Typestate Verification: Abstraction Techniques and Complexity Results

J. Field [a], D. Goyal [a,1], G. Ramalingam [a], E. Yahav [b]

[a] *IBM T.J. Watson Research Center.*

[b] *School of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel.*

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. 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.

In solving a problem of this sort, the grand thing is to be able to reason backward. ... In the everyday affairs of life it is more useful to reason forward.
–Sir Arthur Conan Doyle, *A Study in Scarlet*.

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 [27,26,21,6,8,3,9,13,12,17,1]). In this model, objects of a given type exist in one of finitely many *states*; the operations permitted on an

Email addresses: jfield@watson.ibm.com (J. Field), dgoyal@calypto.com (D. Goyal), rama@watson.ibm.com (G. Ramalingam), yahave@post.tau.ac.il (E. Yahav).

[1] Author's current affiliation: Calypto Design Systems Inc.

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 be used to check that objects satisfy certain kinds of temporal properties; e.g., that an object is not used before it is initialized, or that a file is not used after it is closed. In this paper, we will specify such properties using regular expressions or finite state automata that define the set of *valid* sequences of operations that can be performed on an 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*. Among other things, we will show that not all finite state properties are equally hard to verify. For example, given a *shallow* program (where pointers from program variables to heap-allocated objects are allowed, but where heap-allocated objects may not themselves contain pointers), we show that verifying that a file is not read after it is closed can be done in *polynomial time*, while verifying that a file is not read before it is opened is *PSPACE-Complete*.

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 [23].

Typestate Verification and Shallow Programs

In order to meaningfully compare the complexity of verification algorithms, we need to make some baseline assumptions about the precision of the analysis. In this paper, 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. Specifically, given a finite state 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 typestate verification is to determine if a given program has any error path.

Typestate verification can be done in polynomial time if the program to be verified allows no inter-variable aliasing. Conversely, it is a straightforward consequence of previous results [18,20] that if a program has *two or more* levels of pointers, typestate verification is PSPACE-hard [2]. In this paper, we therefore concentrate on

[2] In the presence of recursive data structures, typestate verification is undecidable [19,24].

understanding the class of *shallow* programs occupying a point in between these extremes.

Assume we wish to perform typestate verification for objects of a type T. A T-*shallow* program is a well-typed procedure-free program where all variables are pointers to T-typed objects, and whose statements are 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 may contain multiple pointers to objects of type T, but allocated objects may not themselves contain T-pointers. In other words, pointers in shallow programs are *single-level* [20]. Our results also apply to programs that manipulate complex or recursive types where allocated objects contain pointers, *provided that those pointers cannot refer to objects of type* T. Programs that are shallow with respect to a given type, e.g. `File`, are not uncommon in practice.

Example: Verifying File Operations

Consider the problem of checking that a closed file is never read or closed again, which we will refer to as `read`*; `close`. In general, we will use regular expressions to designate sequences of *valid* operations on an object of a given type, where a sequence is valid iff it is a prefix of a string in the language defined by the regular expression. For example, `read`; `read` is a prefix of `read`; `read`; `close` and thus a valid sequence.

The principal difficulty in doing precise verification arises from determining how *aliasing* interacts with operations on objects. Some prior work on typestate verification (e.g. [7]) 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. However, we can see from the program fragments in Figure 1 that such an approach can sometimes lead to imprecise results. One can easily verify that in both Figures 1(a) and 1(b), all sequences of file operations on a given object are prefixes of `read`*; `close`; i.e., that no `read` ever follows a `close`.

However, consider a two-phase analysis in which the heap analysis is separate from the typestate analysis. In Fig. 1(a), a precise (and correct) heap analysis will determine that program variable `z` at program point `s2` may point to the object created at `s0` or the object created at `s1`. Furthermore, a precise typestate analysis will determine that the object created at `s1` could be in a *closed* state at `s2`. A two-phase analysis must therefore erroneously conclude 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 `s3` and `s5` could reach the read statement at `s4`. In addition, a typestate analysis would also determine that the objects created

(a)	(b)
`s0 : x := new ();`	`s3 : f := new ();`
`s1 : y := new ();`	`while (?) {`
`z := y;`	`  s4 : f.read();`
`if (?) {`	`  if (?) {`
`  y.close();`	`    f.close();`
`  z := x;`	`    s5 : f := new ();`
`}`	`  }`
`s2 : z.read();`	`}`

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

at program points `s3` and `s5` could be in a closed state at `s4`. The analysis would, however, not be able to discover that `f` can never point to a closed object at `s4`, and would incorrectly indicate a possible error. In this paper we show that for a certain class of problems (including `read`*; `close`), it is possible to formulate a precise polynomial time verification algorithm for shallow programs.

Main Results

The main complexity results established in this paper are as follows (in all cases except the last one, 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: `read`*; `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 an automaton state where a particular sequence γ of operations is invalid, but sequences of the form $\beta^+\gamma$ are valid for some β. Example: `open`$^+$; `read`.
- An integer-valued function f is said to be a bound on the shortest error path length for a typestate 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., 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: `open; 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 width 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 width of k may be verified in time $O(n^{k+1})$ for programs of size n.
- Alias analysis and typestate verification are NP-hard for programs with maximum aliasing width of three and aliasing depth of two. (A program is said to have aliasing depth of two if the program contains pointers to pointers).

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).

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 [23]). The bulk of the abstractions we use are *predicate abstractions* [15]; however we show in the sequel that the choice of predicates used in a predicate abstraction can have a dramatic impact on the efficiency of the resulting analysis. Our predicate vocabularies are carefully designed to yield efficient analyses without sacrificing precision. In addition, in Section 5, we develop a novel *integer* abstraction, which is based on *counting* the number of program paths along which a simple property holds true; this in turn allows inferring whether a more complex property holds.

Related Work

There has been significant recent interest in a variety of property verification techniques, many of them focusing on typestate verification. While significant progress has been made in improving the precision and efficiency of verification, developing verification techniques that are sufficiently precise and scalable to handle industrial-size applications for a wide variety of problems is still a challenge, and motivates our work here.

One of the open challenges in typestate verification is an adequate treatment of aliasing. Some approaches avoid the issue: e.g., the original work on typestate verification [27,26] did not allow any aliasing; more recent work on typestate verification based on linear types [8] also restricts aliasing severely. Other approaches (e.g. [7]) perform alias analysis and typestate verification separately: an initial phase performs

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

Fig. 2. An overview of our complexity results.

a conservative alias analysis for the program, and a subsequent phase uses the information from the alias analysis to do typestate verification. However, this can lead to imprecise results, as illustrated by the examples in Fig. 1.

A second challenge to practical verification is dealing with infeasible program paths. Das et al. [7] address this issue using efficient path-sensitive algorithms (which eliminate certain infeasible paths from consideration during analysis), but do not track certain additional information, e.g., aliasing, precisely. Our algorithms do not address the question of path sensitivity, but there could be merit in combining aspects of our approach with those that eliminate infeasible paths.

One of the primary intuitions behind the algorithms presented in this paper (for shallow programs) is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency. The recent work of [28], following this paper, shows one way to exploit this intuition for verification of arbitrary (i.e. non-shallow) programs as well.

Several recent verification approaches [2,16] combine predicate abstraction [15], counterexample-guided refinement of the predicate vocabulary [4], and exploration of the resulting abstract state space using model-checking. These techniques use symbolic and theorem-proving techniques to identify a set P of predicates relevant to the problem of interest, then model-check the resulting finite state system over a state space constructed from the powerset lattice $2^{P \rightarrow \{\mathit{true}, \mathit{false}\}}$. This process iterates with increasingly larger sets of predicates until a satisfactory result is obtained. In principle, these algorithms have the potential to avoid imprecision due to both aliasing and path infeasibility. However, the worst-case complexity of a *single* iteration is exponential in the number of predicates. By contrast, while most of the algorithms we present are based on abstractions by a set of predicates Q, our analysis is based on the function-space lattice $Q \rightarrow \{\mathit{false}, \mathit{maybe}\}$, and runs in time linear in the size of Q. This approach yields polynomial-time algorithms, while none of the techniques based on model-checking have a polynomial time worst-case complexity for the same problems (even though they may utilize a smaller number of predicates than our algorithm). Our selection of predicates ensures that the use of the smaller function space lattice results in no loss of precision, i.e., we ensure that our abstraction is *complete* (e.g., see [14]). Finally, the predicate abstractions we use are dependent solely on the nature of the typestate problem being verified, and do not require expensive predicate discovery at verification time.

Finally, we note that our lower bound results follow the tradition set by earlier complexity results due to Landi and Ryder [18] and Muth and Debray [20].

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 when a program begins execution all program variables point to separate objects (i.e., initialized to non-aliased values), and all objects reside in their initial state. 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

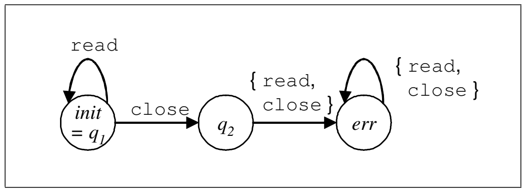

Fig. 3. A finite-state automaton for the property $\texttt{read}^*;\texttt{close}$.

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 [5]), 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) $\mathcal{F} = \langle \Sigma, \mathcal{Q}, \delta, \mathit{init}, \mathcal{Q} \setminus \{\mathit{err}\} \rangle$ where Σ is the automaton alphabet consisting 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 $\sigma \in \Sigma$, $\delta(\mathit{err}, \sigma) = \mathit{err}$, and all states in $\mathcal{Q} \setminus \{\mathit{err}\}$ are accepting states. We say that q' is the successor of a state q on operation* $\texttt{op}$ *when $\delta(q, \texttt{op}) = q'$. Given a sequence of operations $\alpha = \texttt{op}_1; \texttt{op}_2; \ldots; \texttt{op}_k$, we write $\mathit{Valid}_{\mathcal{F}}(\alpha)$ or $\alpha \in \mathit{Valid}_{\mathcal{F}}$ when α is accepted by $\mathcal{F}$, and we write $\mathit{Invalid}_{\mathcal{F}}(\alpha)$ 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 $\texttt{read}^*;\texttt{close}$ we also consider ϵ (the empty sequence) and `read` to be valid sequences.

Example 3 *Consider the property* $\texttt{read}^*;\texttt{close}$ *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.*

When verifying a safety property represented by an automaton $\langle \mathcal{Q}, \mathit{init}, \mathit{err}, \Sigma, \delta \rangle$ 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 4 (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 5 ($\mathbf{SV}_{\mathcal{F}}$) *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 6 *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 7 *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. Any sequence containing some ξ_i is invalid (from (b)), and it is clear that any invalid sequence must contain an acyclic path from the initial state to the error state as a subsequence. (For example, the forbidden subsequences for the automaton in Fig. 3 are ξ_1 = `close`; `read` and $\xi_2 =$ `close`; `close`.) The result follows.

Example 3.1 *Consider the automaton $\mathcal{F}_3$ of Fig. 3. For this automaton, the sequence* `read`; `read`; `close` *is in Valid$_{\mathcal{F}_3}$, and so is the sequence* `read`; `close` *obtained by dropping the intermediate* `read` *operation. Moreover, for any valid sequence* `read`*; `close`*, dropping any subsequence of* `read`*s, or dropping the* `close` *yields a valid sequence.*

For $\mathcal{F}_3$, it is sufficient to consider the forbidden subsequences ξ_1 = `close`; `read` *and ξ_2 =* `close`; `close`*. Each sequence α containing ξ_1 or ξ_2 as a subsequence is in Invalid$_{\mathcal{F}_3}$, and each sequence in Invalid$_{\mathcal{F}_3}$ contains ξ_1 or ξ_2 as a subsequence.*

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 program-state. (We will use the term *program-state* to denote the state of the whole program in the concrete semantics, to distinguish it from a *state in a FSA specifying a property*.) For efficiency reasons, we will utilize an *independent attributes analysis* [22], an analysis that does not maintain the correlation between different predicate values. Specifically, the set of concrete program-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` [10].

Definition 8 *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$(\texttt{St}, P_i) = P_{j_1} \vee \ldots \vee P_{j_m}$*, where $P_{j_1}, \ldots, P_{j_m} \in \mathcal{P}$. We also say that the set of predicates $\mathcal{P}$ is* distributively WP-closed.

Theorem 9 *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 $O(|\mathcal{P}||\mathrm{Pgm}|)$.*

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 [25]. We note that the analysis can also identify paths that will cause a given predicate to become true at a given point when such a path exists. $\square$

We use a designated predicate *Error* that is *true* in a program-state if and only if the program-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, a distributive WP closure for $\{Error\}$ needs to include predicates that refer to 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 10 *We write $In_\sigma(\mathtt{x})$ to denote the fact that the object pointed to by the variable $\mathtt{x}$ 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 $\mathtt{x}$ is in one of the states in S.*

Definition 11 *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 *relevant*, i.e. need to be in a distributive WP closure for $\{Error\}$. The key to obtaining a polynomial size distributive WP closure for $\{Error\}$ is to bound the size of the set A, for any relevant predicate $\langle A, S\rangle$ by a constant. We will do this in two steps. First, we will show that a predicate $\langle A, S\rangle$ is relevant only for certain $S \subseteq \mathcal{Q}$. Then, we will show that for each such set S, the predicate $\langle A, S\rangle$ is relevant 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 $\overleftarrow{\mathcal{F}} = (V_{\overleftarrow{\mathcal{F}}}, E_{\overleftarrow{\mathcal{F}}})$, where each vertex is a subset 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 12 *Let $\overleftarrow{\delta}$ denote the* reverse transition relation *of $\mathcal{F}$, i.e., 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 $S_1, S_2 \subseteq \mathcal{Q}$, S_2 is said to be a* pre-image *of S_1 if $\exists a \in \Sigma. \overleftarrow{\delta}(S_1, a) = S_2$.*

$V_{\overleftarrow{\mathcal{F}}} = \emptyset$; $E_{\overleftarrow{\mathcal{F}}} = \emptyset$; $\mathit{workSet} = \{\{\mathit{err}\}\}$;
`while` $\mathit{workSet} \neq \emptyset$ {
  `select and remove` S `from` $\mathit{workSet}$;
  `for each operation op` $\in \Sigma$ {
    $P = \overleftarrow{\delta}(S, op)$;
    `if` $P \notin V_{\overleftarrow{\mathcal{F}}}$ { $V_{\overleftarrow{\mathcal{F}}} = V_{\overleftarrow{\mathcal{F}}} \cup \{P\}$; $\mathit{workSet} = \mathit{workSet} \cup \{P\}$; }
    $E_{\overleftarrow{\mathcal{F}}} = E_{\overleftarrow{\mathcal{F}}} \cup \{P \to S\}$;
} }

Fig. 4. Backwards exploration of the property automaton.

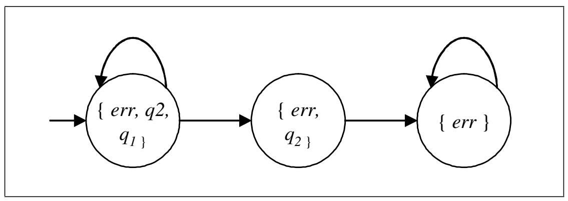

Fig. 5. The graph constructed by backward exploration of the automaton of Fig. 3.

Fig. 5 illustrates the graph constructed by backward exploration of the `read`*; `close` automaton shown in Fig. 3. We now establish a result about the graph $\overleftarrow{\mathcal{F}}$.

Theorem 13 *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. $\square$

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 equations, we can determine which predicates are relevant for verification. The 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, they show 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, 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 $\{\mathit{err}\}$ in $\overleftarrow{\mathcal{F}}$.

Definition 14 *For any $S \in V_{\overleftarrow{\mathcal{F}}}$, define $dist(S)$ to be the number of edges in the*

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 $A = \{\mathtt{x}\} \wedge \mathit{init} \in S$
	false	if $A = \{\mathtt{x}\} \wedge \mathit{init} \notin S$
`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 $\lvert A\rvert = 1 \wedge \mathit{init} \in S$
	false	if $\lvert A\rvert \neq 1 \vee \mathit{init} \notin S$

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.

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

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

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 $\mathcal{P} = \{\langle A, S\rangle | S \in V_{\overleftarrow{\mathcal{F}}}, A \subseteq \mathrm{Vars}, |A| \leq dist(S)\} \cup \{\mathit{Error}\}$.*

Theorem 15 *The set $\mathcal{P} \cup \{\mathit{true}, \mathit{false}\}$ is a distributively WP-closed set of predicates for $\{\mathit{Error}\}$.*

PROOF. Follows from the above discussion.

Theorem 16 *If $\mathcal{F}$ is omission-closed, then $\mathrm{SV}_{\mathcal{F}}$ is in P.*

PROOF. Immediate from Theorem 15 and Theorem 9. 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 13, that k is also bounded by the number of states in $\mathcal{F}$.)

Example 3.2 *Consider the property* $read^{*}close$ *represented by the automaton of Fig. 3. The graph* $\overleftarrow{\mathcal{F}}$ *for this automaton is shown in Fig. 5. The derivation for this property is as follows*[3] *:*

$$\mathrm{WP}(\texttt{x.read()}, \mathit{Error}) = \langle\{x\}, \{\mathit{err}, q_2\}\rangle \vee \mathit{Error}$$

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

$$\mathrm{WP}(\texttt{y.close()}, \langle\{x\}, \{\mathit{err}, q_2\}\rangle) = \langle\{x, y\}, \{\mathit{err}, q_2, q_1\}\rangle \vee \langle\{x\}, \{\mathit{err}, q_2\}\rangle$$

$$\mathrm{WP}(\texttt{w.read()}, \langle\{x, y\}, \{\mathit{err}, q_2, q_1\}\rangle) = \langle\{x, y\}, \{\mathit{err}, q_2, q_1\}\rangle$$

Thus, `read`*; `close` *verification can be done in time* $O(|\mathrm{Vars}|^2|\mathrm{Pgm}|)$.

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:

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

The actual formulation we used

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

actually contains some redundancy. In particular, $\langle\{\mathrm{x}\}, \{\mathit{err}, q_2\}\rangle$ is equivalent to $\langle\{\mathrm{x}\}, \{\mathit{err}\}\rangle \vee \langle\{\mathrm{x}\}, \{q_2\}\rangle$. But the disjunct $\langle\{\mathrm{x}\}, \{\mathit{err}\}\rangle$ 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 $\langle\{\mathrm{x}\}, \{\mathit{err}, q_2\}\rangle$ is possible at any program point.

[3] Note that the variables `x`, `y`, and `w` used in the derivation process are free variables and not variables of a specific program.

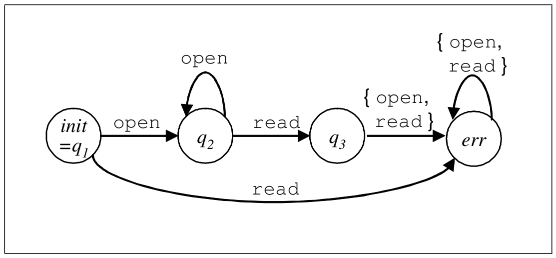

Fig. 8. An automaton for the property $\texttt{open}^{+}; \texttt{read}$.

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*! Note that the set $\{q_2\}$ has a pre-image (namely $\overleftarrow{\delta}(\{q_2\}, \texttt{close})$ = $\{q_1\}$) that is not a superset of $\{q_2\}$, thus not satisfying the requirements of Theorem 13. This is why the proof used for omission-closed properties cannot be used for this predicate.

4 Repeatable Enabling Sequence Properties

In this section we show that verification of Repeatable Enabling Sequence properties (see Definition 17) is NP-complete for acyclic programs and PSPACE-complete in general.

Definition 17 (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 $\texttt{open}^{+}; \texttt{read}$ (see Figure 8) which requires that a $\texttt{read}$ be preceded by one or more $\texttt{open}$ operations is a repeatable enabling sequence property. (The more natural property $\texttt{open}^{+}; \texttt{read}^{*}$ is also a repeatable enabling sequence property, but we use $\texttt{open}^{+}; \texttt{read}$ as the running example to contrast it with the omission-closed property $\texttt{read}^{*}; \texttt{close}$.) We show that verification of repeatable enabling sequence properties is PSPACE-complete by reduction from the *simultaneously false* problem (see [20], [11]).

Definition 18 (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 ?*

Lemma 19 *(1) The simultaneously false problem for acyclic programs is NP-complete. (2) The simultaneously false problem for arbitrary programs is PSPACE-complete.*

PROOF. The binary simultaneous value problem can be easily reduced to the simultaneously false problem by following the construction used in the proof of Theorem 3.6 in Muth and Debray [20]. The idea is to transform a program P into a program P' such that every variable x_i in P corresponds to two variables X_i and $\overline{X_i}$, every assignment $x_i = 0$ is converted to $X_i = 0; \overline{X_i} = 1$, every assignment $x_i = 1$ is converted to $X_i = 1; \overline{X_i} = 0$, and every assignment $x_i = x_j$ is converted into $X_i = X_j; \overline{X_i} = \overline{X_j}$. Consider the simultaneous value problem $x_1 = c_1, x_2 = c_2, \ldots, x_k = c_k$ for P. It can be easily shown that the simultaneously false problem for P' obtained by replacing every conjunct $x_i = 0$ with $X_i = 0$ and $x_i = 1$ with $\overline{X_i} = 0$ is equivalent. Thus, the simultaneously false problem is also NP-complete and PSPACE-complete for acyclic and arbitrary programs respectively. □

Let $\mathcal{F}$ be an automaton representing a repeatable enabling sequence property. We show that SV$_{\mathcal{F}}$ is PSPACE-hard by reduction from the simultaneously false problem. If α, β, γ are such that sequences $\alpha\beta^+\gamma$ are valid and sequence $\alpha\gamma$ is invalid, then β and γ must be non-empty (although α may be empty). Given an instance of the simultaneously false problem $x_1 = 0, x_2 = 0, \ldots, x_k = 0$ at program point p in a 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 $\mathtt{x_i} = 0$ by $\mathtt{x_i} = \mathtt{Zero}$ and $\mathtt{x_i} = 1$ by $\mathtt{x_i} = \mathtt{One}$ respectively. Then, at program point p, we insert the statement `if (?) goto p₁`. Let the sequence α be $a_1, a_2, \ldots, a_l$, let β be $b_1, b_2, \ldots b_m$, and let γ be $c_1, c_2, \ldots c_n$. We insert the following sequence of statements at the end.

$$
\begin{array}{ll}
 & \mathtt{goto\ exit;} \\
\mathtt{p_1}: & \mathtt{Zero.a_1}(); \mathtt{Zero.a_2}(); \ldots; \mathtt{Zero.a_l}(); \\
 & \mathtt{One.a_1}(); \mathtt{One.a_2}(); \ldots; \mathtt{One.a_l}(); \\
 & \mathtt{x_1.b_1}(); \mathtt{x_1.b_2}(); \ldots; \mathtt{x_1.b_m}(); \\
 & \mathtt{x_2.b_1}(); \mathtt{x_2.b_2}(); \ldots; \mathtt{x_2.b_m}(); \\
 & \ldots \\
 & \mathtt{x_k.b_1}(); \mathtt{x_k.b_2}(); \ldots; \mathtt{x_k.b_m}(); \\
 & \mathtt{One.c_1}(); \mathtt{One.c_2}(); \ldots; \mathtt{One.c_n}(); \\
\mathtt{exit}: &
\end{array}
$$

Note that control can reach program point $\mathtt{p_1}$ only through the conditional branch statement `if` $(?)$ `goto` $\mathtt{p_1}$ (because of the statement `goto exit`; just before $\mathtt{p_1}$).

Lemma 20 *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 $Zero$. 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. $\square$

The above lemma shows the hardness of typestate verification for repeatable enabling sequence properties. We now establish a straightforward completeness result.

Lemma 4.1 *For any automaton $\mathcal{F}$,* $\mathrm{SV}_{\mathcal{F}}$ *is in NP for acyclic programs and in PSPACE for arbitrary programs.*

Proof: $\mathrm{SV}_{\mathcal{F}}$ is in NP for acyclic programs since we can non-deterministically choose a path through the program and check to see if any object reaches the error state during execution along that path. To show that $\mathrm{SV}_{\mathcal{F}}$ for an arbitrary program P is in PSPACE, we construct a non-deterministic multi-tape polynomial-space-bounded Turing Machine M to solve the problem. M simulates input program P, non-deterministically choosing the branch to take at branch points. Let us refer to objects pointed to by the variables in P as *live* objects. M keeps track of which variables point to which (live) objects, and tracks the finite-state of each live object. The space needed to maintain this information is trivially bounded by a polynomial in the size of program P. If any of the relevant objects goes into the error state during simulation, M 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 can guess this path and will halt signalling the error. $\square$

Theorem 21 *Consider a repeatable enabling sequence property represented by an automaton $\mathcal{F}$.* $\mathrm{SV}_{\mathcal{F}}$ *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 19 and 20 respectively. Lemma 4.1 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. $\square$

Theorem 21 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 22 (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.*

Definition 23 (Loop Unrolling) *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 contains every path of length v or less in G_P.

Theorem 24 *If NP $\neq$ PSPACE, then there does not exist a polynomial bound on the shortest error path length for repeatable enabling sequence properties.*

PROOF. Let $\mathcal{F}$ be the finite state automaton associated with the repeatable enabling sequence property. From Theorem 21 it follows that verification of $\mathcal{F}$ for acyclic programs is in NP and for arbitrary (cyclic) programs is PSPACE-hard. We prove Theorem 24 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, which would imply that NP = PSPACE.

Let $p(n)$ denote a 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. □

Theorem 24 suggests that it may not be possible to find short counterexample paths exhibiting the violation of properties like `open`$^+$; `read`. This is important to know because many approaches to verification (e.g., [3]) are inherently associated with the generation of a counterexample path that exhibits the violation of the property of interest. Theorem 24 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 `open`; `read` property. Note that `open`; `read` is similar to `open`$^+$; `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 `open`; `open` is the forbidden subsequence) (see Theorem 7(c)). It follows that we can verify if the given program cannot open an object multiple times in polynomial time. Thus, `open`; `read` verification is polynomial-time equivalent to `open`$^+$; `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 `read`*; `close` verification is easy while `open`$^+$; `read` verification is not. Consider the following code fragment:

$$\texttt{...; p}_1\texttt{.open(); ...; p}_k\texttt{.open(); ...; q.read();}$$

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:

$\texttt{...; p}_1\texttt{.close(); ...; p}_k\texttt{.close(); ...; q.read();}$

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* $\texttt{p}_i$ 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.

$\texttt{...; p}_1\texttt{.open(); ...; p}_k\texttt{.open(); ...; q.read();}$

If we now know that no object is opened twice, how can we exploit this for $\texttt{open}^+$; `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 $\texttt{p}_i\texttt{.open()}$ 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 [4] . 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 $\rho[\mathrm{p}]$ denote the value of variable `p` at the end of ρ. 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.

[4] 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 25 *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, define ct$(\alpha, \pi) \triangleq |\{ i \in \mathcal{U}(\pi) \mid \pi[i] = \alpha \}|$ and ct$(\alpha, u) \triangleq \sum_{\pi \in \Pi_u}$ ct(α, π)*

We now define auxiliary counts of the form $\widehat{ct}(\langle X, \alpha \rangle, u)$, which we will subsequently use to compute *ct*(α, 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*(α, u) counts *exact* matches for α, $\widehat{ct}(\langle X, \alpha \rangle, u)$ will count *subsequence* matches for α.

Definition 26 *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.*

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

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

Definition 27 *Given a set of variables X, we define $\mathcal{U}(\pi, X) \triangleq \{ i \in \mathcal{U}(\pi) \mid \forall \mathrm{p} \in X. \pi[\mathrm{p}] = i \}$. 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 28 *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, define $\widehat{ct}(\langle X, \alpha \rangle, \pi) \triangleq \sum_{i \in \mathcal{U}(\pi, X)} \widehat{ct}(\alpha, \pi[i])$ and $\widehat{ct}(\langle X, \alpha \rangle, u) \triangleq \sum_{\pi \in \Pi_u} \widehat{ct}(\langle X, \alpha \rangle, \pi)$*

Example 29 *Consider the following program:*

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

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 below 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

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

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 $\widehat{ct}(\langle \phi, \epsilon \rangle, u)$.)

violation of the `open; read` *property.*

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 30 *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. $\square$

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.

Lemma 31 *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,*

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

PROOF. We will now show that $\mathit{ct}(\alpha, \pi) = \widehat{ct}(\langle \phi, \alpha \rangle, \pi) - \sum_{\beta \succ \alpha} \widehat{ct}(\alpha, \beta)\mathit{ct}(\beta, \pi)$ for any execution path π, from which the lemma follows immediately. Note that $\mathit{ct}(\alpha, \pi)$ 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. $\square$

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

Definition 32 *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 $|\alpha\beta\gamma| > k$ then $\mathit{Valid}_{\mathcal{F}}(\alpha\beta\gamma) \Rightarrow \mathit{Valid}_{\mathcal{F}}(\alpha\gamma)$.*

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 33 *If $\mathcal{F}$ represents an almost-omission-closed property, then* $\mathrm{SV}_{\mathcal{F}}$ *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 31. $\square$

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 23 shows how such unrolling can be done.) This gives us the following theorem.

Theorem 34 *If $\mathcal{F}$ represents an almost-omission-closed property with a polynomial bound on the shortest error path length, then* $\mathrm{SV}_{\mathcal{F}}$ *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 Programs with Width-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. Let us say that a program has a maximum *aliasing width* of k if there is no execution path in the program that will produce an object pointed to by more than k different variables. In this section, we look at the complexity of typestate verification for programs where the maximum aliasing width is bounded by a constant.

6.1 Polynomial Time Verification for Shallow Programs with Width-Limited Aliasing

In this section we present a verification algorithm motivated by the observation that the aliasing width of programs tend to be small in practice. The algorithm runs in time $O(|Pgm|^{k+1})$, where $|Pgm|$ is the size of the program and k is the maximum aliasing width of the program: 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 width 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 program-states that can arise at a program point is exponential in the number of program variables even for programs with a maximum aliasing width of 2.

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

Definition 35 *Let* $A \subseteq \text{Vars}$ *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 11) 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. 11 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*($\texttt{St}$)(φ), defined in Fig. 10, 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 $x \notin A$

Fig. 10. *flow* equations for predicates of the form $[A, \sigma]$.

```
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
} } } }
```

Fig. 11. An iterative algorithm using predicates of the form $[A, S]$.

Theorem 36 *The algorithm of Fig. 11* precisely *computes the set of predicates* $[A, S]$ *that may hold at any program point in time* $O((\sum_{1 \leq i \leq k} \binom{n}{i}) * |\text{Pgm}|) = O(n^k * |\text{Pgm}|)$ *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.*

PROOF. It can be shown that (a) $\cup_{\varphi \in P}\mathit{flow}(\texttt{St})(\varphi)$ computes a precise abstract transfer function for statement `St` with respect to the set of predicates P, and that (b) this is a distributive function. It directly follows from these facts that the algorithm computes the precise solution.

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 $n = |\mathit{Vars}|$ (we treat the number of FSM states as a constant). The complexity of the chaotic iteration algorithm of Fig. 11 is therefore $O((\sum_{1\leq i\leq k}\binom{n}{i}) * |\mathit{Pgm}|)$. The expression is also bounded by $O(n^k * |\mathit{Pgm}|)$. The above assumes that the step of computing *flow*$(\mathtt{stmt}_l)(\psi)$ takes constant time. $\Box$

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 35 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).

6.2 Width-Limited Aliasing in Non-Shallow Programs

We have now seen that typestate verification can be done efficiently for programs where the aliasing is bounded in certain ways. Specifically, the results of the previous subsection show that for shallow programs, typestate verification can be done in polynomial time if the aliasing width is assumed to be bounded by a constant. A natural question is whether any such result holds true for non-shallow programs.

Recall that shallow programs are programs where the aliasing *depth* is restricted to be one: program variables may point to objects, but program contains no variables that point to objects that contain pointers to objects.

Unfortunately, it turns out that typestate verification is hard for non-shallow programs even if aliasing width is bounded by a constant. It is known [18] that alias analysis is intractable for programs where the aliasing depth is two. We now show that the intractability result holds even if in addition the aliasing width is also restricted to three.

Theorem 37 *Alias analysis is NP-hard for programs with aliasing depth two and aliasing width three.*

PROOF. The proof is by reduction from 3-SAT. Consider a 3-SAT formula $C_1 \wedge C_2 \cdots \wedge C_n$ over logical variables w_1 through w_m. We create a program with a type `T` and a second type `PT` consisting of a field `f` of type (pointer to) `T`. Corresponding to every clause C_i, the program consists of variables $\mathtt{X_i}$, $\mathtt{Y_{i,true}}$, and $\mathtt{Y_{i,false}}$ of type (pointer to) `PT` initialized as follows:

```
Y_{i,true} = new PT(); Y_{i,true}.f = new T();
Y_{i,false} = new PT(); Y_{i,false}.f = new T();
X_i = Y_{i,false}
```

Both $\texttt{Y}_{\texttt{i,true}}$ and $\texttt{Y}_{\texttt{i,false}}$ are constants in the program.

After the initialization code, the program consists of one if-then-else statement for every logical variable w_i in the 3-SAT formula. The then-branch of this statement consists of an assignment statement $\texttt{X}_\texttt{i} = \texttt{Y}_{\texttt{i,true}}$ for every clause C_i that contains the literal w_i as one of its disjuncts. The else-branch of this statement consists of a similar assignment statement $\texttt{X}_\texttt{i} = \texttt{Y}_{\texttt{i,true}}$ for every clause C_i that contains the negated literal $\overline{w_i}$ as one of its disjuncts.

Thus, there exists a one-to-one correspondence between execution paths through the m if-then-else statements and possible truth assignments to the m logical variables, where we associate the then-branch of the i-th if-statement with an assignment of true to logical variable w_i. It should be clear that after execution through any path, $\texttt{X}_\texttt{i}$ points to the same object as $\texttt{Y}_{\texttt{i,true}}$ iff the corresponding truth assignment makes clause C_i to evaluate to true.

We now append the following code fragment:

```
S = new T();
Y_{1,true}.f = S;
Y_{2,true}.f = X_1.f; Y_{1,true}.f = new T();
Y_{3,true}.f = X_2.f; Y_{2,true}.f = new T();
...
Y_{n,true}.f = X_{n-1}.f; Y_{n-1,true}.f = new T();
R = Y_{n,true}.f;
```

Now, consider any execution path through the whole program that corresponds to a truth assignment that makes the entire formula true. Then, a pointer to the object created by the statement $\texttt{S} = \texttt{new T}()$; will be successively copied through every $\texttt{Y}_{\texttt{i,true}}\texttt{.f}$ and then finally to R, causing S and R to be aliased at the end of the program. Conversely, it can be verified that an execution path will cause S and R to be aliased to each other at the end of the program only if the path corresponds to a truth assignment that makes the given 3-SAT formula true.

Hence, R and S may alias each other at the end of the program iff the given 3-SAT formula is satisfiable.

Note that the program generated above has an aliasing width of three (i.e., no more than three pointers point to the same object at any point during program execution). In particular, the assignments $\texttt{Y}_{\texttt{i,true}}\texttt{.f}$ = new T(); guarantee that no more than 3 pointers could point to S at any given time. □

The following theorem is a straightforward consequence of the above result.

Theorem 38 *Typestate verification is NP-hard for programs with aliasing depth two and aliasing width three.*

7 Conclusion

In this paper we have shown that verification of omission-closed properties is in P and that verification of repeatable enabling sequence properties is NP-complete for acyclic programs and PSPACE-complete in general. We have shown that verification of almost-omission-closed properties is in P for acyclic programs. However, many questions still remain open. E.g., we do not know if verification of almost-omission-closed properties is in P for cyclic programs. Moreover there are properties which do not lie in any of these classes. E.g., consider the property $\texttt{open};\texttt{read}^*$ which generalizes $\texttt{open};\texttt{read}$ by allowing any number of `read` operations. We can adapt the *counting* method of Section 5 to show that verification of $\texttt{open};\texttt{read}^*$ 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 $\texttt{open};\texttt{read}^*$. Finally, there are also other properties such as $(\texttt{lock};\texttt{unlock})^*$ (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.

On a more pragmatic note, we have presented a typestate verification algorithm, for arbitrary typestate properties, that we expect will perform well based on the reasonable assumption that programs tend to have small aliasing width. However, this algorithm is restricted to shallow programs. A natural question is how these ideas can be generalized to do verification for arbitrary programs. One of the primary intuitions behind our verification algorithm (for shallow programs) is that maintaining just the right correlation required between "analysis facts" can be the key to efficient and precise verification: maintaining no correlations (independent attribute analysis) can lead to imprecision, while maintaining all correlations (relational analysis) can lead to inefficiency. The recent work of [28] shows one way to exploit this intuition for verification of arbitrary (i.e. non-shallow) programs as well.

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, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV'00*, July 2000.

[5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] E. W. Dijkstra. *A Discipline of programing*. Prentice-Hall, 1976.

[11] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Shallow finite state verification. Technical Report RC22673, IBM T.J. Watson Research Center, Dec. 2002.

[12] 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.

[13] 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.

[14] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[15] 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.

[16] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[17] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. ACM Symp. on Principles of Programming Languages*, Portland, January 2002.

[18] 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.

[19] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[20] 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.

[21] 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.

[22] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 2001.

[23] 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.

[24] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[25] 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.

[26] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Software Eng.*, 19(5):478–485, May 1993.

[27] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.

[28] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34. ACM Press, 2004.