

# Accumulation Analysis

Martin Kellogg ✉

University of Washington, USA

Narges Shadab ✉

University of California, Riverside, USA

Manu Sridharan ✉

University of California, Riverside, USA

Michael D. Ernst ✉

University of Washington, 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

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

## 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 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



© Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst;  
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).

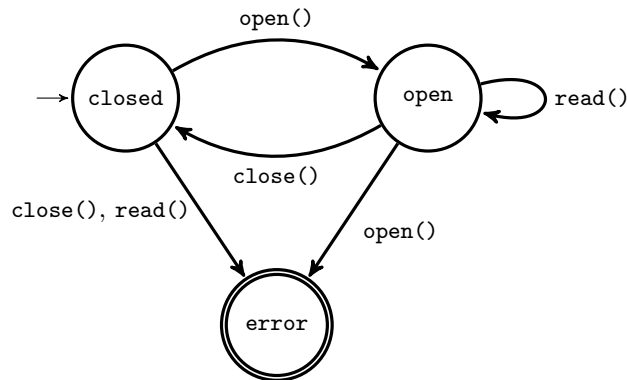
Editors: Karim Ali and Jan Vitek; Article No. 10; pp. 10:1–10:31

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 10:2 Accumulation Analysis



■ **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.

45 `File` object, whose typestate is specified in Figure 1, and the following program in a Java-like  
46 imperative language:

```
47  
48 1   File f = new File(...);  
49 2   f.open();  
50 3   File g = f; // f and g are aliases after this line is executed  
51 4   g.close();  
52 5   f.read(); // an error occurs when this line is executed  
53
```

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

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

69 In recent work [35, 37], we proposed bespoke *accumulation analyses* that soundly and  
70 modularly solve specific problems traditionally addressed with typestate. An accumulation  
71 analysis collects operations—corresponding to typestate transitions—that have definitely  
72 occurred on a given program expression. For example, an accumulation analysis could check  
73 the property “before calling `read()` on a `File`, call `open()`.” The accumulation analysis would  
74 record on which expressions `open()` had definitely been called, and forbid calls to `read()`  
75 that did not occur via such expressions. Note that this is a weaker property than the full  
76 specification in Figure 1—it does not forbid “read after close” defects.

77 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 tpestate 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 tpestate 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 tpestate 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 tpestate-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 tpestate system*, and a proof that the properties checkable via accumulation analysis are all accumulation tpestate properties (Section 3.2);
- a proof that a tpestate system can be checked soundly by a tpestate analysis that does no aliasing reasoning if and only if it is an accumulation tpestate system (Section 3.3);
- a literature survey of work on tpestate analysis, from which we collected 1,355 tpestate specifications and determined that 41% of them are accumulation tpestate 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 Tpestate?

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. Tpestate [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 tpestate automaton for that type. Formally:

► **Definition 1.** A *tpestate automaton*  $A = (\Sigma, S, s_0, \delta, e)$  for type  $\tau$  is a finite-state machine. The language  $\Sigma$  is the set of operations, such as method calls, that can be performed on  $\tau$ . The states  $S$  are called tpestates;  $s_0 \in S$  is the initial state. The edges defined by the transition table  $\delta$  are called transitions and correspond to the effect of operations. There is a distinguished error state  $e \in S$ . Each tpestate has  $k = |\Sigma|$  outgoing transitions; none, some,

## 10:4 Accumulation Analysis

124 or all of these transitions may be to the error state  $e$  or may be self-loops. The error state  $e$   
125 has only self-loops—that is, the error state is a trap state.

126 At every step during the execution of a program, each value/object of type  $\tau$  is in one of  
127 the tpestates of the tpestate system.

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

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

138 ► **Definition 3.** A *tpestate system* is the pair of a tpestate automaton and the corre-  
139 sponding type  $\tau$  whose safe usage it encodes.

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

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

### 3 Definitions and Proofs

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

#### 3.1 Accumulation Analysis

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

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

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

---

<sup>1</sup> Our definition is consistent with but not identical to the definitions used in prior work. See Section 6.1.

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

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

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

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

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

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

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

## 188 3.2 Relationship Between Typestate and Accumulation

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

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

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

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

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

■ **Algorithm 1** A decision procedure for checking whether or not a given tpestate automaton  $T$  is an accumulation tpestate 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:    $U \leftarrow$  FINDERRORINDUCINGTRANSITIONS( $T$ )
4:   //  $E$  and  $E_{subseq}$  are finite-state automata.  $\forall X, \text{UNION}(\emptyset, X) = X$ .
5:    $E \leftarrow \emptyset$ 
6:    $E_{subseq} \leftarrow \emptyset$ 
7:   for  $u_i \in U$  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:     $E_i \leftarrow$  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:     $E_{subseq(i)} \leftarrow$  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:     $E \leftarrow$  UNION( $E, \text{CONCAT}(E_i, u_i)$ )
19:     $E_{subseq} \leftarrow$  UNION( $E_{subseq}, \text{CONCAT}(E_{subseq(i)}, u_i)$ )
20:    // ACCEPTSAMELANGUAGE is true iff the two automata accept the same language.
21:  return ACCEPTSAMELANGUAGE( $E, E_{subseq}$ )

```

---

206 This definition leads to a decision procedure (Algorithm 1) for determining whether a  
207 given tpestate system  $T$  is an accumulation tpestate system. Consider all error-inducing  
208 operations  $U = \{u_1, \dots, u_n\}$ . The elements of  $U$  are the final transitions for every error-  
209 inducing sequence in the automaton of  $T$ . For any  $u_i \in U$ , let  $E_i$  be the language<sup>2</sup> of the  
210 error-inducing sequences of operations in  $T$  that end in  $u_i$ , with the last transition removed  
211 (i.e., the  $u_i$  transition that leads to the error tpestate). Let  $E_{subseq(i)}$  be the language of  
212 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  
213 the union of all error-inducing paths in  $T$ , and  $E_{subseq}$  is the union of all subsequences of  
214 error-inducing paths in  $T$  that end in the same transition as the corresponding error-inducing  
215 path from which they were derived. By Definition 7, if and only if  $E$  and  $E_{subseq}$  recognize  
216 the same language,  $T$  is an accumulation tpestate system.

217 It is easy to check whether  $E$  and  $E_{subseq}$  recognize the same language, because both are  
218 regular.  $E$  is regular, because it can be recognized by  $T$ 's automaton, if the error tpestate is  
219 converted to an accepting state. Since there are finitely-many operations, any  $E_i$  and  $E_{subseq(i)}$   
220 have a finite alphabet. Higman's theorem [31] says that the language of the subsequences  
221 of any language over a finite-alphabet is regular. Therefore, any  $E_{subseq(i)}$  is also regular.  
222  $E_{subseq}$  is regular because regular languages are closed under both union and concatenation.  
223 So, the procedure for checking whether a tpestate automaton is an accumulation tpestate  
224 automaton is as easy as checking whether the two finite state machines for  $E$  and  $E_{subseq}$

---

<sup>2</sup> Throughout, we will abuse notation and refer to both languages and their corresponding language-recognizers by the same name.

225 recognize the same language.

226 ► **Theorem 8.** *Every accumulation analysis has a corresponding accumulation typestate*  
227 *system.*

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

- 231 1. Create an error state **error** with a self-loop transition for each operation on  $\tau$ .
- 232 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  
233  $E_1, \dots, E_n$ . For each element  $S$  of  $\mathcal{P}_E$ , create a corresponding state and label it with  $S$ .  
234 Note that  $S$  refers to both the member of  $\mathcal{P}_E$  and the corresponding state.
- 235 3. Make the state that is labeled by the empty set be the start state of the automaton.
- 236 4. For each state  $S \in \mathcal{P}_E$  and for each transition  $t_e \in E$ , add a transition from state  $S$  to  
237 state  $S \cup \{t_e\}$  labeled  $t_e$ . (This transition might be a self-loop.)
- 238 5. Let  $G = \{g_1, \dots, g_n\}$  be the set of goal transitions. For each element  $g_i$  of  $G$  and for each  
239 state  $S \in \mathcal{P}_E$ :  
240     If there exists a goal  $\langle g_i, E_i \rangle$  such that  $E_i \subseteq S$ ,  
241         then add a self-loop transition to  $S$  labeled  $g_i$  if it does not already have a  
242         transition labeled  $g_i$ . (It might have such a transition if  $g_i$  is both an enabling  
243         transition and a goal transition.)  
244     Else if such a goal does not exist,  
245         add a transition from  $S$  to the **error** state labeled  $g_i$ , removing a transition labeled  
246          $g_i$  if one already exists.
- 247 6. For each operation  $t$  on  $\tau$  such that  $t \notin G$  and  $t \notin E$ —that is, for each operation that is  
248 neither a goal operation nor an enabling operation—add a self-loop transition labeled  $t$   
249 to each non-error state. (Recall that the error state already has self-loop transitions for  
250 each operation, added in step 1.)

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

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

### 257 3.3 Soundness Without Aliasing

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

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

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

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

## 10:8 Accumulation Analysis

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

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

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

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

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

### 282 3.3.1 Preliminaries

283 This section introduces the machinery used to prove Theorem 9.

#### 284 3.3.1.1 Language

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

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

- 289 ■ an assignment:  $x_i := x_j$ .
- 290 ■ a field load:  $x_i := x_j.f_k$ .
- 291 ■ a field store:  $x_i.f_j := x_k$ .
- 292 ■ a method call:  $x_i.m_j()$ .
- 293 ■ a statement sequence:  $s_i ; s_j$ .

294 Source code variables range from  $\mathbf{x}_{-1}$  to  $\mathbf{x}_{-n}$ , where  $n$  is some positive integer. Statements  
295 may only refer to variables in that range. There is a single type  $T$ . Each variable refers to a  
296 *value*—that is, a particular object instance—of type  $T$ . We use  $x_i, x_j, \dots$  as metavariables for  
297 arbitrary variables in the range  $\mathbf{x}_{-1}, \dots, \mathbf{x}_{-n}$ .  $T$  has methods  $\mathbf{m}_{-1}$  to  $\mathbf{m}_{-k}$  and a corresponding  
298 tpestate automaton  $A$  whose  $k$  operations are exactly the methods  $\mathbf{m}_{-1}$  to  $\mathbf{m}_{-k}$ . A method call  
299 statement can only refer to methods in  $T$ . We use  $m_i, m_j, \dots$  as metavariables for arbitrary  
300 methods in  $T$ . Each object of type  $T$  has fields  $\mathbf{f}_{-1}$  to  $\mathbf{f}_{-m}$ , where  $m$  is some positive integer.  
301 Load and store statements may only refer to fields in this range. Each field refers to some  
302 value of type  $T$ . We use  $f_i, f_j, \dots$  as metavariables for arbitrary fields in  $T$ .

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



$$\begin{array}{c}
\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j \Downarrow \langle \rho[x_i \mapsto \rho(x_j)], \sigma, \tau \rangle} \text{ASSIGN} \\
\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i := x_j.f_k \Downarrow \langle \rho[x_i \mapsto \sigma(\langle \rho(x_j), f_k \rangle)], \sigma, \tau \rangle} \text{LOAD} \\
\frac{}{\langle \rho, \sigma, \tau \rangle \vdash x_i.f_j := x_k \Downarrow \langle \rho, \sigma[\langle \rho(x_i), f_j \rangle \mapsto \rho(x_k)], \tau \rangle} \text{STORE} \\
\frac{\langle \rho, \sigma, \tau \rangle \vdash t' = \text{succ}(\tau(\rho(x_i)), m_j, A) \quad t' \neq \text{error}}{\langle \rho, \sigma, \tau \rangle \vdash x_i.m_j() \Downarrow \langle \rho, \sigma, \tau[\rho(x_i) \mapsto t'] \rangle} \text{CALL} \\
\frac{\langle \rho, \sigma, \tau \rangle \vdash s_i \Downarrow \langle \rho', \sigma', \tau' \rangle \quad \langle \rho', \sigma', \tau' \rangle \vdash s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle}{\langle \rho, \sigma, \tau \rangle \vdash s_i ; s_j \Downarrow \langle \rho'', \sigma'', \tau'' \rangle} \text{SEQ}
\end{array}$$

■ **Figure 2** The big-step dynamic semantics of the language expressed as inference rules. The notation  $\mu[x \mapsto y]$  means that the map  $\mu$  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 ( $\rho$ ) mapping each variable to a value of type  $T$ , a store ( $\sigma$ ) mapping each value–field pair to a value, and a typestate store ( $\tau$ ) 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  $A$ .<sup>3</sup> 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  $t' = \text{succ}(\tau(\rho(x_i)), m_j, A)$ . 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).

<sup>3</sup> Initializing all variables before a program starts simplifies the language by removing the need for a **new** expression.

333 **3.3.1.3 Sound Typestate Analysis**

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

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

339 **3.3.1.4 Representation of Aliasing**

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

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

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

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

359 **3.3.1.5 Definition of Typestate Analysis**

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

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

---

<sup>4</sup> 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.

<sup>5</sup> An analysis may use widening, abstraction, or iterative expansion of maps to handle the fact that the set of names is infinite.

$$\begin{array}{c}
\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j/x_i] \wedge T'_{n'} = \phi_s(n') \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]}{\phi_s \vdash x_i := x_j \Downarrow \phi'_s} \text{TS-ASSIGN} \\
\\
\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.*), n' = n[x_j.f_k/x_i] \wedge T'_{n'} = \phi_s(n') \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.*), n \mapsto T'_{n'}]}{\phi_s \vdash x_i := x_j.f_k \Downarrow \phi'_s} \text{TS-LOAD} \\
\\
\frac{\phi_s \vdash \forall n \in \hat{\phi}_s(x_i.f_j.*), n' = n[x_k/x_i.f_j] \wedge T'_{n'} = \phi_s(n') \wedge A_n^{must} = MustOracle(n, s) \wedge A_n^{may} = MayOracle(n, s) \quad \phi'_s = \phi_s[\forall n \in \hat{\phi}_s(x_i.f_j.*), n \mapsto T'_{n'}][\forall a_n \in A_n^{must}, a_n \mapsto T'_{n'}][\forall b_n \in A_n^{may} - A_n^{must}, b_n \mapsto T'_{n'} \cup \phi_s(b_n)]}{\phi_s \vdash x_i.f_j := x_k \Downarrow \phi'_s} \text{TS-STORE} \\
\\
\frac{\phi_s \vdash T = \phi_s(x_i) \quad T' = \bigcup_{t \in T} succ(t, m_j, A) \quad A_n^{must} = MustOracle(x_i, s) \quad A_n^{may} = MayOracle(x_i, s) \quad \phi'_s = \phi_s[x_i \mapsto T'][\forall a \in A_n^{must}, a \mapsto T'][\forall b \in A_n^{may} - A_n^{must}, b \mapsto T' \cup \phi_s(b)]}{\phi_s \vdash x_i.m_j() \Downarrow \phi'_s} \text{TS-CALL} \\
\\
\frac{\phi_s \vdash s_i \Downarrow \phi'_{s_i} \quad \phi'_{s_i} = \phi_{s_j} \quad \phi_{s_j} \vdash s_j \Downarrow \phi'_s}{\phi_s \vdash s_i; s_j \Downarrow \phi'_s} \text{TS-SEQ}
\end{array}$$

■ **Figure 3** Inference rules for a traditional, sound tpestate 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  $\phi_s$  that begin with  $x_i$ .

- 370 ■ 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  
371 its  $x_i$  replaced by  $x_j$ —and let  $T'_{n'} = \phi_s(n')$ , the abstract value of  $n'$  in the pre-state. The  
372 analysis updates the abstract store after  $s$  so that  $n$  is mapped to  $T'_{n'}$ :  $\phi'_s(n) := T'_{n'}$  (rule  
373 TS-ASSIGN). For all other names  $m$  in  $\phi_s$  where  $m \notin \hat{\phi}_s(x_i.*)$ , the analysis copies the  
374 entry from the previous abstract store:  $\phi'_s(m) := \phi_s(m)$ .
- 375 ■ 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  
376  $T'_{n'} = \phi_s(n')$ . The analysis updates the abstract store after  $s$  so that  $n$  is mapped to  $T'_{n'}$ :  
377  $\phi'_s(n) := T'_{n'}$  (rule TS-LOAD). For all other names  $m$  in  $\phi_s$  where  $m \notin \hat{\phi}_s(x_i.*)$ , the  
378 analysis copies the entry from the previous abstract store:  $\phi'_s(m) := \phi_s(m)$ .
- 379 ■ 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  
380  $T'_{n'} = \phi_s(n')$ . Then, for each  $n$  and its  $n'$  and  $T'_{n'}$ , the analysis performs the following  
381 steps (rule TS-STORE):
- 382 1. The analysis updates the abstract store after  $s$  so that  $n$  is mapped to  $T'_{n'}$ :  $\phi'_s(n) := T'_{n'}$ .
  - 383 2. The analysis queries  $MustOracle(n, s)$  (call the result  $A_n^{must}$ ). For each  $a_n \in A_n^{must}$ ,  
384 the analysis performs a *strong update* to the abstract store:  $\phi'_s(a_n) := T'_{n'}$ .
  - 385 3. The analysis queries  $MayOracle(n, s)$  (call the result  $A_n^{may}$ ). For each element  $b_n$  in  
386  $A_n^{may} - A_n^{must}$ —that is, variables that may be aliases but are not guaranteed to be  
387 aliases—the analysis performs a *weak update* to the abstract store so that it maps  $b_n$   
388 to  $T'_{n'} \cup \phi_s(b_n)$ :  $\forall b_n \in A_n^{may} - A_n^{must}, \phi'_s(b_n) := T'_{n'} \cup \phi_s(b_n)$ .

## 10:12 Accumulation Analysis

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

391 ■ For a call statement  $x_i.m_j()$ , let  $T' = \bigcup_{t \in \phi_s(x_i)}$ . The analysis performs the following  
 392 steps (rule TS-CALL):

393 1. If any  $t' \in T'$  is **error**, the analysis reports an error for the statement. Note that while  
 394 the dynamic semantics (Figure 2) do not permit any value to be in the **error** typestate  
 395 (the program crashes instead), this analysis approximates the semantics statically.

396 2. The analysis updates the abstract store so that  $\phi'_s(x_i) := T'$ .

397 3. The analysis queries  $MustOracle(x_i, s)$  (call the result  $A^{must}$ ). For each  $a \in A^{must}$ ,  
 398 the analysis performs a strong update to the abstract store:  $\phi'_s(a) := T'$ .

399 4. The analysis queries  $MayOracle(x_i, s)$  (call the result  $A^{may}$ ). For each  $b \in A^{may} - A^{must}$ ,  
 400 the analysis performs a weak update to the abstract store:  $\phi'_s(b) := T' \cup \phi_s(b)$ .

401 ■ For a sequence  $s = s_i ; s_j$ , the analysis first analyzes  $s_i$ , and then analyzes  $s_j$  with the  
 402 resulting abstract store (rule TS-SEQ). (Note that the analysis does not terminate in  
 403 the case of an error, but keeps reporting errors on subsequent statements.)

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

406 ► **Theorem 13.** *A traditional typestate analysis is sound if its  $MustOracle$  and  $MayOracle$*   
 407 *functions return sound results.*

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

### 411 3.3.1.6 Typestate Analysis with No Aliasing Information

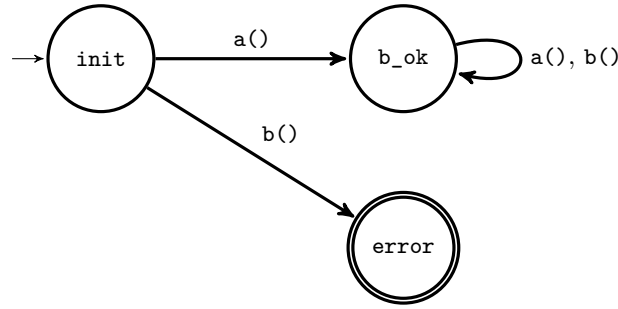
412 ► **Definition 14.** *A typestate analysis with no alias information is a typestate analysis*  
 413 *whose  $MustOracle$  and  $MayOracle$  functions return empty lists for all arguments.*

414 Intuitively, a typestate analysis “with no alias information” assumes that no aliasing  
 415 occurs in the program—even when making such an assumption is unsound.

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

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

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



■ **Figure 4** An accumulation typestate automaton for the property “call `a()` before calling `b()`”.

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

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

Program	Estimated state ( $\phi_s$ ) <sup>6</sup>	Actual state ( $\tau$ ) <sup>7</sup>
<code>x2 = x1</code>	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}\}$	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}\}$
<code>x3.a()</code>	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}, x3 \mapsto \text{b\_ok}\}$	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}, x3 \mapsto \text{b\_ok}\}$
<code>x1.f = x3</code>	$\{x1.f \mapsto \text{b\_ok}, x2.f \mapsto \text{init}, x3 \mapsto \text{b\_ok}\}$	$\{x1.f \mapsto \text{b\_ok}, x2.f \mapsto \text{init}, x3 \mapsto \text{b\_ok}\}$
<code>x2.f = x4</code>	$\{x1.f \mapsto \text{b\_ok}, x2.f \mapsto \text{init}\}$	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}\}$
<code>x5 = x1.f</code>	$\{x1.f \mapsto \text{b\_ok}, x2.f \mapsto \text{init}, x5 \mapsto \text{b\_ok}\}$	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}, x5 \mapsto \text{init}\}$
<code>x5.b()</code>	$\{x1.f \mapsto \text{b\_ok}, x2.f \mapsto \text{init}, x5 \mapsto \text{b\_ok}\}$	$\{x1.f \mapsto \text{init}, x2.f \mapsto \text{init}, x5 \mapsto \text{init}\}$

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

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

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

455 This rule trivially preserves Lemma 15 for field loads, and corresponds with how accu-  
 456 mulation analyses handle field loads in practice (see Section 5.2). Our proof assumes this

<sup>6</sup> Entries in  $\phi_s$  are single-element sets. For simplicity of presentation, set notation has been elided.

<sup>7</sup> Keys in  $\tau$  are values. For simplicity of presentation, the necessary lookups in  $\rho$  and  $\sigma$  have been elided.

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

■ **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.

457 simpler load rule for the typestate analysis with no aliasing information. However, note  
 458 that this rule would make a traditional typestate analysis unsound (i.e., this rule makes  
 459 Theorem 13 untrue): in an arbitrary typestate analysis, the start state is not necessarily a  
 460 safe default assumption. A useful property of accumulation typestate automata, however, is  
 461 that every operation which might ever lead to an error on any path must necessarily lead to  
 462 an error from the start state—otherwise, the definition of accumulation typestate automaton  
 463 could not be met when considering the empty subsequence.

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

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

### 473 3.3.2 Proof of Theorem 9

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

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

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

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

489 Since  $R'$  does not contain the error typestate after observing  $m_j$ , then  $m_j$  must have  
 490 been a legal transition on each typestate in the analysis’ pre-state estimate  $R = \phi_s(x_i)$ .

---

<sup>8</sup>  $s$  must be a method call statement, so  $v_i$  is the same before and after  $s$ .

491 By Lemma 15, there is some typestate  $r \in R$  that is reachable via some subsequence of  
 492 the transitions that led to the actual typestate  $t = \tau(\rho(x_i))$  that  $v_i$  was in during  $e$  before  
 493 transition  $m_j$  was applied.

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

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

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

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

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

### 524 3.4 Discussion: Accumulating Sets vs. Accumulating Subsequences

525 Section 3 uses the term “accumulation” to refer to two subtly different things. Accumula-  
 526 tion analyses (Definition 4) compute *sets* of operations. Accumulation typestate systems  
 527 (Definition 7) are defined by *(sub)sequences* of operations.

528 Definition 4 of accumulation analysis uses sets because that is how accumulation analysis  
 529 is defined and implemented in prior work [35, 37]. For an alternate definition of accumulation  
 530 analysis in terms of subsequences, each goal operation would have an enabling sequence  
 531 rather than an enabling set. Implementing an accumulation analysis based on this alternate  
 532 definition would allow us to check “accumulation-like” properties that cannot be expressed  
 533 as sets. For example, such an analysis could soundly check a property such as “call  $\mathbf{a}()$  at  
 534 least twice before calling  $\mathbf{b}()$ ” (i.e., a goal transition enabled by counting) or a property such  
 535 as “call  $\mathbf{a}()$  and  $\mathbf{b}()$ , in that order, before calling  $\mathbf{c}()$ ” (i.e., a goal transition enabled by

536 ordering). This generalization of the *concept* of accumulation from the specific accumulation  
 537 analyses used in prior work is one of our contributions.

538 In our literature survey (Section 4), we found three specifications with a goal transition  
 539 enabled by ordering, but we did not find any enabled by counting. For example, in Figure  
 540 12 of [56], the authors describe a mined tpestate specification for the Java KeyAgreement  
 541 API. This API contains a method `generateSecret()`. Calling `generateSecret()` before `init()`  
 542 and `doPhase()` is an error, so `generateSecret()` is a goal transition. However, `init()` and  
 543 `doPhase()` also must be ordered: calling `doPhase()` before `init()` is also an error. The other  
 544 two specifications in the literature (which appear in [56, 22]) that rely on ordering had a  
 545 similar character to this example: describing some multi-stage initialization property where  
 546 the initialization steps must be performed in some specific order.

## 547 **4 Literature Survey**

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

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

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

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

### 564 **4.1 Methodology**

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

575 We then examined each of the remaining papers in detail and recorded how many tpestate  
 576 specifications they contained, which specifications those were, and which of the specifications  
 577 were accumulation tpestate systems. When recording which specifications occurred in  
 578 each paper we examined, we also recorded whether the specifications were duplicates of  
 579 specifications that appeared in other papers. Among the papers we examined, 102 ( $\approx 55\%$  of  
 580 those examined closely, and  $\approx 6\%$  of all Google Scholar hits) contained one or more tpestate



■ **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%
<b>Total</b>	All of the above	1355	555	41%

581 specifications. The venues that contributed papers with one or more tpestate specifications  
 582 to this study are: ECOOP (12), ESEC/FSE (12), ICSE (12), OOPSLA (10), PLDI (8),  
 583 ISSTA (7), ASE (6), POPL (5), CCS (4), SAS (4), TOSEM (4), TSE (4), CC (2), ASPLOS  
 584 (1), CAV (1), EuroSys (1), ICPC (1), IWACO (1), SAC (1), SOSP (1), TOPLAS (1), VMCAI  
 585 (1), WWW (1).

## 586 4.2 Results

587 Table 1 summarizes the results. This paper’s artifact<sup>9</sup> contains our analysis of each relevant  
 588 paper. The artifact also contains a finite-state machine for each tpestate problem (as defined  
 589 in Section 4.2.1 below) we saw and the list of the papers we saw it in.

### 590 4.2.1 Papers Containing Examples

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

592 22% of these specifications are accumulation tpestate systems. However, there is a  
 593 significant amount of duplication between the papers in this dataset—many papers use the  
 594 same few examples of tpestate automata to motivate their general work on tpestate.

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

605 Next, we give the three most common examples of tpestate problems that are accumula-  
 606 tion and are not accumulation tpestate systems.

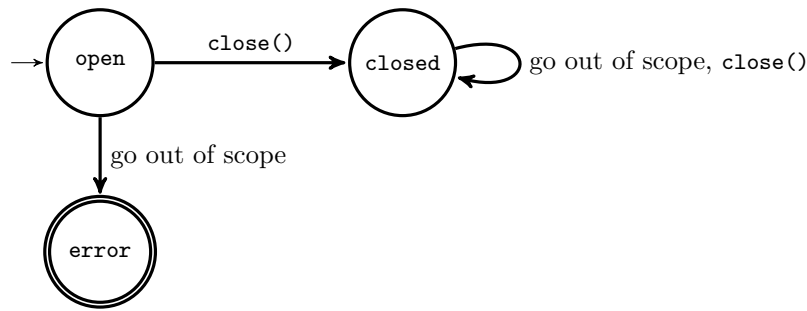
#### 607 4.2.1.1 Examples of Tpestate Problems That Are Accumulation

608 The problem of detecting resource leaks (Figure 6) appears 16 times across 14 papers<sup>10</sup> [17,  
 609 39, 72, 37, 64, 42, 43, 13, 21, 19, 3, 1, 63, 51]. This problem was already known to be

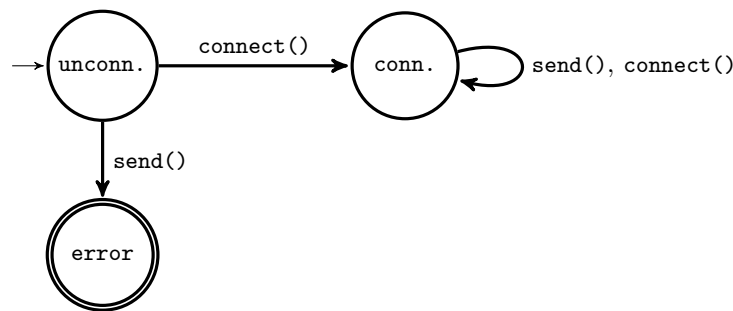
<sup>9</sup> <https://doi.org/10.5281/zenodo.5771196>

<sup>10</sup> 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.

## 10:18 Accumulation Analysis



■ **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.

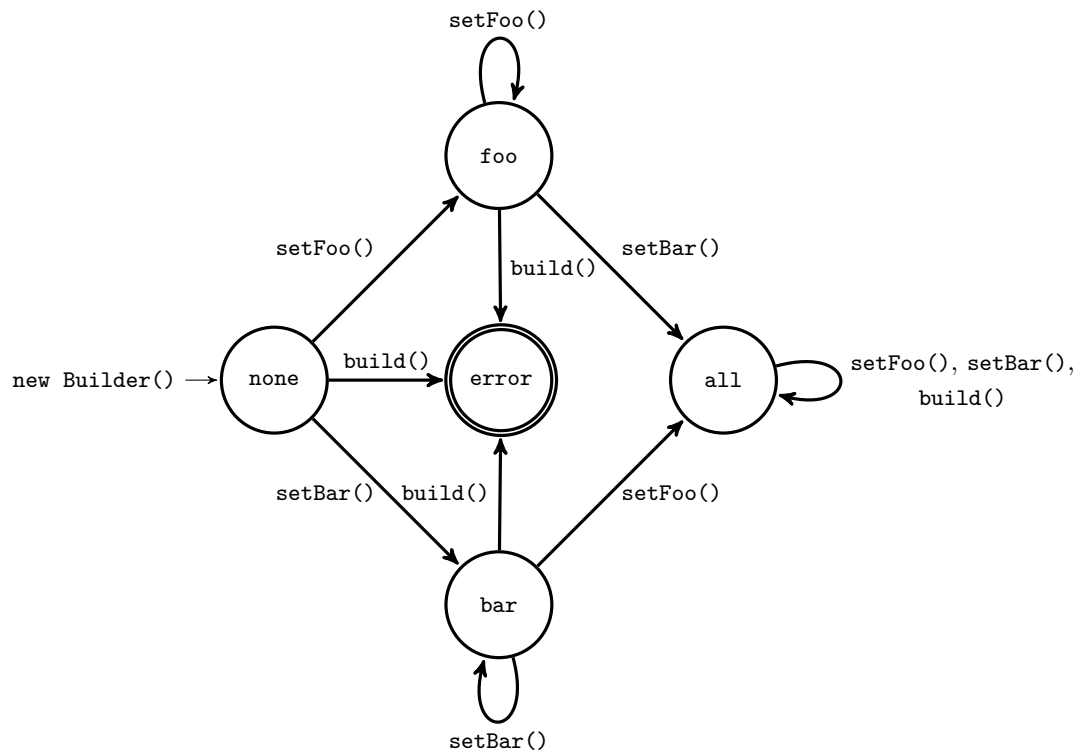
610 accumulation [37].

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

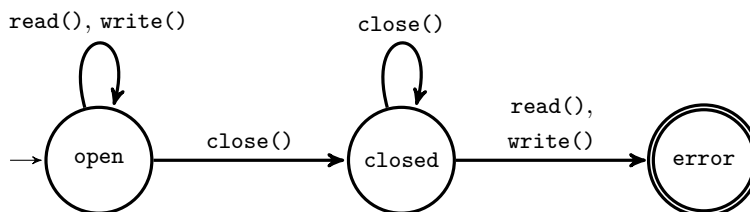
614 A third common accumulation problem is that of object initialization: before an object is  
615 fully constructed, all of its logically-required fields must be set to reasonable values (Figure 8).  
616 This pattern appears 6 times across 6 papers [35, 36, 54, 21, 27, 30]. A variant of this  
617 problem—which arises when using the builder pattern—was known to be accumulation [35].  
618 However, our literature survey has shown that bespoke analyses for other kinds of object  
619 initialization are also, in effect, bespoke accumulation analyses. For example, masked  
620 types [54] are a type system for ensuring that before a constructor exits, all non-null fields of  
621 the constructed class have been set to non-null values. This type system can be viewed as an  
622 accumulation analysis: the goal transition is the end of the constructor, and the enabling  
623 operations are the setting of the fields.

### 624 4.2.1.2 Examples of Typestate Problems That Are Not Accumulation

625 The most common non-accumulation typestate problem is “don’t read or write to a stream  
626 or file after it is closed” (Figure 9), which appeared 31 times across 17 papers [24, 8, 10, 46,  
627 25, 57, 5, 6, 53, 34, 44, 19, 71, 45, 69, 68, 11]. This problem is related to the file specification  
628 in Figure 1, but is slightly weaker—it assumes that the file is never re-opened. That this  
629 example is not accumulation demonstrates that accumulation typestate automata are a



■ **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`.

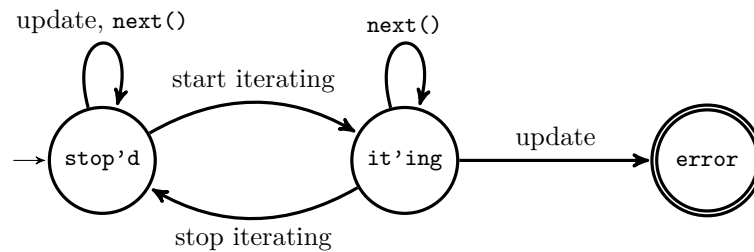


■ **Figure 9** The typestate automaton for not reading or writing a stream after it has been closed, which is not an accumulation typestate problem.

630 different category than automata without loops other than self-loops (a category that includes  
631 both this one and the three accumulation typestate examples in section 4.2.1.1).

632 “Do not update a collection while iterating over it” (Figure 10) appeared 21 times across  
633 14 papers [9, 65, 47, 26, 51, 68, 8, 10, 33, 32, 52, 53, 7, 46]. This property is representative of  
634 an important class of properties that are never accumulation typestate systems: “disable  $x$   
635 after  $y$ ” properties that forbid the programmer from performing operation  $x$  once operation  $y$   
636 has been performed. The key reason that these properties cannot be checked without aliasing  
637 information—and are therefore not accumulation—is that that the “disabling” operation  
638 (“start iterating” in this example) might be performed through any alias, but once it occurs,  
639 “update” must be prevented for all aliases.

## 10:20 Accumulation Analysis



■ **Figure 10** The tpestate automaton for not updating a collection during iteration, which is not an accumulation tpestate 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.).

640 The classic full file specification (Figure 1) appeared 20 times across 19 papers [28, 70,  
641 59, 25, 29, 62, 67, 57, 69, 66, 1, 49, 16, 38, 2, 15, 72, 19, 20]. Some parts of this specification  
642 could be enforced with an accumulation analysis if a slightly different design had been chosen  
643 for the API. In particular, if files could not be re-opened once they had been closed, enforcing  
644 “only call close after open” and “only call read after open” would become accumulation  
645 properties. Since most programmers usually create a new `File` object rather than re-using  
646 an existing one, this restriction would not be particularly burdensome, but would enable  
647 easier analysis.

### 648 4.2.2 Papers With Many Tpestates

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

#### 650 4.2.2.1 Patterns in Property Specifications for Finite-State Verification

651 The first paper reports on 555 tpestate-like specifications collected from a survey of 34  
652 papers from the scientific literature, verification tool authors, and students in 1999 [18].  
653 These 555 specifications were not de-duplicated. **[[It might be possible to de-duplicate  
654 these, but I don't think we have time right now. They did not say how many  
655 specifications came from which sources in their paper, either, but if we can find  
656 the data it might be there.]]** This paper inspired us to conduct the updated survey in  
657 Section 4.2.1. Because it precedes the start date for our survey, it is not included in the  
658 187 papers in Section 4.2.1. We include its data here for completeness, and to discuss the  
659 differences between their results and ours (Section 4.3).

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

#### 669 4.2.2.2 An Empirical Study of Object Protocols in the Wild

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

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

678 Instances of their most common category, Initialization, are always accumulation tpestate  
679 specifications. This category contains 182 of the 542 protocols (34%). The other 6 categories  
680 (66%) are not accumulation.

### 681 4.3 Discussion

682 Both of the papers that reported on large sets of tpestate properties included larger  
683 proportions of accumulation properties than our literature survey found otherwise. One  
684 possible explanation is that papers on novel analysis techniques tend to include “exciting” or  
685 “challenging” problems—and, in the case of general tpestate analysis, those problems usually  
686 involve aliasing (perhaps to justify the need for an alias analysis when analyzing an arbitrary  
687 tpestate system, as we do in Section 1 in reference to Figure 1). Another possible explanation  
688 is that neither of the papers that reported on large sets of specifications de-duplicated their  
689 specifications, so maybe they contain many duplicate accumulation properties. When we  
690 de-duplicated the specifications in Section 4.2.1, we found that non-accumulation tpestate  
691 properties tended to be duplicated more often than accumulation tpestate properties. This  
692 suggests that our results may be understating the prevalence of accumulation properties. If  
693 our results understate how common accumulation properties are in practice, that is good  
694 news for practitioners interested in applying verification: we have shown that accumulation  
695 properties are easier to check than general tpestate properties.

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

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

714 **5 Practicality of Accumulation Analysis**

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

725 We tested our implementations on the test suites of the bespoke analyses from our prior  
 726 work and on the case studies that those papers describe, and found that the implementations  
 727 using the common framework produced the expected results. The test suites contain both  
 728 positive examples (i.e., expected errors) and negative examples (i.e., safe code). The test  
 729 suites consist of 153 source files comprising 5,452 lines of non-comment, non-blank Java code.  
 730 The case studies together comprise 635,006 lines of non-comment, non-blank Java code.

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

- 733 ■ An accumulation analysis for verifying the absence of an initialization-related security  
 734 vulnerability had 100% recall (as this paper proves, the accumulation analysis was  
 735 sound!) and 82% precision—16 true bugs vs. 3 false positives—in 9 million non-comment,  
 736 non-blank lines of Java code (table 1 of [35]).
- 737 ■ An accumulation analysis for verifying the absence of resource leaks had 100% recall and  
 738 26% precision on 3 pieces of distributed-systems infrastructure used as a benchmark by  
 739 prior work (table 4 of [37]). This compares favorably to the 13% recall and 25% precision  
 740 achieved by an unsound heuristic bug-finder and the 7% recall and 50% precision achieved  
 741 by a state-of-the-art typestate-based analysis that uses a (very slow) whole-program alias  
 742 analysis. This precision might seem disappointing for a bug-finding tool, but we think it  
 743 is acceptable for a verification tool — especially for an important and difficult problem  
 744 such as resource leaks.

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

---

<sup>11</sup><https://checkerframework.org/manual/#accumulation-checker>

## 757 5.1 Aliasing in Practical Accumulation Analyses

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

764 Our prior work [35, 37] used cheap, targeted must-alias reasoning to improve the precision—  
765 that is, the false positive rate—of the analyses. For example, section 4.3 of [35] and sections  
766 3–5 of [37] give lightweight aliasing analyses. These lightweight alias analyses compute only  
767 the aliasing information necessary to remove false positives that occurred in practice for  
768 these analyses, which makes them much cheaper than computing precise aliasing information  
769 for all variables (of types with tpestate automata) in the program, as a whole-program alias  
770 analysis would.

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

## 774 5.2 Handling Other Features of Real Programming Languages

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

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

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

795 An advantage of our choice of a pluggable type system to implement our accumulation  
796 analyses is that the “start state” of a field can be changed by changing its declared type to  
797 specify a different tpestate. This restricts that field to only contain values whose tpestates  
798 are in the states reachable from the declared tpestate—that is, the sub-automaton composed  
799 of states reachable from the declared type. For the accumulation analyses we implemented,  
800 we found that this ability to refine a field’s declared type to be sufficient to enable precise  
801 analysis of field reads.

802 **6 Related Work**803 **6.1 Previous Work on Accumulation**

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

814 **6.2 Heap Monotonic Tpestates**

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

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

829 **6.3 Other Categories of Tpestate Systems**

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



## 845 6.4 Typestate Surveys

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

## 850 6.5 Practical Typestate Analyses

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

## 860 6.6 Typestate With Aliasing Restrictions

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

## 865 6.7 Other Work on Typestate

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

## 869 7 Conclusion

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

---

<sup>12</sup><https://doi.org/10.5281/zenodo.5771196>

## 879 — References

- 880 1 Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle,  
881 and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In  
882 *International Static Analysis Symposium*, pages 230–246. Springer, 2002.
- 883 2 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Tpestate-oriented  
884 programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object  
885 oriented programming systems languages and applications*, pages 1015–1022, 2009.
- 886 3 Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: An efficient runtime for detecting  
887 defects in deployed systems. In *Proceedings of the 23rd ACM SIGPLAN conference on  
888 Object-oriented programming systems languages and applications*, pages 143–162, 2008.
- 889 4 Nels E. Beckman, Duri Kim, and Jonathan Aldrich. An empirical study of object protocols  
890 in the wild. In *European Conference on Object-Oriented Programming*, pages 2–26. Springer,  
891 2011.
- 892 5 Kevin Bierhoff and Jonathan Aldrich. Lightweight object specification with tpestates. *ACM  
893 SIGSOFT Software Engineering Notes*, 30(5):217–226, 2005.
- 894 6 Kevin Bierhoff and Jonathan Aldrich. Modular tpestate checking of aliased objects. *ACM  
895 SIGPLAN Notices*, 42(10):301–320, 2007.
- 896 7 Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical API protocol checking  
897 with access permissions. In *European Conference on Object-Oriented Programming*, pages  
898 195–219. Springer, 2009.
- 899 8 Eric Bodden. Efficient hybrid tpestate analysis by determining continuation-equivalent states.  
900 In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages  
901 5–14. IEEE, 2010.
- 902 9 Eric Bodden, Laurie Hendren, and Ondřej Lhoták. A staged static program analysis to  
903 improve the performance of runtime monitoring. In *European Conference on Object-Oriented  
904 Programming*, pages 525–549. Springer, 2007.
- 905 10 Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by  
906 evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT  
907 International Symposium on Foundations of software engineering*, pages 36–47, 2008.
- 908 11 Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection:  
909 Aiding static analysis in the presence of reflection and custom class loaders. In *2011 33rd  
910 International Conference on Software Engineering (ICSE)*, pages 241–250. IEEE, 2011.
- 911 12 Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin  
912 Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation.  
913 In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages  
914 45–58, 2009.
- 915 13 Sigmund Chereh, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection  
916 using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on  
917 Programming Language Design and Implementation*, pages 480–491, 2007.
- 918 14 Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In  
919 *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, Berlin,  
920 Heidelberg, 2013.
- 921 15 Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in  
922 polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming  
923 language design and implementation*, pages 57–68, 2002.
- 924 16 Markus Degen, Peter Thiemann, and Stefan Wehr. Tracking linear and affine resources with  
925 Java(X). In *European Conference on Object-Oriented Programming*, pages 550–574. Springer,  
926 2007.
- 927 17 Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software.  
928 In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and  
929 implementation*, pages 59–69, 2001.

- 930 18 Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifica-  
931 tions for finite-state verification. In *International Conference on Software Engineering*, pages  
932 411–420, 1999.
- 933 19 Matthew B. Dwyer, Madeline Diep, and Sebastian Elbaum. Reducing the cost of path  
934 property monitoring through sampling. In *2008 23rd IEEE/ACM International Conference on*  
935 *Automated Software Engineering*, pages 228–237. IEEE, 2008.
- 936 20 Matthew B. Dwyer and Rahul Purandare. Residual dynamic tpestate analysis exploiting  
937 static analysis: results to reformulate and reduce the cost of dynamic analysis. In *Proceedings*  
938 *of the twenty-second IEEE/ACM international conference on Automated software engineering*,  
939 pages 124–133, 2007.
- 940 21 Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra  
941 Sengupta, and Willem Visser. RAPID: Checking API usage for the cloud in the cloud. In  
942 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference*  
943 *and Symposium on the Foundations of Software Engineering*, pages 1416–1426, 2021.
- 944 22 Manuel Fähndrich and K. Rustan M. Leino. Heap monotonic tpestates. In *IWACO 2003: In-*  
945 *ternational Workshop on Aliasing, Confinement and Ownership in object-oriented programming*,  
946 pages 58–72, Darmstadt, Germany, July 2003.
- 947 23 John Field, Deepak Goyal, G. Ramalingam, and Eran Yahav. Tpestate verification: Abstrac-  
948 tion techniques and complexity results. In *International Static Analysis Symposium*, pages  
949 439–462. Springer, 2003.
- 950 24 Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective  
951 tpestate verification in the presence of aliasing. *ACM Transactions on Software Engineering*  
952 *and Methodology (TOSEM)*, 17(2):1–34, 2008.
- 953 25 Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings*  
954 *of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*,  
955 pages 1–12, 2002.
- 956 26 Asya Frumkin, Yotam M. Y. Feldman, Ondřej Lhoták, Oded Padon, Mooly Sagiv, and Sharon  
957 Shoham. Property directed reachability for proving absence of concurrent modification errors.  
958 In *International Conference on Verification, Model Checking, and Abstract Interpretation*,  
959 pages 209–227. Springer, 2017.
- 960 27 Mark Gabel and Zhendong Su. Testing mined specifications. In *Proceedings of the ACM*  
961 *SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages  
962 1–11, 2012.
- 963 28 Qi Gao, Wenbin Zhang, Zhezhe Chen, Mai Zheng, and Feng Qin. 2ndStrike: toward manifesting  
964 hidden concurrency tpestate bugs. *ACM Sigplan Notices*, 46(3):239–250, 2011.
- 965 29 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of tpestate-  
966 oriented programming. *ACM Transactions on Programming Languages and Systems*  
967 *(TOPLAS)*, 36(4):1–44, 2014.
- 968 30 Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In  
969 *European Conference on Object-Oriented Programming*, pages 520–545. Springer, 2009.
- 970 31 Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London*  
971 *Mathematical Society*, 3(1):326–336, 1952.
- 972 32 Jeff Huang, Qingzhou Luo, and Grigore Rosu. GPredict: Generic predictive concurrency  
973 analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*,  
974 volume 1, pages 847–857. IEEE, 2015.
- 975 33 Dongyun Jin, Patrick O’Neil Meredith, Dennis Griffith, and Grigore Rosu. Garbage collection  
976 for monitoring parametric properties. *ACM SIGPLAN Notices*, 46(6):415–424, 2011.
- 977 34 Pallavi Joshi and Koushik Sen. Predictive tpestate checking of multithreaded Java programs.  
978 In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages  
979 288–296. IEEE, 2008.

- 980 35 Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäfer, and Michael D. Ernst. Verifying  
981 object construction. In *ICSE 2020, Proceedings of the 42nd International Conference on*  
982 *Software Engineering*, pages 1447–1458, Seoul, Korea, May 2020.
- 983 36 Martin Kellogg, Martin Schäfer, Serdar Tasiran, and Michael D. Ernst. Continuous compliance.  
984 In *ASE 2020: Proceedings of the 35th Annual International Conference on Automated Software*  
985 *Engineering*, pages 511–523, Melbourne, Australia, Sep. 2020.
- 986 37 Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and  
987 modular resource leak verification. In *ESEC/FSE 2021: The ACM 29th joint European*  
988 *Software Engineering Conference and Symposium on the Foundations of Software Engineering*  
989 *(ESEC/FSE)*, Athens, Greece, Aug. 2021.
- 990 38 Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order pro-  
991 grams. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles*  
992 *of Programming Languages*, pages 416–428, 2009.
- 993 39 Goh Kondoh and Tamiya Onodera. Finding bugs in Java Native Interface programs. In  
994 *Proceedings of the 2008 international symposium on Software testing and analysis*, pages  
995 109–118, 2008.
- 996 40 Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proceedings of the 29th*  
997 *ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 17–32,  
998 2002.
- 999 41 William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification.  
1000 In *POPL '91: Proceedings of the Eighteenth Annual ACM Symposium on Principles of*  
1001 *Programming Languages*, pages 93–103, Orlando, FL, Jan. 1991.
- 1002 42 Wei Le and Mary Lou Soffa. Marple: Detecting faults in path segments using automatically  
1003 generated analyses. *ACM Transactions on Software Engineering and Methodology (TOSEM)*,  
1004 22(3):1–38, 2013.
- 1005 43 Junhee Lee, Seongjoon Hong, and Hakjoo Oh. Memfix: static analysis-based repair of memory  
1006 deallocation errors for C. In *Proceedings of the 2018 26th ACM Joint Meeting on European*  
1007 *Software Engineering Conference and Symposium on the Foundations of Software Engineering*,  
1008 pages 95–106, 2018.
- 1009 44 Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. Program tailoring: Slicing by sequential  
1010 criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*.  
1011 Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- 1012 45 Filipe Militao, Jonathan Aldrich, and Luís Caires. Rely-guarantee protocols. In *European*  
1013 *Conference on Object-Oriented Programming*, pages 334–359. Springer, 2014.
- 1014 46 Nomair A. Naeem and Ondrej Lhoták. Typestate-like analysis of multiple interacting objects.  
1015 *ACM Sigplan Notices*, 43(10):347–366, 2008.
- 1016 47 Mangala Gowri Nanda, Christian Grothoff, and Satish Chandra. Deriving object typestates  
1017 in the presence of inter-object references. In *Proceedings of the 20th annual ACM SIGPLAN*  
1018 *conference on Object-oriented programming, systems, languages, and applications*, pages 77–96,  
1019 2005.
- 1020 48 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst.  
1021 Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International*  
1022 *Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 2008.
- 1023 49 Hila Peleg, Sharon Shoham, Eran Yahav, and Hongseok Yang. Symbolic automata for static  
1024 specification mining. In *International Static Analysis Symposium*, pages 63–83. Springer, 2013.
- 1025 50 Goran Piskachev, Tobias Petrasch, Johannes Späth, and Eric Bodden. AuthCheck: Program-  
1026 state analysis for access-control vulnerabilities. In *International Symposium on Formal Methods*,  
1027 pages 557–572. Springer, 2019.
- 1028 51 Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R. Gross. Statically checking  
1029 API protocol conformance with mined multi-object specifications. In *2012 34th International*  
1030 *Conference on Software Engineering (ICSE)*, pages 925–935. IEEE, 2012.

- 1031 52 Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitor optimization via  
1032 stutter-equivalent loop transformation. In *Proceedings of the ACM international conference*  
1033 *on Object oriented programming systems languages and applications*, pages 270–285, 2010.
- 1034 53 Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Optimizing monitoring of  
1035 finite state properties through monitor compaction. In *Proceedings of the 2013 International*  
1036 *Symposium on Software Testing and Analysis*, pages 280–290, 2013.
- 1037 54 Xin Qi and Andrew C. Myers. Masked types for sound object initialization. *ACM SIGPLAN*  
1038 *Notices*, 44(1):53–65, 2009.
- 1039 55 Rust team. Rust programming language. <https://www.rust-lang.org/>, 2021. Accessed 30  
1040 November 2021.
- 1041 56 Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification  
1042 mining using automata-based abstractions. *IEEE Transactions on Software Engineering*,  
1043 34(5):651–666, 2008.
- 1044 57 Johannes Späth, Karim Ali, and Eric Bodden. IDEal: Efficient and precise alias-aware dataflow  
1045 analysis. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–27, 2017.
- 1046 58 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for  
1047 enhancing software reliability. *IEEE TSE*, SE-12(1):157–171, Jan. 1986.
- 1048 59 Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. First-class  
1049 state change in Plaid. In *OOPSLA 2011, Object-Oriented Programming Systems, Languages,*  
1050 *and Applications*, pages 713–732, Portland, OR, USA, Oct. 2011.
- 1051 60 Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. Making pointer analysis  
1052 more precise by unleashing the power of selective context sensitivity. *Proceedings of the ACM*  
1053 *on Programming Languages*, 5(OOPSLA):1–27, 2021.
- 1054 61 Jesse A. Tov and Riccardo Pucella. Practical affine types. *ACM SIGPLAN Notices*, 46(1):447–  
1055 458, 2011.
- 1056 62 Cláudio Vasconcelos and António Ravara. From object-oriented code with assertions to  
1057 behavioural types. In *Proceedings of the Symposium on Applied Computing*, pages 1492–1497,  
1058 2017.
- 1059 63 Haowei Wu, Shengqian Yang, and Atanas Rountev. Static detection of energy defect patterns  
1060 in android applications. In *Proceedings of the 25th International Conference on Compiler*  
1061 *Construction*, pages 185–195, 2016.
- 1062 64 Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang.  
1063 Light-weight, inter-procedural and callback-aware resource leak detection for android apps.  
1064 *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.
- 1065 65 Xusheng Xiao, Gogul Balakrishnan, Franjo Ivančić, Naoto Maeda, Aarti Gupta, and Deepak  
1066 Chhetri. Arc++: effective typestate and lifetime dependency analysis. In *Proceedings of the*  
1067 *2014 International Symposium on Software Testing and Analysis*, pages 116–126, 2014.
- 1068 66 Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary  
1069 Sevitsky. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions*  
1070 *on Software Engineering and Methodology (TOSEM)*, 23(3):1–50, 2014.
- 1071 67 Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure  
1072 summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on*  
1073 *Principles of programming languages*, pages 221–234, 2008.
- 1074 68 Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. Symbolic verification of  
1075 regular properties. In *2018 IEEE/ACM 40th International Conference on Software Engineering*  
1076 *(ICSE)*, pages 871–881. IEEE, 2018.
- 1077 69 Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in  
1078 multithreaded applications. In *Proceedings of the 2014 International Symposium on Software*  
1079 *Testing and Analysis*, pages 1–12, 2014.
- 1080 70 Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric  
1081 dataflow analysis. *ACM SIGPLAN Notices*, 48(6):365–376, 2013.

- 1082 71 Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. Regular property  
1083 guided dynamic symbolic execution. In *2015 IEEE/ACM 37th IEEE International Conference*  
1084 *on Software Engineering*, volume 1, pages 643–653. IEEE, 2015.
- 1085 72 Zhiqiang Zuo, John Thorpe, Yifei Wang, Qihong Pan, Shenming Lu, Kai Wang, Guo-  
1086 qing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static  
1087 finite-state property checking of large-scale systems code. In *EuroSys*, pages 1–17, 2019.

## 1088 **A** Proof of Lemma 15

1089 This appendix contains the full proof of Lemma 15, which appears in section 3.3.1.6 and is  
1090 used by Lemma 16, the forwards direction of the proof of Theorem 9. We begin by restating  
1091 Lemma 15:

1092 ► **Lemma 15.** *Let  $R = \phi_s(x_i)$  be the set of estimated tpestates produced by a tpestate*  
1093 *analysis with no aliasing information for a variable  $x_i$  before a statement  $s$ . Let  $S$  be the*  
1094 *trace of an arbitrary execution leading up to some occurrence of  $s$ , and let  $t = \tau(\rho(x_i))$  be*  
1095 *the tpestate of the actual value to which  $x_i$  refers before that occurrence of  $s$ . Applying  $S$  to*  
1096 *the automaton leads to tpestate  $t$ . There exists a tpestate  $r \in R$  such that applying some*  
1097 *subsequence of  $S$  leads to  $r$ . That is, there is some estimated tpestate  $r \in R$  that is reachable*  
1098 *by a subsequence of the transitions that lead to  $t$ .*

1099 The proof is by co-induction on the dynamic semantics of the language in Figure 2 and  
1100 the definition of a tpestate analysis with no aliasing information in Definition 14, with one  
1101 change to its rule for load operations (rule TS-LOAD-FIX in Figure 5). In particular, the  
1102 load rule our tpestate analysis with no aliasing uses in this proof is the following:

1103 ■ For a load statement  $s$ , where  $s$  is  $x_i := x_j.f_k$ , let  $s_0$  be the start state of the automaton  
1104  $A$  which is being checked. The analysis updates its estimate for  $x_i$  so that it is mapped  
1105 to  $s_0$ :  $\phi'_s(x_i) := s_0$ . For all other names  $m$  in  $\phi_s$  where  $m \neq x_i$ , the analysis copies the  
1106 entry from the previous abstract store:  $\phi'_s(m) := \phi_s(m)$ .

1107 (See the discussion of why this modified rule is necessary in section 3.3.1.6, after the  
1108 original statement of Lemma 15.)

1109 **Proof. Base case:** when a program begins executing, the dynamic semantics say that all  
1110 names refer to values in the start state. A tpestate analysis with no aliasing information  
1111 estimates that at a program’s entry point, all names are in the start state, as well. Trivially,  
1112 the start state is reachable by the same sequence of operations as itself.

1113 **Case assignment:** For an assignment  $s$ , where  $s$  is  $x_i := x_j$ , the invariant is preserved  
1114 by the inductive hypothesis. Consider that by the inductive hypothesis, the invariant is  
1115 preserved for  $x_j$ . Then consider the rule used by the tpestate analysis with no aliasing  
1116 information for an assignment: every mention of  $x_i$  in the abstract store is replaced by  $x_j$ .  
1117 Further, the dynamic semantics for an assignment require that the previous value of  $x_i$  is no  
1118 longer accessible via  $x_i$ :  $x_i$  after the assignment refers only to  $x_j$ . Since  $x_i$  and  $x_j$  after the  
1119 assignment are treated entirely the same, but the abstract store is otherwise unchanged by  
1120 the analysis, what was true of  $x_j$  before the statement is true for  $x_i$  after.

1121 **Case load:** The special load rule TS-LOAD-FIX trivially guarantees that the invariant  
1122 is preserved: the start state is reachable by a subsequence of the operations that reach any  
1123 other state (in particular, by the empty subsequence).

1124 **Case store:** This rule trivially preserves the invariant, because the invariant must be  
1125 maintained only for the estimates for variables—not for fields—and rule TS-STORE only  
1126 updates estimates for fields.

1127     **Case method call:** For a method call  $s = x_i.m_j()$ , only steps 1 and 2 of rule TS-CALL  
1128 are applied, because a typestate analysis with no aliasing information never performs strong  
1129 or weak updates on possible aliases. The invariant is preserved via the inductive hypothesis:  
1130 for  $x_i$  itself, let  $r_1$  be the element of  $R$  that is reachable by a subsequence of the actual  
1131 sequence  $S$  in the inductive hypothesis. The analysis updates its estimate to include  $r_1 + m_j$   
1132 (that is, the sequence  $r_1$  followed by the transition  $m_j$ ). After  $s$  is executed, the actual  
1133 sequence is  $S + m_j$ , and since we know that  $r_1$  is reachable by a subsequence of  $S$ ,  $r_1 + m_j$   
1134 must be reachable by a subsequence of  $S + m_j$ —the same subsequence used to reach  $r_1$ ,  
1135 with  $m_j$  added on. For any aliases of  $x_i$ , the inductive hypothesis also guarantees that the  
1136 invariant holds: the estimate contains some  $r$  that is a subsequence of  $S$ , and any subsequence  
1137 of  $S$  is also a subsequence of  $S + m_j$ .

1138     **Case sequence:** For a sequence, the invariant is trivially preserved by induction.

1139

