPACE-hard. Therefore, if NP $\neq$ PSPACE, then the verification problem for cyclic programs cannot be polynomial-time reducible to the verification problem for acylic programs. We prove Theorem 7 by contradiction by showing that if there is a polynomial bound on the shortest error path, then the verification problem for cyclic programs can be polynomial-time reduced to the verification problem for acyclic programs.

Let, if possible, $p(n)$ denote the polynomial bound on the size of the shortest error path where n denotes the size of the program. Given an arbitrary program P with control flow fraph G_P, we construct the acyclic program *Unroll*$(G_P, p(n))$ which is acyclic and contains all paths of length $p(n)$ or less in G_P. The size of *Unroll*$(G_P, p(n))$ and the time taken to construct it are both polynomial in n. Thus, the problem of verification of G_P is polynomially reduced to the problem of verifying *Unroll*$(G_P, p(n))$, which is a contradiction.

Proof of Theorem 10

The proof that the algorithm computes the precise solution is straightforward and requires showing that $\cup_{\varphi \in P}$*flow*$(\texttt{St})(\varphi)$ computes a precise abstract transfer function for statement `St` with respect to the set of predicates P, and that this is a distributive function.

We now establish the complexity of the algorithm. Assume that the maximal size of an alias-set occurring in the program is k. The algorithm may generate predicates of the form $[A, S]$ for all subsets of any size up to k of program variables *Vars*. The number of predicates that may have a *true* value in a program point is therefore $O(\sum_{1 \leq i \leq k} \binom{n}{i})$ where $n = |Vars|$ (we treat the number of FSM states as a constant).

The complexity of the chaotic iteration algorithm of Fig. 10 is therefore $O((\sum_{1 \leq i \leq k} \binom{n}{i}) * |Pgm|)$. The expression is also bounded by $O(n^k * |Pgm|)$.

The above assumes that the step of computing *flow*$(\texttt{stmt}_l)(\psi)$ takes constant time.