Accumulation Analysis

Martin Kellogg
University of Washington, Seattle, WA, USA

Narges Shadab
University of California, Riverside, CA, USA

Manu Sridharan
University of California, Riverside, CA, USA

Michael D. Ernst
University of Washington, Seattle, WA, USA

Abstract

A typestate specification indicates which behaviors of an object are permitted in each of the object's states. In the general case, soundly checking a typestate specification requires precise information about aliasing (i.e., an alias or pointer analysis), which is computationally expensive. This requirement has hindered the adoption of sound typestate analyses in practice.

This paper identifies *accumulation typestate specifications*, which are the subset of typestate specifications that can be soundly checked without any information about aliasing. An accumulation typestate specification can be checked instead by an accumulation analysis: a simple, fast dataflow analysis that conservatively approximates the operations that have been performed on an object.

This paper formalizes the notions of accumulation analysis and accumulation typestate specification. It proves that accumulation typestate specifications are exactly those typestate specifications that can be checked soundly without aliasing information. Further, 41% of the typestate specifications that appear in the research literature are accumulation typestate specifications.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Typestate, finite-state property

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.10

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.22`

Funding This research was supported in part by the National Science Foundation under grants CCF-2007024 and CCF-2005889, DARPA contract FA8750-20-C-0226, a gift from Oracle Labs, and a Google Research Award.

Acknowledgements Thanks to Max Willsey, Gus Smith, and the anonymous reviewers for their helpful feedback on early drafts.

1 Introduction

A typestate specification [58] associates a finite-state machine (FSM) with program values of a given type. As a value transitions through the states of the FSM, different operations are enabled or disabled; that is, the FSM encodes a behavioral specification for the type.

A typestate analysis checks that a program follows a typestate specification – that is, the program does not attempt to perform a disabled operation. Typestate analyses are well-studied in the literature, and have been deployed for many purposes, including enforcing a locking discipline [28, 17], verification of Windows device drivers [12], and preventing security vulnerabilities [50]. However, *sound* typestate analyses – those with no false negatives – are rarely deployed in practice; for example, a recent paper [21] describing how AWS has deployed

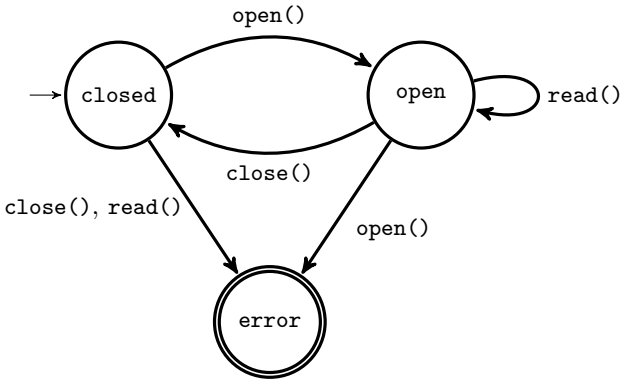

Figure 1 The typestate automaton for a `File` object that can be re-opened after being closed. This typestate specification is not an accumulation typestate system: soundly enforcing it statically requires an alias analysis.

a typestate-based analysis at cloud-scale explicitly omits soundness as a goal. However, building a sound analysis is an important goal: without a soundness guarantee, an analysis might find some bugs, but could not guarantee that no more bugs remain.

A key barrier to sound typestate analyses is the need to reason about aliasing. Consider the classic example [28, 70, 59, 25, 29, 62, 67, 57, 69, 66, 1, 49, 16, 38, 2, 15, 72, 19, 20] of a `File` object, whose typestate is specified in Figure 1, and the following program in a Java-like imperative language:

```
1    File f = new File(...);
2    f.open();
3    File g = f;  // f and g are aliases after this line is executed
4    g.close();
5    f.read();    // an error occurs when this line is executed
```

On line 3, the shared object – which both aliases `f` and `g` refer to – is in the `open` typestate. When `g.close()` is called on line 4, the state of the underlying object transitions to the `closed` state. It is therefore an error when `f.read()` is called on line 5. However, if a static typestate analysis analyzing this program does not consider that `f` and `g` are aliased, then the analysis's estimate of `f`'s typestate does not transition to the `closed` state, and the analysis unsoundly concludes that the call on line 5 is safe – that is, the analysis suffers from a false negative.

For a sound typestate analysis, there are two high-level approaches to handling aliasing: restrict how the programmer creates aliases (e.g., via ownership types [14, 55] or access permissions [7]), or use a sound inter-procedural may-alias analysis that conservatively over-approximates which program variables might be aliases. In practical imperative programming languages with unrestricted aliasing, inter-procedural may-alias analysis is NP-hard [41], and scaling alias analysis to real programs while maintaining acceptable precision remains an open research problem. State-of-the-art analyses often run for an hour or more on practical programs [60].

In recent work [35, 37], we proposed bespoke *accumulation analyses* that soundly and modularly solve specific problems traditionally addressed with typestate. An accumulation analysis collects operations – corresponding to typestate transitions – that have definitely occurred on a given program expression. For example, an accumulation analysis could check the property "before calling `read()` on a `File`, call `open()`." The accumulation analysis would

record on which expressions `open()` had definitely been called, and forbid calls to `read()` that did not occur via such expressions. Note that this is a weaker property than the full specification in Figure 1 – it does not forbid "read after close" defects.

Unlike a traditional typestate analysis, an accumulation analysis is sound without any aliasing information. This means that checking a specification with an accumulation analysis is cheaper – often by an order of magnitude or more – than checking that same specification with a general-purpose typestate analysis. Further, effective incremental analysis – i.e., modularity – is possible for an accumulation analysis, because no whole-program alias analysis is needed. Practical accumulation analyses do use limited, cheap, local aliasing information to improve precision; see Section 5.1. A practical accumulation analysis using limited aliasing information is sound because no aliasing information at all is required for soundness.

Our prior work argued informally that our accumulation analyses are sound, despite their lack of alias reasoning, due to the monotonicity of the particular typestate properties being checked. However, we neither formalized our arguments nor generalized our arguments beyond the specific problems that we targeted. Though our prior work has demonstrated good empirical results – running quickly and finding many real bugs – its soundness claim relies on accumulation analyses being sound without any aliasing information.

The primary goals of this paper are to prove that accumulation analysis does not require aliasing information, to demarcate exactly those typestate specifications that can be soundly checked via an accumulation analysis, and to explore how common such specifications are. Our hope is that analysis designers facing typestate-like problems in the future can use our work to determine whether the property they are interested in is an accumulation property, and hence could be verified without resorting to an expensive, whole-program alias analysis.

Our contributions are:

- a formal definition of an accumulation analysis (Section 3.1);
- a formal definition of an *accumulation typestate system*, and a proof that the properties checkable via accumulation analysis are all accumulation typestate properties (Section 3.2);
- a proof that a typestate system can be checked soundly by a typestate analysis that does no aliasing reasoning if and only if it is an accumulation typestate system (Section 3.3);
- a literature survey of work on typestate analysis, from which we collected 1,355 typestate specifications and determined that 41% of them are accumulation typestate specifications (Section 4); and
- a discussion of the practical issues related to implementing a useful accumulation analysis, and an implementation of a generic accumulation analysis (Section 5).

2 Background: What Is Typestate?

In a standard type system, the type of an expression is immutable throughout the program and the set of operations available on the expression is correspondingly immutable. However, type systems fail to capture the behavioral specifications of many real-world objects that change over time. For example, a chess pawn might become a queen and gain new movement operations, a caterpillar might become a chrysalis and lose the ability to crawl before eventually becoming a butterfly and gaining the ability to fly, or a `File` might be opened and gain the ability to be read. In each of these examples, the logical identity of the object stays the same, but its state – and what that state enables it to do – changes. Typestate [58] extends types to account for possible state changes by encoding the various states and behaviors of a type as a finite-state machine – the typestate automaton for that type. Formally:

▶ **Definition 1.** *A **typestate automaton** $A = (\Sigma, S, s_0, \delta, e)$ for type τ is a finite-state machine. The language Σ is the set of operations, such as method calls, that can be performed on τ. The states S are called* typestates*; $s_0 \in S$ is the initial state. The edges defined by the transition table δ are called transitions and correspond to the effect of operations. There is a distinguished error state $e \in S$. Each typestate has $k = |\Sigma|$ outgoing transitions; none, some, or all of these transitions may be to the error state e or may be self-loops. The error state e has only self-loops – that is, the error state is a trap state.*

At every step during the execution of a program, each value/object of type τ is in one of the typestates of the typestate system.

▶ **Definition 2.** *An **operation** is an event that may cause an object to change state. Every type has a set of operations that can be performed on it, but not all operations are necessarily legal in all states. Traditionally, operations are method calls. However, they can be generalized to include any other event, such as assigning a field or a reference going out of scope.*

Without loss of generality, we represent typestate automata as having no distinguished accepting states (or, equivalently, all non-error states are accepting). If a typestate automaton were to have one or more accepting states, we could transform it to have no accepting states but encode the same behavioral specification in the following way: add a "go out of scope" transition to each typestate; in accepting states (and the error state), this is a self-loop transition, but in non-accepting states, this is a transition to the error state.

▶ **Definition 3.** *A **typestate system** is the pair of a typestate automaton and the corresponding type τ whose safe usage it encodes.*

As an example of a typestate system, Figure 1 shows the automaton, and the type is `File`. Note how each edge is labeled with the corresponding operation. A double circle around the state represents the distinguished error state e. We always draw all transitions, with the exception of those from the error state (which are, by definition, always self-loops).

This paper considers only static typestate analyses. Dynamic run-time monitoring to detect typestate violations exists, but a run-time monitor – like any dynamic analysis – cannot prevent errors before they happen. See Section 6 for more details on related techniques that are outside the scope of the present work.

3 Definitions and Proofs

This section has three goals. First, Section 3.1 formally defines accumulation analysis in a way that is consistent with prior work. Second, Section 3.2 defines an *accumulation typestate system* and shows that every accumulation analysis has a corresponding accumulation typestate system. Finally, Section 3.3 proves that accumulation typestate systems are exactly those typestate systems that can be soundly checked by a static typestate analysis with no aliasing information – that is, a typestate-like analysis that assumes that no aliasing occurs in the program.

3.1 Accumulation Analysis

First, we formalize the notion of an accumulation analysis, as used in prior work [35, 37]:[1]

[1] Our definition is consistent with but not identical to the definitions used in prior work. See Section 6.1.

▶ **Definition 4.** *An **accumulation analysis** is a static program analysis that approximates, for each in-scope expression x of type τ at each program point, a set of operations S that have definitely occurred on the value to which x refers.*

*An accumulation analysis has one or more **goals**. A goal is a pair $\langle g, E \rangle$ where g is the **goal operation** and E is a set of **enabling operations**.*

Informally, an accumulation analysis enforces that a goal operation g does not occur until after every enabling operation $e \in E$ for g has already occurred.

An operation in an accumulation analysis is defined identically to an operation in a typestate automaton (Definition 2).

▶ **Definition 5.** *A **sound accumulation analysis** must issue an error if some goal operation may occur before its enabling operations. More formally, it must issue an error if, for some expression x of type τ and some operation g, both of the following are true:*

1. *There exists at least one goal $\langle g, \_ \rangle$ – that is, g is a goal operation.*
2. *There exists an execution of the program where the set of operations S that have actually occurred on the value of x before an occurrence of g on x is not a superset of one of the enabling sets for g. That is, where there does not exist some goal $\langle g, E \rangle$ such that $S \supseteq E$.*

Intuitively, a sound accumulation analysis is "accumulating" enabling operations, and once everything in the enabling set is accumulated, there is no way to "disable" the goal operation. For example, if g is a goal operation for some goal $\langle g, E \rangle$, an object must first perform some set of operations to make g legal (i.e., the operations in E), and once g becomes legal, it *stays* legal.

Note that soundness, as in Definition 5, only precludes false negative warnings. It says nothing about whether the accumulation analysis might issue a false positive, and a trivially-sound "accumulation analysis" could simply issue an error any time a goal operation might be executed. In practice, a useful accumulation analysis tracks whether the transitions in an enabling set have occurred, and it permits the goal operation if they have.

Note that if an accumulation analysis has multiple goals, their goal operations may or may not be the same. Multiple goals with the same goal operation are useful to express disjunctive specifications. For example, prior work [35] used the disjunctive specification "call either `withOwners()` or `withImageIds()` before calling `describeImages()`."

3.2 Relationship Between Typestate and Accumulation

Next, we need to describe the relationship between a typestate system and an accumulation analysis. As an aid to doing so, we introduce the following:

▶ **Definition 6.** *An **error-inducing sequence** in a typestate automaton T is a sequence of transitions $S = t_1, \ldots, t_i$ such that T is in the error state after all transitions in S are applied (and not before).*

▶ **Definition 7.** *An **accumulation typestate system** is a typestate system such that for any error-inducing sequence $S = t_1, \ldots, t_i$, all subsequences (including both contiguous and non-contiguous subsequences) of S that end in t_i also result in the typestate automaton being in the error typestate. That is, all subsequences of S that end in t_i are also error-inducing.*

Intuitively, an accumulation typestate system is any typestate system whose error-inducing paths are closed under subsequence so long as the final error-inducing operation is held constant. That is, removing operations from the beginning or middle of an error-inducing sequence always produces another error-inducing sequence.

Algorithm 1 A decision procedure for checking whether or not a given typestate automaton T is an accumulation typestate automaton. The complexity of the algorithm is $O(max(n \log n, en))$ where n is the number of states and e is the number of edges.

```
1:  procedure IsAccumulation(T)
2:      // FindErrorInducingTransitions returns all transitions into the error state.
3:      U ← FindErrorInducingTransitions(T)
4:      // E and E_subseq are finite-state automata. ∀ X, Union(∅, X) = X.
5:      E ← ∅
6:      E_subseq ← ∅
7:      for u_i ∈ U do
8:          // ErrorInducingAutomatonVia is an automaton that accepts a sequence of
9:          // transitions S iff S followed by u_i causes an error in the original automaton T.
10:         // Its implementation contains two steps: (1) modify T so that states from which
11:         // u_i is error-inducing are accepting, and then (2) minimize and return the result.
12:         E_i ← ErrorInducingAutomatonVia(u_i, T)
13:         // Subsequences produces the automaton that accepts the subsequence language
14:         // for the input automaton, which Higman's theorem guarantees exists.
15:         E_subseq(i) ← Subsequences(E_i)
16:         // Concat produces an automaton that accepts iff it receives a sequence
17:         // that the input automaton accepts followed by the concatenated transition.
18:         E ← Union(E, Concat(E_i, u_i))
19:         E_subseq ← Union(E_subseq, Concat(E_subseq(i), u_i))
20:     // AcceptSameLanguage is true iff the two automata accept the same language.
21:     return AcceptSameLanguage(E, E_subseq)
```

Note that a vacuous sound typestate analysis such as "issue an error at every program statement" is trivially enforcing an accumulation typestate system. The typestate automaton that such an analysis enforces only has transitions to the error state, so all sequences are error-inducing.

This definition leads to a decision procedure (Algorithm 1) for determining whether a given typestate system T is an accumulation typestate system. Consider all error-inducing operations $U = \{u_1, \ldots, u_n\}$. The elements of U are the final transitions for every error-inducing sequence in the automaton of T. For any $u_i \in U$, let E_i be the language[2] of the error-inducing sequences of operations in T that end in u_i, with the last transition removed (i.e., the u_i transition that leads to the `error` typestate). Let $E_{subseq(i)}$ be the language of subsequences of E_i. Let $E = \bigcup_{i=1}^{n} E_i * u_i$ and $E_{subseq} = \bigcup_{i=1}^{n} E_{subseq(i)} * u_i$. That is, E is the union of all error-inducing paths in T, and E_{subseq} is the union of all subsequences of error-inducing paths in T that end in the same transition as the corresponding error-inducing path from which they were derived. By Definition 7, if and only if E and E_{subseq} recognize the same language, T is an accumulation typestate system.

It is easy to check whether E and E_{subseq} recognize the same language, because both are regular. E is regular, because it can be recognized by T's automaton, if the error typestate is converted to an accepting state. Since there are finitely-many operations, any E_i and $E_{subseq(i)}$ have a finite alphabet. Higman's theorem [31] says that the language of the subsequences

[2] Throughout, we will abuse notation and refer to both languages and their corresponding language-recognizers by the same name.

of any language over a finite-alphabet is regular. Therefore, any $E_{subseq(i)}$ is also regular. E_{subseq} is regular because regular languages are closed under both union and concatenation. So, the procedure for checking whether a typestate automaton is an accumulation typestate automaton is as easy as checking whether the two finite state machines for E and E_{subseq} recognize the same language.

▶ **Theorem 8.** *Every accumulation analysis has a corresponding accumulation typestate system.*

Proof. Consider some accumulation analysis *acc* with goals $(g_1, E_1), \ldots, (g_n, E_n)$ over type τ. The corresponding accumulation typestate system is the pair of the type τ and the accumulation typestate automaton constructed by the following procedure:

1. Create an error state `error` with a self-loop transition for each operation on τ.
2. Let $\mathcal{P}_E$ be the powerset of E, where $E = \bigcup_{i=1}^{n} E_i$ is the union of the enabling sets $E_1, \ldots, E_n$. For each element S of $\mathcal{P}_E$, create a corresponding state and label it with S. Note that S refers to both the member of $\mathcal{P}_E$ and the corresponding state.
3. Make the state that is labeled by the empty set be the start state of the automaton.
4. For each state $S \in \mathcal{P}_E$ and for each transition $t_e \in E$, add a transition from state S to state $S \cup \{t_e\}$ labeled t_e. (This transition might be a self-loop.)
5. Let $G = \{g_1, \ldots, g_n\}$ be the set of goal transitions. For each element g_i of G and for each state $S \in \mathcal{P}_E$:
   If there exists a goal $\langle g_i, E_i \rangle$ such that $E_i \subseteq S$,
   then add a self-loop transition to S labeled g_i if it does not already have a transition labeled g_i. (It might have such a transition if g_i is both an enabling transition and a goal transition.)
   Else if such a goal does not exist,
   add a transition from S to the `error` state labeled g_i, removing a transition labeled g_i if one already exists.
6. For each operation t on τ such that $t \notin G$ and $t \notin E$ – that is, for each operation that is neither a goal operation nor an enabling operation – add a self-loop transition labeled t to each non-error state. (Recall that the error state already has self-loop transitions for each operation, added in step 1.)

The resulting accumulation typestate automaton encodes the same behavior as the original accumulation analysis. ◀

Note that this construction is a existence proof, not an efficient translation: it does induce an exponential blowup in the number of states. A practical accumulation analysis does not track states directly – rather, it tracks only the enabling sets – so state explosion is not a problem in practice.

3.3 Soundness Without Aliasing

This section proves that accumulation typestate systems are exactly the typestate systems that are soundly checkable without reasoning about aliasing (i.e., by a *typestate analysis with no aliasing information*, which we will formally define in Definition 14):

▶ **Theorem 9.** *A typestate system $T = (A, \tau)$ is an accumulation typestate system if and only if there exists a typestate analysis with no aliasing information that can soundly check T.*

The high-level intuition behind the proof of Theorem 9 is the consequence of two facts:

- without using aliasing information, a typestate analysis observes only a subsequence of the actual operations that are applied to the object to which some expression refers, and
- accumulation typestate automata are exactly those that are error-closed under subsequence, when the last transition is held constant.

The formal proof is split into Lemmas 16 and 17 (which are the forward and backward directions of the bi-implication respectively), and appears in Section 3.3.2. Section 3.3.1 defines the supporting machinery of the proof: the language, relevant definitions, etc.

Accumulation analyses as defined in Section 3.1 (and therefore as defined in prior work [35, 37]) are sound without access to aliasing information:

▶ **Corollary 10.** *An accumulation analysis, even without aliasing information, is sound.*

Proof. Convert the accumulation analysis to an accumulation typestate system via the procedure in the proof of Theorem 8. By Theorem 9, the accumulation typestate system can be soundly checked. ◀

An important consequence of the ability to soundly check an accumulation typestate system with *no* aliasing information is that approaches that utilize *limited* aliasing information are also sound. In practice, analyses can compute inexpensive, typically local, alias information to improve precision (i.e., to avoid issuing false positive warnings); see Section 5.1.

3.3.1 Preliminaries

This section introduces the machinery used to prove Theorem 9.

3.3.1.1 Language

We will prove Theorem 9 over a core calculus that represents a simple imperative programming language. This language contains the essential parts of a programming language related to typestate checking and aliasing – method calls, fields, and assignments.

A program P in this language is a statement s of one of the following kinds:

- an assignment: x_i `:=` x_j.
- a field load: x_i `:=` x_j`.`f_k.
- a field store: x_i`.`f_j `:=` x_k.
- a method call: x_i`.`m_j`()`.
- a statement sequence: s_i `;` s_j.

Source code variables range from `x_1` to `x_`n, where n is some positive integer. Statements may only refer to variables in that range. There is a single type T. Each variable refers to a *value* – that is, a particular object instance – of type T. We use $x_i, x_j, \ldots$ as metavariables for arbitrary variables in the range `x_1`,...,`x_`n. T has methods `m_1` to `m_`k and a corresponding typestate automaton A whose k operations are exactly the methods `m_1` to `m_`k. A method call statement can only refer to methods in T. We use $m_i, m_j, \ldots$ as metavariables for arbitrary methods in T. Each object of type T has fields `f_1` to `f_`m, where m is some positive integer. Load and store statements may only refer to fields in this range. Each field refers to some value of type T. We use $f_i, f_j, \ldots$ as metavariables for arbitrary fields in T.

To simplify the presentation and proofs, this language lacks conditionals, loops, method bodies, return values, etc. – which makes precise alias and typestate analysis trivial. However, our algorithms are general (they do not take advantage of the straight-line nature of the code) and can be extended to a richer language without changing the essence of the proof. Section 5.2 discusses practical concerns when implementing an accumulation analysis for a real programming language.

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j \Downarrow \langle \rho[x_i \mapsto \rho(x_j)], \sigma, \tau \rangle} \text{ ASSIGN}$$

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j.f_k \Downarrow \langle \rho[x_i \mapsto \sigma(\langle \rho(x_j), f_k \rangle)], \sigma, \tau \rangle} \text{ LOAD}$$

$$\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i.f_j := x_k \Downarrow \langle \rho, \sigma[\langle \rho(x_i), f_j \rangle \mapsto \rho(x_k)], \tau \rangle} \text{ STORE}$$

$$\frac{\langle \rho, \sigma, \tau \rangle \vdash t' = succ(\tau(\rho(x_i)), m_j, A) \qquad t' \neq \texttt{error}}{\langle \rho, \sigma, \tau \rangle \vdash x_i.m_j() \Downarrow \langle \rho, \sigma, \tau[\rho(x_i) \mapsto t'] \rangle} \text{ CALL}$$

$$\frac{\langle \rho, \sigma, \tau \rangle \vdash s_i \Downarrow \langle \rho', \sigma', \tau' \rangle \qquad \langle \rho', \sigma', \tau' \rangle \vdash s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle}{\langle \rho, \sigma, \tau \rangle \vdash s_i; s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle} \text{ SEQ}$$

Figure 2 The big-step dynamic semantics of the language expressed as inference rules. The notation $\mu[x \mapsto y]$ means that the map μ is updated so that x maps to y. $M \vdash s \Downarrow M'$ means that executing statement s in machine-state M results in machine-state M'.

3.3.1.2 Dynamic Semantics

To execute a program, we maintain a *machine state* $\langle \rho, \sigma, \tau \rangle$ composed of an environment (ρ) mapping each variable to a value of type T, a store (σ) mapping each value–field pair to a value, and a typestate store (τ) mapping each value to a typestate in A. The initial environment maps each x_i to a distinct value v_j. The initial store maps each value–field pair $\langle v_i, f_j \rangle$ to a distinct value v_k. The initial typestate store maps each value v_i to the start typestate s_0 of A.[3] Executing a statement in machine state $\langle \rho, \sigma, \tau \rangle$ either produces an updated machine state $\langle \rho', \sigma', \tau' \rangle$, or it terminates the program in an error if any value's entry in the typestate store would be A's `error` typestate. The dynamic semantics (Figure 2) are as follows:

- For an assignment x_i `:=` x_j, produce a new machine state with an updated environment: $\rho'(x_i) = \rho(x_j)$ (rule ASSIGN).
- For a field load x_i `:=` x_j`.`f_k, produce a new machine state with an updated environment: $\rho'(x_i) = \sigma(\rho(x_j), f_k)$ (rule LOAD).
- For a field store x_i`.`f_j `:=` x_k, produce a new machine state with an updated store: $\sigma'(\rho(x_i), f_j) = \rho(x_k)$ (rule STORE).
- For a call x_i`.`m_j`()`, let $t' = succ(\tau(\rho(x_i)), m_j, A)$. That is, t' is the successor typestate in A when transition m_j occurs in the current typestate of the value that x_i is a reference to. If t' is not the `error` typestate, produce a new machine state with an updated typestate store: $\tau'(\rho'(x_i)) = t'$ (rule CALL). If t' is the `error` typestate, the semantics "get stuck" and the program terminates in an error.
- For a sequence s_i `;` s_j, first execute s_i. If the program terminates in an error while executing s_i, the semantics for the sequence statement "get stuck." Otherwise, let $\langle \rho', \sigma', \tau' \rangle$ be the machine state after executing s_i. Execute s_j in $\langle \rho', \sigma', \tau' \rangle$ (rule SEQ).

[3] Initializing all variables before a program starts simplifies the language by removing the need for a `new` expression.

3.3.1.3 Sound Typestate Analysis

▶ **Definition 11.** *A **typestate analysis** is a static program analysis. Its inputs are a program P and a typestate system $T = (A, \tau)$. It reports call statements within P that may cause the program to terminate in an error when running P.*

▶ **Definition 12.** *A typestate analysis is **sound** if it reports each call statement that causes the program to terminate in an error at run time in any execution of the program.*

3.3.1.4 Representation of Aliasing

Suppose that a typestate analysis has access to two oracle functions $\mathit{MustOracle}(x_i, s)$ and $\mathit{MayOracle}(x_i, s)$ for aliasing information. Each oracle takes a variable x_i and a program statement s and returns a list of *names* – variables or arbitrarily-nested field load expressions – that the input variable must (respectively, may) alias before the given statement.

MustOracle returns a list of names that definitely do alias x_i at s. More formally, for a sound oracle, if the list returned by $\mathit{MustOracle}(x_i, s)$ contains x_j, then x_i and x_j are definitely aliased before statement s on all executions. If the list does not contain x_j, then x_i and x_j may or may not be aliased before s. A trivial *MustOracle* that always returns an empty list is sound.

MayOracle returns a list of names that *might or might not* alias x_i at s. More formally, for a sound oracle, if the list returned by $\mathit{MayOracle}(x_i, s)$ does not contain x_j, then x_i and x_j are definitely not aliased before statement s on all executions. If the list does contain x_j, then x_i and x_j may or may not be aliased before s. A trivial *MayOracle* that always returns every in-scope name in the program is sound.

These oracles can represent an external alias analysis, an on-demand alias analysis, aliasing tracking built into the typestate analysis, etc. If the oracles are sound, then for all x_i and s, $\mathit{MustOracle}(x_i, s) \subseteq \mathit{MayOracle}(x_i, s)$. For a traditional typestate analysis (as defined in section 3.3.1.5) to be sound for an arbitrary typestate system such as the `File` example in Figure 1, both oracles must be sound.[4]

3.3.1.5 Definition of Typestate Analysis

A typestate analysis is a fixpoint analysis that can be viewed as a dataflow analysis or an abstract interpretation. It operates by maintaining a set of *abstract stores*, one for each program point. An abstract store is a map from names to sets of estimated typestates. We write $\phi_s(x_i)$ for the estimated typestates of name x_i before program statement s, and $\phi'_s(x_i)$ for those after. For any sequencing statement $r\,;s$, for all x_i, $\phi'_r(x_i) = \phi_s(x_i)$. The notation $\hat{\phi}_s(x_i.*)$ means all names in ϕ_s that begin with x_i.

At the beginning of the analysis, at every program point, the abstract store maps all names[5] to the set containing only the start state s_0 of the typestate automaton A. Then, the analysis processes each statement s using the following rules (which also appear in Figure 3) until the set of abstract stores reaches a fixpoint:

---

[4] For the language of section 3.3.1.1, it is trivial to construct a sound alias analysis that never includes a name in the result of a *MayOracle* query unless the corresponding *MustOracle* query would also include that name. In a richer programming language, the *MayOracle* is necessary to handle analysis imprecision and control flow joins.

[5] An analysis may use widening, abstraction, or iterative expansion of maps to handle the fact that the set of names is infinite.

$$\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j/x_i] \wedge T'_{n'} = \phi_s(n') \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]}{\phi_s \vdash x_i := x_j \Downarrow \phi'_s} \text{ TS-ASSIGN}$$

$$\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j.f_k/x_i] \wedge T'_{n'} = \phi_s(n') \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s} \text{ TS-LOAD}$$

$$\frac{\begin{array}{c}\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.f_j.*), n' = n[x_k/x_i.f_j] \wedge T'_{n'} = \phi_s(n') \wedge \\ A_n^{must} = \mathit{MustOracle}(n,s) \wedge A_n^{may} = \mathit{MayOracle}(n,s) \\ \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.f_j.*), n \mapsto T'_{n'}][\forall a_n \in A_n^{must}, a_n \mapsto T'_{n'}] \\ [\forall b_n \in A_n^{may} - A_n^{must}, b_n \mapsto T'_{n'} \cup \phi_s(b_n)]\end{array}}{\phi_s \vdash x_i.f_j := x_k \Downarrow \phi'_s} \text{ TS-STORE}$$

$$\frac{\begin{array}{c}\phi_s \vdash T = \phi_s(x_i) \qquad T' = \bigcup_{t \in T} \mathit{succ}(t, m_j, A) \\ A^{must} = \mathit{MustOracle}(x_i, s) \qquad A^{may} = \mathit{MayOracle}(x_i, s) \\ \phi'_s = \phi_s[x_i \mapsto T'][\forall a \in A^{must}, a \mapsto T'][\forall b \in A^{may} - A^{must}, b \mapsto T' \cup \phi_s(b)]\end{array}}{\phi_s \vdash x_i.m_j() \Downarrow \phi'_s} \text{ TS-CALL}$$

$$\frac{\phi_s \vdash s_i \Downarrow \phi'_{s_i} \qquad \phi'_{s_i} = \phi_{s_j} \qquad \phi_{s_j} \vdash s_j \Downarrow \phi'_s}{\phi_s \vdash s_i; s_j \Downarrow \phi'_s} \text{ TS-SEQ}$$

Figure 3 Inference rules for a traditional, sound typestate analysis. Each rule applies to some statement s, which appears in the consequent. The notation $x[y/z]$ means "x with each z replaced by y." The notation $\hat{\phi}_s(x_i.*)$ means all names in ϕ_s that begin with x_i.

- For an assignment x_i := x_j, for each $n \in \hat{\phi}_s(x_i.*)$, let $n' = n[x_j/x_i]$ – that is, n' is n with its x_i replaced by x_j – and let $T'_{n'} = \phi_s(n')$, the abstract value of n' in the pre-state. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$ (rule TS-ASSIGN). For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.*)$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.
- For a load statement x_i := $x_j.f_k$, for each $n \in \hat{\phi}_s(x_i.*)$, let $n' = n[x_j.f_k/x_i]$ and let $T'_{n'} = \phi_s(n')$. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$ (rule TS-LOAD). For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.*)$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.
- For a store statement $x_i.f_j$:= x_k, for each $n \in \hat{\phi}_s(x_i.f_j.*)$, let $n' = n[x_k/x_i.f_j]$ and let $T'_{n'} = \phi_s(n')$. Then, for each n and its n' and $T'_{n'}$, the analysis performs the following steps (rule TS-STORE):
  1. The analysis updates the abstract store after s so that n is mapped to $T'_{n'}$: $\phi'_s(n) := T'_{n'}$.
  2. The analysis queries $\mathit{MustOracle}(n, s)$ (call the result A_n^{must}). For each $a_n \in A_n^{must}$, the analysis performs a *strong update* to the abstract store: $\phi'_s(a_n) := T'_{n'}$.
  3. The analysis queries $\mathit{MayOracle}(n, s)$ (call the result A_n^{may}). For each element b_n in $A_n^{may} - A_n^{must}$ – that is, variables that may be aliases but are not guaranteed to be aliases – the analysis performs a *weak update* to the abstract store so that it maps b_n to $T'_{n'} \cup \phi_s(b_n)$: $\forall b_n \in A_n^{may} - A_n^{must}, \phi'_s(b_n) := T'_{n'} \cup \phi_s(b_n)$.

For all other names m in ϕ_s where $m \notin \hat{\phi}_s(x_i.f_j.*) \wedge \forall A_n^{may}, m \notin A_n^{may}$, the analysis copies the entry from the previous abstract store: $\phi'_s(m) := \phi_s(m)$.

- For a call statement $x_i.m_j()$, let $T' = \bigcup_{t \in \phi_s(x_i)}$. The analysis performs the following steps (rule TS-CALL):
  1. If any $t' \in T'$ is `error`, the analysis reports an error for the statement. Note that while the dynamic semantics (Figure 2) do not permit any value to be in the `error` typestate (the program crashes instead), this analysis approximates the semantics statically.
  2. The analysis updates the abstract store so that $\phi'_s(x_i) := T'$.
  3. The analysis queries $MustOracle(x_i, s)$ (call the result A^{must}). For each $a \in A^{must}$, the analysis performs a strong update to the abstract store: $\phi'_s(a) := T'$.
  4. The analysis queries $MayOracle(x_i, s)$ (call the result A^{may}). For each $b \in A^{may} - A^{must}$, the analysis performs a weak update to the abstract store: $\phi'_s(b) := T' \cup \phi_s(b)$.
- For a sequence $s = s_i \ ; \ s_j$, the analysis first analyzes s_i, and then analyzes s_j with the resulting abstract store (rule TS-SEQ)). (Note that the analysis does not terminate in the case of an error, but keeps reporting errors on subsequent statements.)

This standard formulation of a traditional typestate analysis is sound for any arbitrary typestate system, as long as its aliasing oracles are sound:

▶ **Theorem 13.** *A traditional typestate analysis is sound if its MustOracle and MayOracle functions return sound results.*

Proof. By co-induction on the dynamic semantics (Figure 2) and the rules for a traditional typestate analysis (Figure 3). The key invariant is that the actual typestate to which a name refers on any particular execution at some statement is always in the abstract store. ◀

3.3.1.6 Typestate Analysis with No Aliasing Information

▶ **Definition 14.** *A **typestate analysis with no alias information** is a typestate analysis whose MustOracle and MayOracle functions return empty lists for all arguments.*

Intuitively, a typestate analysis "with no alias information" assumes that no aliasing occurs in the program – even when making such an assumption is unsound.

A typestate analysis with no alias information has a simpler method call rule: it never updates its abstract store in response to an aliasing query, so steps 3 and 4 may be omitted. Similarly, there is a simpler store rule: only the $n \in \hat{\phi}_s(x_i.f_j.*)$ need to be updated, because all *MayOracle* and *MustOracle* queries (unsoundly) return false.

Informally, having no aliasing information means that the analysis might not be aware that one or more transitions have occurred on the value to which some expression refers, because those operations occurred via an alias. That is, the analysis's estimate of the typestate of an expression that actually refers (at run time) to a value v in typestate t is must include a typestate reachable by a subsequence of the sequence of transitions that results in $\tau(v)$ being t. Stated more formally:

▶ **Lemma 15.** *Let $R = \phi_s(x_i)$ be the set of estimated typestates produced by a typestate analysis with no aliasing information for a variable x_i before a statement s. Let S be the trace of an arbitrary execution leading up to some occurrence of s, and let $t = \tau(\rho(x_i))$ be the typestate of the actual value to which x_i refers before that occurrence of s. Applying S to the automaton leads to typestate t. There exists a typestate $r \in R$ such that applying some subsequence of S leads to r. That is, there is some estimated typestate $r \in R$ that is reachable by a subsequence of the transitions that lead to t.*

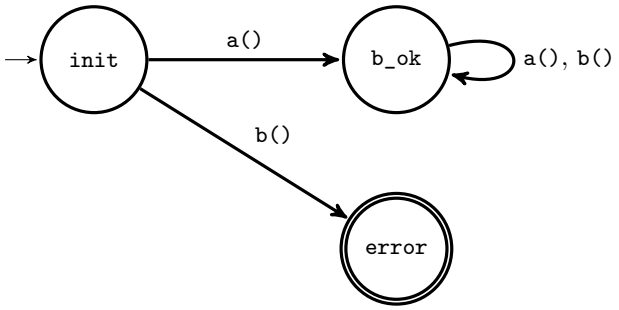

Figure 4 An accumulation typestate automaton for the property "call a() before calling b()".

$$\frac{\phi_s \vdash \phi'_s = \phi_s[x_i \mapsto s_0]}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s} \text{ TS-LOAD-FIX}$$

Figure 5 A modified load rule for a typestate analysis with no aliasing information, which preserves Lemma 15. s_0 is the start state of the automaton A being checked.

Stated another way, Lemma 15 says that for every possible trace S through the program that reaches s, there is at least one $r \in R$ that "corresponds to" S, in the sense that r is reachable by a subsequence of S.

Lemma 15 is not quite true of a typestate analysis as defined in Figure 3: field loads do not necessarily preserve it. Because the store rule is unsound due to the unsoundness of the aliasing oracles, the entry in the abstract store for a given field may not actually be related to the value to which that name refers, due to possible aliasing. For example, consider the following program, being analyzed with respect to the "only call b() after a()" typestate automaton in Figure 4 (note that "Estimated state" and "Actual state" columns only show entries for names that are relevant to the problem):

Program	Estimated state (ϕ_s)[6]	Actual state (τ)[7]
`x2 = x1`	{`x1.f`↦`init`, `x2.f`↦`init`}	{`x1.f`↦`init`, `x2.f`↦`init`}
`x3.a()`	{`x1.f`↦`init`, `x2.f`↦`init`, `x3`↦`b_ok`}	{`x1.f`↦`init`, `x2.f`↦`init`, `x3`↦`b_ok`}
`x1.f = x3`	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x3`↦`b_ok`}	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x3`↦`b_ok`}
`x2.f = x4`	{`x1.f`↦`b_ok`, `x2.f`↦`init`}	{`x1.f`↦`init`, `x2.f`↦`init`}
`x5 = x1.f`	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x5`↦`b_ok`}	{`x1.f`↦`init`, `x2.f`↦`init`, `x5`↦`init` }
`x5.b()`	{`x1.f`↦`b_ok`, `x2.f`↦`init`, `x5`↦`b_ok`}	{`x1.f`↦`init`, `x2.f`↦`init`, `x5`↦`init` }

This program (left side of the table above) leads to Lemma 15 being untrue at the final statement, because the actual state of `x5` (`init`) is not reachable from the estimated state (`b_ok`). The key issue is aliasing: `x1` and `x2` are aliases, so `x1.f` and `x2.f` actually refer to the same value. When `x2.f` is re-assigned to `x4`, the actual value to which `x1.f` refers changes – but with no aliasing information, the typestate analysis is unaware, leading to the problem.

Note that this problem applies to arbitrary typestate systems: both accumulation typestate systems and non-accumulation typestate systems. Lemma 15 discusses both.

[6] Entries in ϕ_s are single-element sets. For simplicity of presentation, set notation has been elided.
[7] Keys in τ are values. For simplicity of presentation, the necessary lookups in ρ and σ have been elided.

There is a simple solution to this problem that makes Lemma 15 hold for a typestate analysis with no aliasing information: update the load rule so that the analysis assumes that all loads return a value whose typestate is the start state of the automaton (rule TS-LOAD-FIX in Figure 5).

This rule trivially preserves Lemma 15 for field loads, and corresponds with how accumulation analyses handle field loads in practice (see Section 5.2). Our proof assumes this simpler load rule for the typestate analysis with no aliasing information. However, note that this rule would make a traditional typestate analysis unsound (i.e., this rule makes Theorem 13 untrue): in an arbitrary typestate analysis, the start state is not necessarily a safe default assumption. A useful property of accumulation typestate automata, however, is that every operation which might ever lead to an error on any path must necessarily lead to an error from the start state – otherwise, the definition of accumulation typestate automaton could not be met when considering the empty subsequence.

We now prove Lemma 15 (see Appendix A for the full proof):

Proof. By co-induction on the dynamic semantics and the rules for a typestate analysis with no aliasing information. The interesting cases are method calls, assignments, and loads. Method calls preserve the inductive invariant via the inductive hypothesis. Assignments preserve the inductive invariant because the left-hand side's estimate is updated to the right-hand side's estimate, which also preserves the invariant by the inductive hypothesis. Loads preserve the inductive invariant only because of the modified rule described above, which says that after a load, the estimate is always the start state, which trivially preserves the invariant. ◀

3.3.2 Proof of Theorem 9

The proof is split into two parts – the forwards and backwards direction of the bi-implication, which are Lemmas 16 and 17, respectively.

▶ **Lemma 16.** *T is an accumulation typestate system $\implies$ there exists a sound typestate analysis with no aliasing information that can check T.*

Proof. The proof is by contradiction. Suppose that an arbitrary typestate analysis with no aliasing information (as defined by Definition 14) for an accumulation typestate system T is unsound. That is, suppose that it fails to issue an error at some method call statement $s = x_i.m_j()$, but the program terminates in an error in some execution e, because $\tau(\rho(x_i))$ after s would be `error`.

Let $v_i = \rho(x_i)$. That is, x_i actually refers to v_i at[8] s on execution e. m_j must be the transition that would lead v_i to enter the error typestate at the call $x_i.m_j()$, because the program would have already terminated if some other transition might have caused v_i to enter the error state before s was reached. Let $R' = \phi'_s(x_i)$ be the analysis's estimate of the possible typestates of x_i after the call statement is executed. Because the analysis did not issue an error at s, R' must not contain the `error` typestate.

Since R' does not contain the error typestate after observing m_j, then m_j must have been a legal transition on each typestate in the analysis' pre-state estimate $R = \phi_s(x_i)$. By Lemma 15, there is some typestate $r \in R$ that is reachable via some subsequence of the transitions that led to the actual typestate $t = \tau(\rho(x_i))$ that v_i was in during e before transition m_j was applied.

[8] s must be a method call statement, so v_i is the same before and after s.

The typestate r is reachable by a subsequence of the sequence of transitions that actually occurred on v_i that led it to reach t, but m_j is a legal transition in r. This is a contradiction: m_j must be both an error-inducing and a legal transition in r. m_j must be an error-inducing transition in r by the definition of an accumulation typestate system (Definition 7): m_j must be an error-inducing transition in typestates reachable via subsequences of the transitions that lead to t, including r. But, m_j must also be a legal transition in r because the analysis did not issue an error when its estimate included r. Since one transition cannot be both error-inducing and legal, by contradiction, the analysis must have been sound. ◀

▶ **Lemma 17.** *T is an accumulation typestate system $\Longleftarrow$ there exists a sound typestate analysis with no aliasing information that can check T.*

Proof. The proof is by contradiction. Suppose that there is a typestate analysis with no aliasing information that can soundly check a typestate system T that is not an accumulation typestate system. Since T is not an accumulation typestate system, there exists some sequence of transitions $S = t_1, \ldots, t_i$ that ends in an error typestate that has a subsequence S' that ends in t_i that does not end in an error typestate. Let D be the difference between S' and S: the sequence of transitions that appear in S but do not appear in S'.

Construct a program P with two variables $x_{S'}$ and x_D. The first statement in P is x_D `:=` $x_{S'}$, which aliases these expressions. Then augment the program in the following manner: for each transition $t \in S$, if t is an element of S', then add the statement $x_{S'}.t()$ to P. Otherwise, add the statement $x_D.t()$ to P.

Because $x_{S'}$ and x_D were aliased by P's first statement, we know that they both point to a single value v to which every transition in S has been applied by the end of P; thus, P terminates in an error when the final transition t_i is applied. However, no error is issued: the analysis will not issue an error for $x_{S'}.t_i()$, which is the program statement that causes the error, because the sequence R that was applied to $x_{S'}$ is a legal sequence of transitions (and the error-inducing transition t_i is guaranteed to be in S', not in D, by definition). This is a contradiction of our original premise that a typestate analysis with no aliasing information could soundly check T: an error-inducing transition (t_i) occurs, but the analysis with no aliasing information fails to issue an error. Thus, T must have been an accumulation typestate system. ◀

3.4 Discussion: Accumulating Sets vs. Accumulating Subsequences

Section 3 uses the term "accumulation" to refer to two subtly different things. Accumulation analyses (Definition 4) compute *sets* of operations. Accumulation typestate systems (Definition 7) are defined by *(sub)sequences* of operations.

Definition 4 of accumulation analysis uses sets because that is how accumulation analysis is defined and implemented in prior work [35, 37]. For an alternate definition of accumulation analysis in terms of subsequences, each goal operation would have an enabling sequence rather than an enabling set. Implementing an accumulation analysis based on this alternate definition would allow us to check "accumulation-like" properties that cannot be expressed as sets. For example, such an analysis could soundly check a property such as "call `a()` at least twice before calling `b()`" (i.e., a goal transition enabled by counting) or a property such as "call `a()` and `b()`, in that order, before calling `c()`" (i.e., a goal transition enabled by ordering). This generalization of the *concept* of accumulation from the specific accumulation analyses used in prior work is one of our contributions.

In our literature survey (Section 4), we found three specifications with a goal transition enabled by ordering, but we did not find any enabled by counting. For example, in Figure 12 of [56], the authors describe a mined typestate specification for the Java KeyAgreement

API. This API contains a method `generateSecret()`. Calling `generateSecret()` before `init()` and `doPhase()` is an error, so `generateSecret()` is a goal transition. However, `init()` and `doPhase()` also must be ordered: calling `doPhase()` before `init()` is also an error. The other two specifications in the literature (which appear in [56, 22]) that rely on ordering had a similar character to this example: describing some multi-stage initialization property where the initialization steps must be performed in some specific order.

4 Literature Survey

This section aims to answer the research question: **RQ1: What fraction of typestate problems can be solved modularly with an accumulation analysis?**

We will approximate the answer by using the population of typestate problems that appear in the scientific literature. Note that this is likely to be an under-approximation of incidence in practice, because scientific papers usually address the most complex problems.

We performed a literature survey of papers in the research literature since 2000 that contain typestate specifications. We chose the year 2000 because a similar survey [18], which we discuss in section 4.2.2.1, was published in 1999. For each typestate specification that we discovered, we used the decision procedure in Algorithm 1 to determine whether the specification was an accumulation typestate system – and therefore soundly analyzable without any aliasing information by Theorem 9. The vast majority of the papers that we analyzed use typestate for some small number of examples. We report on these papers in aggregate and describe specific, common examples (Section 4.2.1). There are two outliers [18, 4] that reported on categories containing hundreds of specifications, which we discuss in detail (Section 4.2.2).

The remainder of this section details our methodology, discusses the results, and gives examples of specifications that can and cannot be checked via accumulation.

4.1 Methodology

We searched Google Scholar for papers since 2000 whose full-text includes "typestate", resulting in 1,760 hits. (We originally included "type-state" and "type state" as search terms, but discovered no computer science results in the first 100 hits for each that "typestate" did not also return.) We discarded any paper that was not published in the research track of a reputable computer science conference or journal or was duplicative with another paper in the dataset (e.g., for work with both a conference paper and a journal extension, we only included the journal extension), resulting in a set of 187 papers. The authors are familiar with the relevant conferences and journals in programming languages and software engineering, and we used our judgment for these, erring on the side of inclusivity. For conferences or journals outside PL and SE, we included papers in any venue with a CORE ranking of A or A*.

We then examined each of the remaining papers in detail and recorded how many typestate specifications they contained, which specifications those were, and which of the specifications were accumulation typestate systems. When recording which specifications occurred in each paper we examined, we also recorded whether the specifications were duplicates of specifications that appeared in other papers. Among the papers we examined, 102 ($\approx 55\%$ of those examined closely, and $\approx 6\%$ of all Google Scholar hits) contained one or more typestate specifications. The venues that contributed papers with one or more typestate specifications to this study are: ECOOP (12), ESEC/FSE (12), ICSE (12), OOPSLA (10), PLDI (8), ISSTA (7), ASE (6), POPL (5), CCS (4), SAS (4), TOSEM (4), TSE (4), CC (2), ASPLOS (1), CAV (1), EuroSys (1), ICPC (1), IWACO (1), SAC (1), SOSP (1), TOPLAS (1), VMCAI (1), WWW (1).

Table 1 The results of the literature survey. "TSA" stands for "TypeState Automata"; "ATSA" stands for "Accumulation TypeState Automata". All specification counts are without de-duplication.

Dataset	Source	TSA	ATSA	ATSA%
Papers since 2000 with <20 TSAs	101 scientific papers	302	67	22%
Dwyer et al. (1999) [18]	34 papers, tools, students	511	306	60%
Beckman et al. (2011) [4]	4 real Java projects	542	182	34%
Total	All of the above	1355	555	41%

4.2 Results

Table 1 summarizes the results. This paper's artifact[9] contains our analysis of each relevant paper. The artifact also contains a finite-state machine for each typestate problem (as defined in Section 4.2.1 below) we saw and the list of the papers we saw it in.

4.2.1 Papers Containing Examples

These 101 papers contain 302 specifications, with a mean of 3 and a median of 2.

22% of these specifications are accumulation typestate systems. However, there is a significant amount of duplication between the papers in this dataset – many papers use the same few examples of typestate automata to motivate their general work on typestate.

We de-duplicated the typestate automata in these papers by combining instances of the same automaton into a single *typestate problem*: for example, we counted every one of the 19 papers that we observed using the classic `File` example (Figure 1) as a single instance of the `File` typestate problem. Considering problems rather than specifications, we found that these 101 papers only contain 114 problems. Of those 114, 31 are accumulation typestate problems (27%), indicating that there is slightly more duplication among the non-accumulation typestate specifications. Perhaps this is because papers dealing with general typestate analysis want to motivate their use of an alias analysis – which requires at least one non-accumulation typestate example. We discuss this discrepancy further in Section 4.3.

Next, we give the three most common examples of typestate problems that are accumulation and are not accumulation typestate systems.

4.2.1.1 Examples of Typestate Problems That Are Accumulation

The problem of detecting resource leaks (Figure 6) appears 16 times across 14 papers[10] [17, 39, 72, 37, 64, 42, 43, 13, 21, 19, 3, 1, 63, 51]. This problem was already known to be accumulation [37].

The need to call a distinguished initialization method on an object after its constructor finishes but before using it appears 7 times across 4 papers [24, 17, 57, 69]. For example, when using a `Socket` object, one must call `connect()` before using it to send data (Figure 7).

A third common accumulation problem is that of object initialization: before an object is fully constructed, all of its logically-required fields must be set to reasonable values (Figure 8). This pattern appears 6 times across 6 papers [35, 36, 54, 21, 27, 30]. A variant of this problem

[9] `https://doi.org/10.5281/zenodo.5771196`

[10] We tried to stay as true as possible to the story each paper presented, which is why some automata appear multiple times in the same paper. The paper treated them differently, but we believe them to be the same example. For instance, [17] discusses memory leaks and leaked sockets, which are both resource leaks.

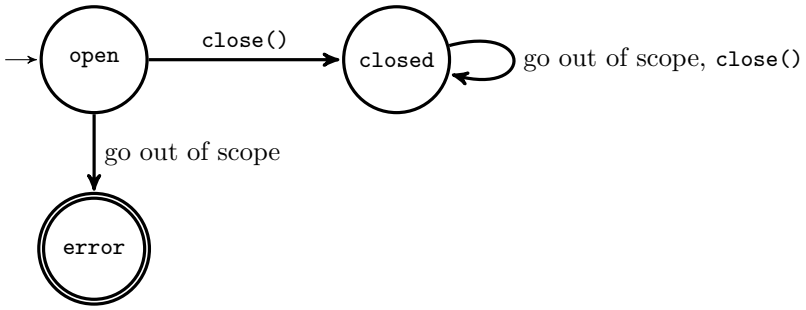

Figure 6 The typestate automaton for a resource leak, which is an accumulation typestate problem.

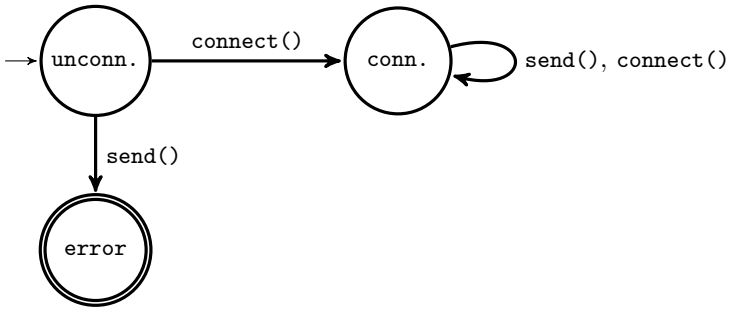

Figure 7 The typestate automaton for connecting a socket before sending data using it, which is an accumulation typestate problem.

– which arises when using the builder pattern – was known to be accumulation [35]. However, our literature survey has shown that bespoke analyses for other kinds of object initialization are also, in effect, bespoke accumulation analyses. For example, masked types [54] are a type system for ensuring that before a constructor exits, all non-null fields of the constructed class have been set to non-null values. This type system can be viewed as an accumulation analysis: the goal transition is the end of the constructor, and the enabling operations are the setting of the fields.

4.2.1.2 Examples of Typestate Problems That Are Not Accumulation

The most common non-accumulation typestate problem is "don't read or write to a stream or file after it is closed" (Figure 9), which appeared 31 times across 17 papers [24, 8, 10, 46, 25, 57, 5, 6, 53, 34, 44, 19, 71, 45, 69, 68, 11]. This problem is related to the file specification in Figure 1, but is slightly weaker – it assumes that the file is never re-opened. That this example is not accumulation demonstrates that accumulation typestate automata are a different category than automata without loops other than self-loops (a category that includes both this one and the three accumulation typestate examples in section 4.2.1.1).

"Do not update a collection while iterating over it" (Figure 10) appeared 21 times across 14 papers [9, 65, 47, 26, 51, 68, 8, 10, 33, 32, 52, 53, 7, 46]. This property is representative of an important class of properties that are never accumulation typestate systems: "disable x after y" properties that forbid the programmer from performing operation x once operation y has been performed. The key reason that these properties cannot be checked without aliasing information – and are therefore not accumulation – is that that the "disabling" operation ("start iterating" in this example) might be performed through any alias, but once it occurs, "update" must be prevented for all aliases.

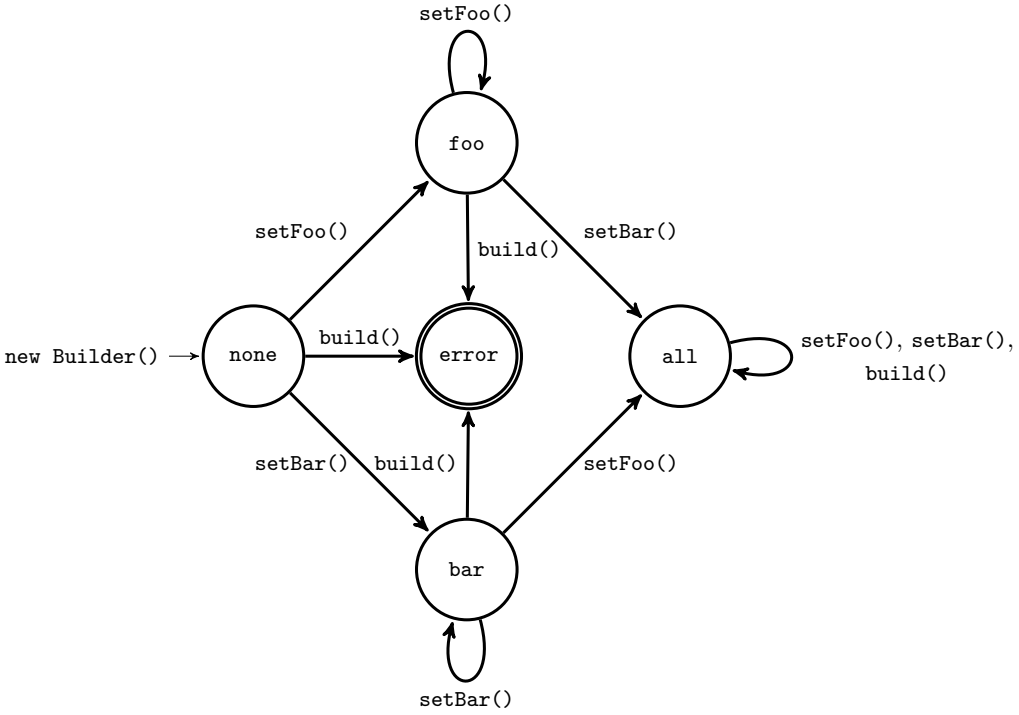

Figure 8 The typestate automaton for setting the required fields of an object before it is built, which is an accumulation typestate problem. This instance of the general pattern is specifically for a builder-pattern-style object construction pattern of a class with two required fields `foo` and `bar`.

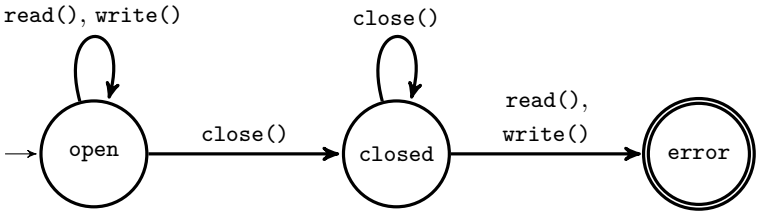

Figure 9 The typestate automaton for not reading or writing a stream after it has been closed, which is not an accumulation typestate problem.

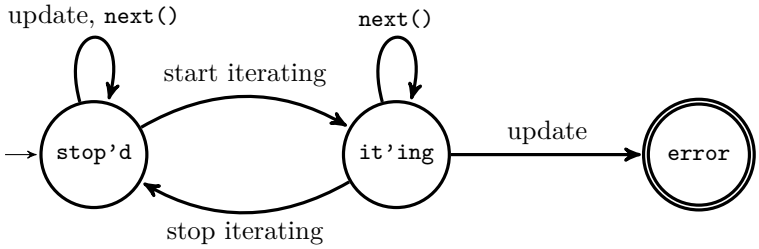

Figure 10 The typestate automaton for not updating a collection during iteration, which is not an accumulation typestate problem. Note that this automaton includes operations that are not method calls (e.g., "start iterating", which can refer to a `while` loop, a `for` loop, a map or filter operation, etc.).

The classic full file specification (Figure 1) appeared 20 times across 19 papers [28, 70, 59, 25, 29, 62, 67, 57, 69, 66, 1, 49, 16, 38, 2, 15, 72, 19, 20]. Some parts of this specification could be enforced with an accumulation analysis if a slightly different design had been chosen for the API. In particular, if files could not be re-opened once they had been closed, enforcing "only call close after open" and "only call read after open" would become accumulation properties. Since most programmers usually create a new `File` object rather than re-using an existing one, this restriction would not be particularly burdensome, but would enable easier analysis.

4.2.2 Papers With Many Typestates

This section discusses two papers that report on large collections of typestate automata.

4.2.2.1 Patterns in Property Specifications for Finite-State Verification

The first paper reports on 555 typestate-like specifications collected from a survey of 34 papers from the scientific literature, verification tool authors, and students in 1999 [18]. These 555 specifications were not de-duplicated. This paper inspired us to conduct the updated survey in Section 4.2.1. Because it precedes the start date for our survey, it is not included in the 187 papers in Section 4.2.1. We include its data here for completeness, and to discuss the differences between their results and ours (Section 4.3).

The primary goal of the paper was to categorize "finite-state properties" – that is, those expressible as finite-state machines – into patterns to help users of verification tools that take an FSM as input (such as typestate verifiers) create their own specifications by instantiating existing patterns. They categorized 511 of the 555 specifications into eight "patterns." Our analysis of these patterns is that instances of 5 of the 8 are always accumulation typestate systems (Existence, Precedence, Chain Precedence, Response, Chain Response), and some instances of a 6th (Bounded Existence, when the property is "at least" rather than "exactly" or "at most") are, as well. The 5 "always accumulation" patterns account for 306 of the 511 specifications that were categorized (60%).

4.2.2.2 An Empirical Study of Object Protocols in the Wild

The second paper [4] studies the object protocols – that is, the behavioral specifications – of all classes in four large, open-source Java projects (one of which is the Java standard library). They also categorized these specifications based on common characteristics, much like the previous study, but they created their own set of categories.

The found 648 object protocols, which were not de-duplicated. We exclude their "type qualifier" category (106 specifications), which contains classes that behave as one of a fixed set of subtypes and can never change state. The remaining 542 protocols are typestate specifications.

Instances of their most common category, Initialization, are always accumulation typestate specifications. This category contains 182 of the 542 protocols (34%). The other 6 categories (66%) are not accumulation.

4.3 Discussion

Both of the papers that reported on large sets of typestate properties included larger proportions of accumulation properties than our literature survey found otherwise. One possible explanation is that papers on novel analysis techniques tend to include "exciting" or

"challenging" problems – and, in the case of general typestate analysis, those problems usually involve aliasing (perhaps to justify the need for an alias analysis when analyzing an arbitrary typestate system, as we do in Section 1 in reference to Figure 1). Another possible explanation is that neither of the papers that reported on large sets of specifications de-duplicated their specifications, so maybe they contain many duplicate accumulation properties. When we de-duplicated the specifications in Section 4.2.1, we found that non-accumulation typestate properties tended to be duplicated more often than accumulation typestate properties. This suggests that our results may be understating the prevalence of accumulation properties. If our results understate how common accumulation properties are in practice, that is good news for practitioners interested in applying verification: we have shown that accumulation properties are easier to check than general typestate properties.

Beckman et al. [4] is the most relevant to practical programmers interested in deploying accumulation analysis. A promising avenue of future work would be a similar study to Beckman et al.'s [4] (section 4.2.2.2) on a larger corpus of software combined with automation of our decision procedure for checking whether a typestate specification is accumulation, which would permit a more reliable estimate of the percentage of typestate specifications that appear in practice that are accumulation.

Another observation is the relationship between different typestate specifications of the same type. For example, three of the examples we gave in Section 4.2.1 are applicable to `File` objects: resource leaks (Figure 6), the classic file specification (Figure 1), and reading/writing a closed file (Figure 9). Enforcing all these properties with a single typestate analysis would necessarily require alias analysis, but enforcing just the resource leak property does not – and the same might be true of other partial specifications, such as "only call read after open" – especially if files cannot be re-opened after being closed. We suspect this may be a reason why prior work did not identify a category equivalent to accumulation: many accumulation properties are sub-properties of the full typestate specification of the relevant type. That said, accumulation properties are often interesting on their own – resource leaks, for example, are harder to detect dynamically than most other types of misuses of files – and we have shown that they are easier to enforce statically.

5 Practicality of Accumulation Analysis

We implemented a general accumulation checker for Java using the Checker Framework [48] and have made it publicly available.[11] We have re-implemented the bespoke "accumulation for the builder pattern" analysis from our prior work [35] on top of it, and our "accumulation for resource leaks" analysis [37] used the general infrastructure from its inception. An accumulation analysis could be implemented modularly using any sound program analysis technique: dataflow analysis, abstract interpretation, type systems, etc. We chose a type system for convenience, and because types are naturally modular: type annotations on procedure boundaries and fields act as summaries, and local type inference infers operations that may have occurred within each procedure. Our implementation tracks enabling sets rather than enabling sequences (see Section 3.4).

We tested our implementations on the test suites of the bespoke analyses from our prior work and on the case studies that those papers describe, and found that the implementations using the common framework produced the expected results. The test suites contain both

[11] `https://checkerframework.org/manual/#accumulation-checker`

positive examples (i.e., expected errors) and negative examples (i.e., safe code). The test suites consist of 153 source files comprising 5,452 lines of non-comment, non-blank Java code. The case studies together comprise 635,006 lines of non-comment, non-blank Java code.

Our prior work also demonstrates the utility and practicality of accumulation analyses (see Section 6.1). Here are some examples from prior work:

- An accumulation analysis for verifying the absence of an initialization-related security vulnerability had 100% recall (as this paper proves, the accumulation analysis was sound!) and 82% precision – 16 true bugs vs. 3 false positives – in 9 million non-comment, non-blank lines of Java code (table 1 of [35]).
- An accumulation analysis for verifying the absence of resource leaks had 100% recall and 26% precision on 3 pieces of distributed-systems infrastructure used as a benchmark by prior work (table 4 of [37]). This compares favorably to the 13% recall and 25% precision achieved by an unsound heuristic bug-finder and the 7% recall and 50% precision achieved by a state-of-the-art typestate-based analysis that uses a (very slow) whole-program alias analysis. This precision might seem disappointing for a bug-finding tool, but we think it is acceptable for a verification tool – especially for an important and difficult problem such as resource leaks.

If the low precision of 26% for resource leaks is primarily due to lack of whole-program alias analysis – that is, if precision is much higher with comprehensive aliasing information – then there might be little point in running an accumulation analysis: it might be better to run a slow standard typestate analysis and reduce the human effort to examine false positives. This is not the case, however. We examined each false positive in [37] to determine its cause. Even with a hypothetical alias analysis that can reason precisely and flow-sensitively about the contents of collection data structures like lists or maps (which is known to be very challenging), the typestate analysis would achieve only 34% precision. A more realistic state-of-the-art (and still slow) alias analysis would give less than half of that benefit. Proving the absence of resource leaks is a difficult problem, and aliasing is not the only complication – other significant causes of false positives included bugs in the underlying analysis platform, the need to reason about nullness, and the need to reason about boolean logic.

5.1 Aliasing in Practical Accumulation Analyses

A benefit of the accumulation analysis approach is that the core accumulation analysis (Definition 4) is sound even without any alias reasoning, by Corollary 10. But it is easy to utilize aliasing information that is readily available (or cheap to compute) to improve precision. In practice, using some aliasing information is necessary to achieve acceptable precision, and untracked aliasing is usually the single biggest cause of remaining false positives even after acceptable precision has been achieved.

Our prior work [35, 37] used cheap, targeted must-alias reasoning to improve the precision – that is, the false positive rate – of the analyses. For example, section 4.3 of [35] and sections 3–5 of [37] give lightweight aliasing analyses. These lightweight alias analyses compute only the aliasing information necessary to remove false positives that occurred in practice for these analyses, which makes them much cheaper than computing precise aliasing information for all variables (of types with typestate automata) in the program, as a whole-program alias analysis would.

Our general accumulation checker includes both the suite of built-in cheap sound must-alias analyses from prior work and hooks for analysis developers to add further aliasing information.

5.2 Handling Other Features of Real Programming Languages

The core calculus in section 3.3.1.1 does not model features that are present in a practical programming language, including unanalyzed dependencies, open programs, class definitions, conditionals, inheritance, etc. Our formalism already handles some of these: for example, handling conditionals requires a may-aliasing oracle and estimated sets of typestates rather than a single typestate, both of which our formalism includes. Extending our proofs to other features is straightforward and does not require new proof techniques.

An advantage of accumulation analysis is that in practice it is possible to soundly handle code with unknown or "arbitrarily-bad" effects – including unmodeled features of the target language – by reverting to a safe default, in the same manner as an abstract interpretation might "go to top" in the presence of side effects. For example, if a call to an un-analyzed method might re-assign a field, an accumulation analysis can conservatively assume that that field's value is in the typestate automaton's start state after the call. This is sound as a consequence of Lemma 15 and the definition of accumulation (in the same manner as Lemma 16): the start state is necessarily a sound default assumption, because all goal transitions must be forbidden in it.

By contrast, in a non-accumulation typestate system it is not sound to fall back to the automaton's start state. For example, consider the `File` example in Figure 1: the start state is `closed`, where `open()` is a legal call. But treating all field reads as returning `closed` files would not be sound, because if the underlying `File` value was actually in the `open` state, a sound analysis should issue an error for a subsequent call to `open()`.

An advantage of our choice of a pluggable type system to implement our accumulation analyses is that the "start state" of a field can be changed by changing its declared type to specify a different typestate. This restricts that field to only contain values whose typestates are in the states reachable from the declared typestate – that is, the sub-automaton composed of states reachable from the declared type. For the accumulation analyses we implemented, we found that this ability to refine a field's declared type to be sufficient to enable precise analysis of field reads.

6 Related Work

6.1 Previous Work on Accumulation

Our prior work [35, 37] uses accumulation analyses to solve specific typestate-like problems (object initialization via the builder pattern and resource leak prevention). One of these [35] gives an informal relationship between accumulation and typestate: we claimed that a typestate automaton can be checked with an accumulation analysis if "(1) the order in which operations are performed does not affect what is subsequently legal, and (2) the accumulation does not add restrictions; that is, as more operations are performed, more operations become legal." We did not substantiate this definition with a proof, and it is not quite equivalent to the definition of an accumulation typestate system we use in this paper, which does permit some kinds of ordering properties (see Section 3.4). This paper makes more precise claims and provides a proof that the analyses are sound (Corollary 10).

6.2 Heap Monotonic Typestates

Heap-monotonic typestates [22] are a class of typestate that, like accumulation typestate systems, do not require aliasing information for soundness. A heap monotonic typestate system is one in which the statically observable invariants of the relevant type become monotonically stronger as an object transitions through its typestates. Every heap-monotonic typestate system is an accumulation typestate system.

The present work goes further than the work on heap-monotonic typestates in three important ways. First, we have shown exactly which typestate systems (the accumulation typestate systems) can be checked without aliasing; heap-monotonic typestate systems were proven to be sound without aliasing information, but not proven to encompass all typestate systems that can be soundly checked without aliasing. Second, we have surveyed the literature to locate examples of typestate systems that can be checked soundly without aliasing; the paper on heap-monotonic typestates gives a few examples, but no procedure for discovering more. Third, we have implemented practical accumulation analyses: the prior work on heap-monotonic typestates was, to the best of our knowledge, entirely theoretical.

6.3 Other Categories of Typestate Systems

Others have identified interesting sub-categories of typestate systems that may be amenable to different kinds of analysis. While as far as we are aware we are the first to identify the accumulation typestate systems, the omission-closed typestate systems [23] are a close relative. An omission-closed typestate system is one in which every subsequence of every valid (i.e., not ending in the `error` state) path is also a valid path. In other words, omission-closed properties are those whose *valid* paths are closed under subsequence. By contrast, accumulation typestate systems are those whose *error-inducing* paths are closed under subsequence, if the last error-inducing transition is held constant. Unlike accumulation typestate systems, not all omission-closed typestate systems can be checked soundly without aliasing: for example, the typestate system for a `File` object whose FSM is defined by the regular expression "read*;close" is omission-closed, but cannot be checked soundly without aliasing information, because it is an error to call "close" more than once – or, put another way, "close" disables itself. Omission-closed typestate properties are of interest because they can be verified in polynomial time for *shallow* programs – programs where all pointers are "single-level": that is, where no pointer refers to a value that itself contains a pointer.

6.4 Typestate Surveys

Section 4.2.2 describes two previous papers that report on large quantities of typestate specifications [4, 18]. We have extended their work by surveying 101 papers that neither of those works considered and locating all typestates within them, and by identifying which typestate systems are accumulation typestate systems.

6.5 Practical Typestate Analyses

There have been many attempts to improve the scalability of typestate analyses. We mention only some of the most recent here. Rapid [21] is a modern typestate analysis built at AWS. Rapid's scalability is a design choice: it is intentionally unsound and therefore scales by not tracking all aliasing. Another recent example is Grapple [72], which uses a novel graph-reachability algorithm and a modern alias analysis together. Some of Grapple's optimizations make it unsound despite access to aliasing information. Because Grapple does track aliasing, it scales much poorly than accumulation-based systems: for example, Grapple is more than an order of magnitude slower than an accumulation-based approach to resource-leak detection [37].

6.6 Typestate With Aliasing Restrictions

Another method to avoid the need to do an expensive whole-program alias analysis is to limit the programmer's use of aliasing. Examples include linear or affine type systems [16, 61], role analysis [40], ownership types [14, 55], and access permissions [7]. Accumulation analyses, unlike all of these approaches, do not impose any restrictions on the programming model.

6.7 Other Work on Typestate

Typestate is well-studied in the scientific literature, and there is not space to give a full survey here. However, our artifact[12] mentions all the papers that we examined as part of our literature survey (Section 4).

7 Conclusion

Soundly checking an accumulation typestate system is significantly cheaper than soundly checking an arbitrary typestate system because it is not necessary to compute exhaustive aliasing information. Since the expense of computing exhaustive aliasing information has been a key barrier for the adoption of sound typestate analyses in practice, we believe that accumulation analysis is a promising approach for the estimated 41% (Table 1) of typestate specifications that are actually accumulation typestate specifications. Typestate analysis designers or users can use our work to check whether their specification is an accumulation typestate specification, and if it is, they can use an accumulation analysis – gaining an order of magnitude or more in analysis speed at only a small cost in precision.

References

1 Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *International Static Analysis Symposium*, pages 230–246. Springer, 2002.

2 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022, 2009.

3 Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 143–162, 2008.

4 Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols in the wild. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer, 2011.

5 Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with typestates. *ACM SIGSOFT Software Engineering Notes*, 30(5):217–226, 2005.

6 Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *ACM SIGPLAN Notices*, 42(10):301–320, 2007.

7 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking with access permissions. In *European Conference on Object-Oriented Programming*, pages 195–219. Springer, 2009.

8 Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 5–14. IEEE, 2010.

[12] `https://doi.org/10.5281/zenodo.5771196`

9 Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *European Conference on Object-Oriented Programming*, pages 525–549. Springer, 2007.

10 Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47, 2008.

11 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.

12 Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58, 2009.

13 Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 480–491, 2007.

14 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, Berlin, Heidelberg, 2013.

15 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, 2002.

16 Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with Java(X). In *European Conference on Object-Oriented Programming*, pages 550–574. Springer, 2007.

17 Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, 2001.

18 Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *International Conference on Software Engineering*, pages 411–420, 1999.

19 Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path property monitoring through sampling. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 228–237. IEEE, 2008.

20 Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 124–133, 2007.

21 Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. RAPID: Checking API usage for the cloud in the cloud. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1416–1426, 2021.

22 Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic typestates. In *IWACO 2003: International Workshop on Aliasing, Confinement and Ownership in object-oriented programming*, pages 58–72, Darmstadt, Germany, July 2003.

23 John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. Typestate verification: Abstraction techniques and complexity results. In *International Static Analysis Symposium*, pages 439–462. Springer, 2003.

24 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(2):1–34, 2008.

25 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12, 2002.

26 Asya Frumkin, Yotam M. Y. Feldman, Ondřej Lhoták, Oded Padon, Mooly Sagiv, and Sharon Shoham. Property directed reachability for proving absence of concurrent modification errors. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 209–227. Springer, 2017.

27 Mark Gabel and Zhendong Su. Testing mined specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

28 Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: toward manifesting hidden concurrency typestate bugs. *ACM Sigplan Notices*, 46(3):239–250, 2011.

29 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(4):1–44, 2014.

30 Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *European Conference on Object-Oriented Programming*, pages 520–545. Springer, 2009.

31 Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society*, 3(1):326–336, 1952.

32 Jeff Huang, Qingzhou Luo, and Grigore Rosu. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 847–857. IEEE, 2015.

33 Dongyun Jin, Patrick O'Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage collection for monitoring parametric properties. *ACM SIGPLAN Notices*, 46(6):415–424, 2011.

34 Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded Java programs. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 288–296. IEEE, 2008.

35 Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. Verifying object construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*, pages 1447–1458, Seoul, Korea, May 2020.

36 Martin Kellogg, Martin Schäf, Serdar Tasiran, and Michael D. Ernst. Continuous compliance. In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software Engineering*, pages 511–523, Melbourne, Australia, September 2020.

37 Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Athens, Greece, August 2021.

38 Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 416–428, 2009.

39 Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 109–118, 2008.

40 Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32, 2002.

41 William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL '91: Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, Orlando, FL, January 1991.

42 Wei Le and Mary Lou Soffa. Marple: Detecting faults in path segments using automatically generated analyses. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):1–38, 2013.

43 Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Memfix: static analysis-based repair of memory deallocation errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 95–106, 2018.

44 Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

45 Filipe Militao, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In *European Conference on Object-Oriented Programming*, pages 334–359. Springer, 2014.

46 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects. *ACM Sigplan Notices*, 43(10):347–366, 2008.

47 Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates in the presence of inter-object references. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 77–96, 2005.

48 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008.

49 Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. Symbolic automata for static specification mining. In *International Static Analysis Symposium*, pages 63–83. Springer, 2013.

50 Goran Piskachev, Tobias Petrasch, Johannes Späth, and Eric Bodden. AuthCheck: Program-state analysis for access-control vulnerabilities. In *International Symposium on Formal Methods*, pages 557–572. Springer, 2019.

51 Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012.

52 Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 270–285, 2010.

53 Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Optimizing monitoring of finite state properties through monitor compaction. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 280–290, 2013.

54 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. *ACM SIGPLAN Notices*, 44(1):53–65, 2009.

55 Rust team. Rust programming language. `https://www.rust-lang.org/`, 2021. Accessed 30 November 2021.

56 Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. *IEEE Transactions on Software Engineering*, 34(5):651–666, 2008.

57 Johannes Späth, Karim Ali, and Eric Bodden. IDEal: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.

58 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, SE-12(1):157–171, January 1986.

59 Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class state change in Plaid. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages, and Applications*, pages 713–732, Portland, OR, USA, October 2011.

60 Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–27, 2021.

61 Jesse A. Tov and Riccardo Pucella. Practical affine types. *ACM SIGPLAN Notices*, 46(1):447–458, 2011.

62 Cláudio Vasconcelos and António Ravara. From object-oriented code with assertions to behavioural types. In *Proceedings of the Symposium on Applied Computing*, pages 1492–1497, 2017.

63 Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns in android applications. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 185–195, 2016.

64 Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.

65 Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak Chhetri. Arc++: effective typestate and lifetime dependency analysis. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 116–126, 2014.

66 Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):1–50, 2014.

67 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–234, 2008.

68 Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. Symbolic verification of regular properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 871–881. IEEE, 2018.

69 Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 1–12, 2014.

70 Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. *ACM SIGPLAN Notices*, 48(6):365–376, 2013.

71 Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. Regular property guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 643–653. IEEE, 2015.

72 Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*, pages 1–17, 2019.

A Proof of Lemma 15

This appendix contains the full proof of Lemma 15, which appears in section 3.3.1.6 and is used by Lemma 16, the forwards direction of the proof of Theorem 9. We begin by restating Lemma 15:

▶ **Lemma 15.** *Let $R = \phi_s(x_i)$ be the set of estimated typestates produced by a typestate analysis with no aliasing information for a variable x_i before a statement s. Let S be the trace of an arbitrary execution leading up to some occurrence of s, and let $t = \tau(\rho(x_i))$ be the typestate of the actual value to which x_i refers before that occurrence of s. Applying S to the automaton leads to typestate t. There exists a typestate $r \in R$ such that applying some subsequence of S leads to r. That is, there is some estimated typestate $r \in R$ that is reachable by a subsequence of the transitions that lead to t.*

The proof is by co-induction on the dynamic semantics of the language in Figure 2 and the definition of a typestate analysis with no aliasing information in Definition 14, with one change to its rule for load operations (rule TS-LOAD-FIX in Figure 5). In particular, the load rule our typestate analysis with no aliasing uses in this proof is the following:

- For a load statement s, where s is x_i `:=` $x_j.f_k$, let s_0 be the start state of the automaton A which is being checked. The analysis updates its estimate for x_i so that it is mapped to s_0: $\phi'_s(x_i) := s_o$. For all other names m in ϕ_s where $m \neq x_i$, the analysis copies the entry from the previous abstract store: $\phi'_S(m) := \phi_s(m)$.

(See the discussion of why this modified rule is necessary in section 3.3.1.6, after the original statement of Lemma 15.)

Proof.

Base case: when a program begins executing, the dynamic semantics say that all names refer to values in the start state. A typestate analysis with no aliasing information estimates that at a program's entry point, all names are in the start state, as well. Trivially, the start state is reachable by the same sequence of operations as itself.

Case assignment: For an assignment s, where s is x_i `:=` x_j, the invariant is preserved by the inductive hypothesis. Consider that by the inductive hypothesis, the invariant is preserved for x_j. Then consider the rule used by the typestate analysis with no aliasing information for an assignment: every mention of x_i in the abstract store is replaced by x_j. Further, the dynamic semantics for an assignment require that the previous value of x_i is no longer accessible via x_i: x_i after the assignment refers only to x_j. Since x_i and x_j after the assignment are treated entirely the same, but the abstract store is otherwise unchanged by the analysis, what was true of x_j before the statement is true for x_i after.

Case load: The special load rule TS-LOAD-FIX trivially guarantees that the invariant is preserved: the start state is reachable by a subsequence of the operations that reach any other state (in particular, by the empty subsequence).

Case store: This rule trivially preserves the invariant, because the invariant must be maintained only for the estimates for variables – not for fields – and rule TS-STORE only updates estimates for fields.

Case method call: For a method call $s = x_i.m_j()$, only steps 1 and 2 of rule TS-CALL are applied, because a typestate analysis with no aliasing information never performs strong or weak updates on possible aliases. The invariant is preserved via the inductive hypothesis: for x_i itself, let r_1 be the element of R that is reachable by a subsequence of the actual sequence S in the inductive hypothesis. The analysis updates its estimate to include $r_1 + m_j$ (that is, the sequence r_1 followed by the transition m_j). After s is executed, the actual sequence is $S + m_j$, and since we know that r_1 is reachable by a subsequence of S, $r_1 + m_j$ must be reachable by a subsequence of $S + m_j$ – the same subsequence used to reach r_1, with m_j added on. For any aliases of x_i, the inductive hypothesis also guarantees that the invariant holds: the estimate contains some r that is a subsequence of S, and any subsequence of S is also a subsequence of $S + m_j$.

Case sequence: For a sequence, the invariant is trivially preserved by induction. ◀