Scalable Typestate Analysis for Low-Latency Environments

Alen Arslanagić[1], Pavle Subotić[2], and Jorge A. Pérez[1]

[1] University of Groningen, The Netherlands
[2] Microsoft, Serbia

Abstract. Static analyses based on *typestates* are important in certifying correctness of code contracts. Such analyses rely on Deterministic Finite Automata (DFAs) to specify properties of an object. We target the analysis of contracts in low-latency environments, where many useful contracts are impractical to codify as DFAs and/or the size of their associated DFAs leads to sub-par performance. To address this bottleneck, we present a *lightweight* typestate analyzer, based on an expressive specification language that can succinctly specify code contracts. By implementing it in the static analyzer Infer, we demonstrate considerable performance and usability benefits when compared to existing techniques. A central insight is to rely on a sub-class of DFAs with efficient *bit-vector* operations.

1 Introduction

Industrial-scale software is generally composed of multiple interacting components, which are typically produced separately. As a result, software integration is a major source of bugs [18]. Many integration bugs can be attributed to violations of *code contracts*. Because these contracts are implicit and informal in nature, the resulting bugs are particularly insidious. To address this problem, formal code contracts are an effective solution [12], because static analyzers can automatically check whether client code adheres to ascribed contracts.

Typestate is a fundamental concept in ensuring the correct use of contracts and APIs. A typestate refines the concept of a type: whereas a type denotes the valid operations on an object, a typestate denotes operations valid on an object in its *current program context* [20]. Typestate analysis is a technique used to enforce temporal code contracts. In object-oriented programs, where objects change state over time, typestates denote the valid sequences of method calls for a given object. The behavior of the object is prescribed by the collection of typestates, and each method call can potentially change the object's typestate.

Given this, it is natural for static typestate checkers, such as Fugue [9], SAFE [23], and Infer's Topl checker [2], to define the analysis property using Deterministic Finite Automata (DFAs). The abstract domain of the analysis is a set of states in the DFA; each operation on the object modifies the set of possible reachable states. If the set of abstract states contains an error state, then the analyzer warns the user that a code contract may be violated. Widely applicable and conceptually simple, DFAs are the de facto model in typestate analyses.

Here we target the analysis of realistic code contracts in low-latency environments such as, e.g., Integrated Development Environments (IDEs) [22,21]. In this context, to avoid noticeable disruptions in the users' workflow, the analysis should ideally run *under a second* [1]. However, relying on DFAs jeopardizes this goal, as it can lead to scalability issues. Consider, e.g., a class with n methods in which each method *enables* another one and then *disables* itself: the contract can lead to a DFA with 2^n states. Even with a small n, such a contract can be impractical to codify manually and will likely result in sub-par performance.

Interestingly, many practical contracts do not require a full DFA. In our enable/disable example, the method dependencies are *local* to a subset of methods: a enabling/disabling relation is established between pairs of methods. DFA-based approaches have a *whole class* expressivity; as a result, local method dependencies can impact transitions of unrelated methods. Thus, using DFAs for contracts that specify dependencies that are local to each method (or to a few methods) is redundant and/or prone to inefficient implementations. Based on this observation, we present a *lightweight* typestate analyzer for *locally dependent* code contracts in low-latency environments. It rests upon two insights:

1. *Allowed and disallowed sequences of method calls for objects can be succinctly specified without using DFAs.* To unburden the task of specifying typestates, we introduce *lightweight annotations* to specify *method dependencies* as annotations on methods. Lightweight annotations can specify code contracts for usage scenarios commonly encountered when using libraries such as File, Stream, Socket, etc. in considerably fewer lines of code than DFAs.
2. *A sub-class of DFAs suffices to express many useful code contracts.* To give semantics to lightweight annotations, we define *Bit-Vector Finite Automata (BFAs)*: a sub-class of DFAs whose analysis uses *bit-vector* operations. In many practical scenarios, BFAs suffice to capture information about the enabled and disabled methods at a given point. Because this information can be codified using bit-vectors, associated static analyses can be performed efficiently; in particular, our technique is not sensitive to the number of BFA states, which in turn ensures scalability with contract and program size.

We have implemented our lightweight typestate analysis in the industrial-strength static analyzer Infer [7]. Our analysis exhibits concrete usability and performance advantages and is expressive enough to encode many relevant typestate properties in the literature. On average, compared to state-of-the-art typestate analyses, our approach requires less annotations than DFA-based analyzers and does not exhibit slow-downs due to state increase. We summarise our contributions as follows:

– A specification language for typestates based on *lightweight annotations* (§2). Our language rests upon BFAs, a new sub-class of DFA based on bit-vectors.
– A lightweight analysis technique for code contracts, implemented in Infer (our artifact is available at [4]).[3]

[3] Our code is available at `https://github.com/aalen9/lfa.git`

– Extensive evaluations for our lightweight analysis technique, which demonstrate considerable gains in performance and usability (§4).

2 Bit-vector Typestate Analysis

2.1 Annotation Language

We introduce BFA specifications, which succinctly encode temporal properties by only describing *local method dependencies*, thus avoiding an explicit DFA specification. BFA specifications define code contracts by using atomic combinations of annotations '`@Enable`(n)' and '`@Disable`(n)', where n is a set of method names. Intuitively, '`@Enable`(n) m' asserts that invoking method m makes calling methods in n valid in a continuation. Dually, '`@Disable`(n) m' asserts that a call to m disables calls to all methods in n in the continuation. More concretely, we give semantics for BFA annotations by defining valid method sequences:

Definition 1 (Annotation Language). *Let $C = \{m_0, \dots, m_n\}$ be a set of method names where each $m_i \in C$ is annotated by*

$$\texttt{@Enable}(E_i)\ \texttt{@Disable}(D_i)\ m_i$$

where $E_i \subseteq C$, $D_i \subseteq C$, and $E_i \cap D_i = \emptyset$. Further, we have $E_0 \cup D_0 = C$. Let $s = x_0, x_1, x_2, \dots$ be a method sequence where each $x_i \in C$. A sequence s is valid (w.r.t. annotations) *if there is no substring $s' = x_i, \dots, x_k$ of s such that $x_k \in D_i$ and $x_k \notin E_j$, for $j \in \{i+1, \dots, k\}$.*

The formal semantics for these specifications is given in § 2.2. We note, if E_i or D_i is $\emptyset$ then we omit the corresponding annotation. Moreover, the BFA language can be used to derive other useful annotations defined as follows:

$$\texttt{@EnableOnly}(E_i)\ m_i \overset{\text{def}}{=} \texttt{@Enable}(E_i)\ \texttt{@Disable}(C \setminus E_i)\ m_i$$

$$\texttt{@DisableOnly}(D_i)\ m_i \overset{\text{def}}{=} \texttt{@Disable}(D_i)\ \texttt{@Enable}(C \setminus E_i)\ m_i$$

$$\texttt{@EnableAll}\ m_i \overset{\text{def}}{=} \texttt{@Enable}(C)\ m_i$$

This way, '`@EnableOnly`(E_i) m_i' asserts that a call to method m_i enables only calls to methods in E_i while disabling all other methods in C; '`@DisableOnly`(D_i) m_i' is defined dually. Finally, '`@EnableAll` m_i' asserts that a call to method m_i enables all methods in a class; '`@DisableAll` m_i' can be defined dually.

To illustrate the expressivity and usability of BFA annotations, we consider the `SparseLU` class from `Eigen C++` library[4]. For brevity, we consider representative methods for a typestate specification (we also omit return types):

```
class SparseLU {
    void analyzePattern(Mat a);
    void factorize(Mat a);
    void compute(Mat a);
    void solve(Mat b);  }
```

[4] `https://eigen.tuxfamily.org/dox/classEigen_1_1SparseLU.html`

```
class SparseLU {
    states q0, q1, q2, q3;
    @Pre(q0) @Post(q1)
    @Pre(q3) @Post(q1)
    void analyzePattern(Mat a);
    @Pre(q1) @Post(q2)
    @Pre(q3) @Post(q2)
    void factorize(Mat a);
    @Pre(q0) @Post(q2)
    @Pre(q3) @Post(q2)
    void compute(Mat a);
    @Pre(q2) @Post(q3)
    @Pre(q3)
    void solve(Mat b);  }
```

Listing (1.1) SparseLU DFA Contract

```
class SparseLU {


    @EnableOnly(factorize)
    void analyzePattern(Mat a);

    @EnableOnly(solve)
    void factorize(Mat a);

    @EnableOnly(solve)
    void compute(Mat a);

    @EnableAll
    void solve(Mat b);  }
```

Listing (1.2) SparseLU BFA Contract

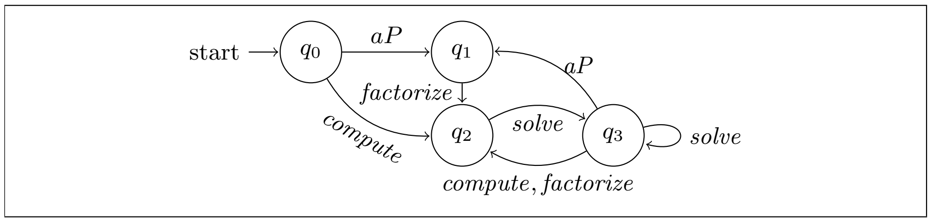

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[r, "aP"] \arrow[dr, "compute"'] & q_1 \arrow[d, "factorize"] & \\
& & q_2 \arrow[r, bend left, "solve"] & q_3 \arrow[l, bend left, "{compute, factorize}"] \arrow[ul, bend right, "aP"'] \arrow[loop right, "solve"]
\end{tikzcd}$$

Fig. 2: SparseLU DFA

The `SparseLU` class implements a lower-upper (LU) decomposition of a sparse matrix. `Eigen`'s implementation uses assertions to dynamically check that: (i) `analyzePattern` is called prior to `factorize` and (ii) `factorize` or `compute` are called prior to `solve`. At a high-level, this contract tells us that `compute` (or `analyzePattern().factorize()`) prepares resources for invoking `solve`.

We notice that there are method call sequences that do not cause errors, but have redundant computations. For example, we can disallow consecutive calls to `compute` as in, e.g., sequences like '`compute().compute().solve()`' as the result of the first compute is never used. Further, `compute` is essentially implemented as '`analyzePattern`().`factorize`()'. Thus, it is also redundant to call `factorize` after `compute`. The DFA that substitutes dynamic checks and avoids redundancies is given in Figure 2. Following the literature [9], this DFA can be annotated inside a class definition as in Listing 1.1. Here states are listed in the class header and transitions are specified by *@Pre* and *@Post* conditions on methods. However, this specification is too low-level and unreasonable for software engineers to annotate their APIs with, due to high annotation overheads.

In contrast, using BFA annotations the entire `SparseLU` class contract can be succinctly specified as in Listing 1.2. Here, the starting state is unspecified; it is determined by annotations. In fact, methods that are not *guarded* by other methods (like `solve` is guarded by `compute`) are enabled in the starting state. Concretely, methods are guarded if they are enabled by some method and not disabled by any other method. We remark that this can be overloaded by specifying annotations on the constructor method. We can specify the contract with only 4 annotations; the corresponding DFA requires 8 annotations and

4 states specified in the class header. We remark that a small change in local method dependencies by BFA annotations can result in a substantial change of the equivalent DFA. Let $\{m_1, m_2, m_3, \ldots, m_n\}$ be methods of some class with DFA associated (with states Q) in which m_1 and m_2 are enabled in each state of Q. Adding `@Enable(m2) m1` doubles the number of states of the DFA as we need the set of states Q where m_2 is enabled in each state, but also states from Q with m_2 disabled in each state. Accordingly, transitions have to be duplicated for the new states and the remaining methods $(m_3, \ldots, m_n)$.

2.2 Bit-vector Finite Automata

We define a class of DFAs, dubbed Bit-vector Finite Automata (BFA), that captures enabling/disabling dependencies between the methods of a class leveraging a bit-vector abstraction on typestates.

Definition 2 (Sets and Bit-vectors). *Let $\mathcal{B}^n$ denote the set of bit-vectors of length $n > 0$. We write $b, b', \ldots$ to denote elements of $\mathcal{B}^n$, with $b[i]$ denoting the i-th bit in b. Given a finite set S with $|S| = n$, every $A \subseteq S$ can be represented by a bit-vector $b_A \in \mathcal{B}^n$, obtained via the usual characteristic function. By a small abuse of notation, given sets $A, A' \subseteq S$, we may write $A \subseteq A'$ to denote the subset operation applied on b_A and $b_{A'}$ (and similarly for $\cup, \cap$).*

We first define a BFA per class. Let us write $\mathcal{C}$ to denote the finite set of all classes $c, c', \ldots$ under consideration. Given a $c \in \mathcal{C}$ with n methods, and assuming a total order on method names, we represent them by the set $\Sigma_c = \{m_1, \ldots, m_n\}$.

A BFA for a class with n methods considers states q_b, where, following Def. 2, the bit-vector $b_A \in \mathcal{B}^n$ denotes the set $A \subseteq \Sigma_c$ enabled at that point. We often write 'b' (and q_b) rather than 'b_A' (and 'q_{b_A}'), for simplicity. As we will see, the intent is that if $m_i \in b$ (resp. $m_i \notin b$), then the i-th method is enabled (resp. disabled) in q_b. Def. 3 will give a mapping from methods to triples of bit-vectors.

Given $k > 0$, let us write 1^k (resp. 0^k) to denote a sequence of 1s (resp. 0s) of length k. The initial state of the BFA is then $q_{10^{n-1}}$, i.e., the state in which only the first method is enabled and all the other $n-1$ methods are disabled.

Given a class c, we define its associated mapping $\mathcal{L}_c$ as follows:

Definition 3 (Mapping $\mathcal{L}_c$). *Given a class c, we define $\mathcal{L}_c$ as a mapping from methods to triples of subsets of Σ_c as follows*

$$\mathcal{L}_c : \Sigma_c \to \mathcal{P}(\Sigma_c) \times \mathcal{P}(\Sigma_c) \times \mathcal{P}(\Sigma_c)$$

Given $m_i \in \Sigma_c$, we shall write E_i, D_i and P_i to denote each of the elements of the triple $\mathcal{L}_c(m_i)$. The mapping $\mathcal{L}_c$ is induced by the annotations in class c: for each m_i, the sets E_i and D_i are explicit, and P_i is simply the singleton $\{m_i\}$.

In an BFA, transitions between states $q_b, q_{b'}, \cdots$ are determined by $\mathcal{L}_c$. Given $m_i \in \Sigma_c$, we have $j \in E_i$ if and only if the m_i enables m_j; similarly, $k \in D_i$ if and only if m_i disables m_k. A transition from q_b labeled by method m_i leads to state

$q_{b'}$, where b' is determined by $\mathcal{L}_c$ using b. Such a transition is defined only if a pre-condition for m_i is met in state q_b, i.e., $P \subseteq b$. In that case, $b' = (b \cup E_i) \setminus D_i$.

These intuitions should suffice to illustrate our approach and, in particular, the local nature of enabling and disabling dependencies between methods. The following definition makes them precise.

Definition 4 (BFA). *Given a $c \in \mathcal{C}$ with $n > 0$ methods, a Bit-vector Finite Automaton (BFA) for c is defined as a tuple $M = (Q, \Sigma_c, \delta, q_{10^{n-1}}, \mathcal{L}_c)$ where:*

- *Q is a finite set of states $q_{10^{n-1}}, q_b, q_{b'}, \ldots$, where $b, b', \ldots \in \mathcal{B}^n$;*
- *$q_{10^{n-1}}$ is the initial state;*
- *$\Sigma_c = \{m_1, \ldots, m_n\}$ is the alphabet (method identities);*
- *$\mathcal{L}_c$ is a BFA mapping (cf. Def. 3);*
- *$\delta : Q \times \Sigma_c \to Q$ is the transition function, where $\delta(q_b, m_i) = q_{b'}$ (with $b' = (b \cup E_i) \setminus D_i$) if $P_i \subseteq b$, and is undefined otherwise.*

We remark that in a BFA all states in Q are accepting states.

Example 1 (SparseLU). We give the BFA derived from the annotations in the SparseLU example (Listing 1.2). We associate indices to methods:

$$[0 : \mathit{constructor}, 1 : \mathit{aP}, 2 : \mathit{compute}, 3 : \mathit{factorize}, 4 : \mathit{solve}]$$

The constructor annotations are implicit: it enables methods that are not guarded by annotations on other methods (in this case, *aP* and *compute*). The mapping $\mathcal{L}_{\text{SparseLU}}$ is as follows:

$$\begin{aligned}\mathcal{L}_{\text{SparseLU}} = \{&0 \mapsto (\{1,2\}, \{\}, \{0\}),\ 1 \mapsto (\{3\}, \{1,2,4\}, \{1\}),\\ &2 \mapsto (\{4\}, \{1,2,3\}, \{2\}),\ 3 \mapsto (\{4\}, \{1,2,3\}, \{3\}), 4 \mapsto (\{1,2,3\}, \{\}, \{4\})\}\end{aligned}$$

The set of states is $Q = \{q_{1000}, q_{1100}, q_{0010}, q_{0001}, q_{1111}\}$ and the transition function δ is given by following nine transitions:

$$\begin{array}{lll}
\delta(q_{1000}, \mathit{constr}) = q_{1100} & \delta(q_{1100}, \mathit{aP}) = q_{0010} & \delta(q_{1100}, \mathit{compute}) = q_{0010}\\
\delta(q_{0010}, \mathit{factorize}) = q_{0001} & \delta(q_{0001}, \mathit{solve}) = q_{1111} & \delta(q_{1111}, \mathit{aP}) = q_{0010}\\
\delta(q_{1111}, \mathit{compute}) = q_{0001} & \delta(q_{1111}, \mathit{factorize}) = q_{0001} & \delta(q_{1111}, \mathit{solve}) = q_{1111}
\end{array}$$

BFAs vs DFAs First, we need define some convenient notations:

Definition 5 (Method sequences and concatenation). *We use $\widetilde{m}$ to denote a finite sequence of method names in Σ. Further, we use '$\cdot$' to denote sequence concatenation, defined as expected.*

In the following theorem, we use $\hat{\delta}(q_b, \widetilde{m})$ to denote the extension of the one-step transition function $\delta(q_b, m_i)$ to a sequence of method calls (i.e., $\widetilde{m}$). BFAs determine a strict sub-class of DFAs. First, because all states in Q are accepting states, BFA cannot encode the *"must call"* property (cf. § 5). Next, we define the *context-independency* property, satisfied by all BFAs but not by all DFAs:

Theorem 1 (Context-independency). *Let $M = (Q, \Sigma_c, \delta, q_{10^{n-1}}, \mathcal{L}_c)$ be a* BFA*. Also, let $L = \{\widetilde{m} : \hat{\delta}(q_{10^{n-1}}, \widetilde{m}) = q' \wedge q' \in Q\}$ be the language accepted by M. Then, for $m_n \in \Sigma_c$ we have*

1. *If there is $\widetilde{p} \in L$ and $m_{n+1} \in \Sigma_c$ s.t. $\widetilde{p} \cdot m_{n+1} \notin L$ and $\widetilde{p} \cdot m_n \cdot m_{n+1} \in L$ then there is no $\widetilde{m} \in L$ s.t. $\widetilde{m} \cdot m_n \cdot m_{n+1} \notin L$.*
2. *If there is $\widetilde{p} \in L$ and $m_{n+1} \in \Sigma_c$ s.t. $\widetilde{p} \cdot m_{n+1} \in L$ and $\widetilde{p} \cdot m_n \cdot m_{n+1} \notin L$ then there is no $\widetilde{m} \in L$ s.t. $\widetilde{m} \cdot m_n \cdot m_{n+1} \in L$.*

Proof. Directly by Def. 4. See [3] for details.

Informally, the above theorem tells that previous calls ($\widetilde{m}$) (*i.e.*, context) cannot impact the effect of a call to m_n to subsequent calls (m_{n+1}). That is, Item 1. (resp. Item 2.) tells that method m_n enables (resp. disables) the same set of methods in any context. For example, a DFA that disallows modifying a collection while iterating is not a BFA (as in Fig. 3 in [5]). Let *it* be a Java Iterator with its usual methods for collection *c*. For the illustration, we assume a single DFA relates the iterator and its collection methods. Then, the sequence '`it.hasNext;it.next;c.remove;it.hasNext`' should not be allowed, although '`c.remove;it.hasNext`' should be allowed. That is, `c.remove` disables `it.hasNext` *only if* `it.hasNext` is previously called. Thus, the effect of calling `c.remove` depends on the calls that precedes it.

BFA subsumption Using BFAs, checking class subsumption boils down to usual set inclusion. Suppose M_1 and M_2 are BFAs for classes c_1 and c_2, with c_2 being the superclass of c_1. The class inheritance imposes an important question on how we check that c_1 is a proper refinement of c_2. In other words, c_1 must subsume c_2: any valid sequence of calls to methods of c_2 must also be valid for c_1. Using BFAs, we can verify this simply by checking annotations method-wise. We can check whether M_2 subsumes M_1 only by considering their respective annotation mappings $\mathcal{L}_{c_2}$ and $\mathcal{L}_{c_1}$. Then, we have $M_2 \succeq M_1$ iff for all $m_j \in \mathcal{L}_{c_1}$ we have $E_1 \subseteq E_2$, $D_1 \supseteq D_2$, and $P_1 \subseteq P_2$ where $\langle E_i, D_i, P_i \rangle = \mathcal{L}_{c_i}(m_j)$ for $i \in \{1, 2\}$.

3 Compositional Analysis Algorithm

Since BFAs can be ultimately encoded as bit-vectors, for the non-compositional case e.g., intra-procedural, standard data-flow analysis frameworks can be employed [15]. However, in the case of member objects methods being called, we present a compositional algorithm that is tailored for the Infer compositional static analysis framework. We motivate our compositional analysis technique with the example below.

Example 2. Let `Foo` be a class that has member `lu` of class `SparseLU` (cf. Listing 1.3). For each method of `Foo` that invokes methods on `lu` we compute a *symbolic summary* that denotes the effect of executing that method on typestates of `lu`. To check against client code, a summary gives us: (i) a pre-condition (i.e., which methods should be allowed before calling a procedure) and (ii) the effect

```
class Foo {
  SparseLU lu; Matrix a;
  void setupLU1(Matrix b) {
    this.lu.compute(this.a);
    if (?) this.lu.solve(b); }
  void setupLU2() {
    this.lu.analyzePattern(this.a);
    this.lu.factorize(this.a); }
  void solve(Matrix b) {
    this.lu.solve(b); } }
```

Listing (1.3) Class Foo using SparseLU

```
void wrongUseFoo() {
  Foo foo; Matrix b;
  foo.setupLU1();
  foo.setupLU2();
  foo.solve(b);
}
```

Listing (1.4) Client code for Foo

on the *typestate* of an argument when returning from the procedure. A simple instance of a client is `wrongUseFoo` in Listing 1.4.

The central idea of our analysis is to accumulate enabling and disabling annotations. For this, the abstract domain maps object access paths to triplets from the definition of $\mathcal{L}_{\text{SparseLU}}$. A *transfer function* interprets method calls in this abstract state. We illustrate the transfer function, presenting how abstract state evolves as comments in the following code listing.

```
1 void setupLU1(Matrix b) {
2   // s1 = this.lu -> ({}, {}, {})
3   this.lu.compute(this.a);
4   // s2 = this.lu -> ({solve}, {aP, factorize, compute}, {compute})
5   if (?) this.lu.solve(b); }
6   // s3 = this.lu -> ({solve, aP, factorize, compute}, {}, {compute})
7   // join s2 s3 = s4
8   // s4 = sum1 = this.lu -> ({solve}, {aP, factorize, compute}, {compute})
```

At the procedure entry (line 2) we initialize the abstract state as a triplet with empty sets (s_1). Next, the abstract state is updated at the invocation of `compute` (line 3): we copy the corresponding tuple from $\mathcal{L}_{\text{SparseLU}}(compute)$ to obtain s_2 (line 4). Notice that `compute` is in the pre-condition set of s_2. Further, given the invocation of `solve` within the if-branch in line 5 we transfer s_2 to s_3 as follows: the enabling set of s_3 is the union of the enabling set from $\mathcal{L}_{\text{SparseLU}}(solve)$ and the enabling set of s_2 with the disabling set from $\mathcal{L}_{\text{SparseLU}}(solve)$ removed (i.e., an empty set here). Dually, the disabling set of s_3 is the union of the disabling set of $\mathcal{L}_{\text{SparseLU}}(solve)$ and the disabling set of s_1 with the enabling set of $\mathcal{L}_{\text{SparseLU}}(solve)$ removed. Here we do not have to add `solve` to the pre-condition set, as it is in the enabling set of s_2. Finally, we join the abstract states of two branches at line 7 (i.e., s_2 and s_3). Intuitively, join operates as follows: (i) a method is enabled only if it is enabled in both branches and not disabled in any branch; (ii) a method is disabled if it is disabled in either branch; (iii) a method called in either branch must be in the pre-condition (cf. Def. 6). Accordingly, in line 8 we obtain the final state s_4 which is also a summary for `SetupLU1`.
Now, we illustrate checking client code `wrongUseFoo()` with computed summaries:

```
void wrongUseFoo() {
  Foo foo; Matrix b;
  // d1 = foo.lu -> ({aP, compute}, {solve, factorize}, {})
  foo.setupLU1(); // apply sum1 to d1
  // d2 = foo.lu -> ({solve}, {aP, factorize, compute}, {})
  foo.setupLU2(); // apply sum2 = {this.lu -> ({solve}, {aP, factorize,
    compute}, {aP}) }
  // warning! 'analyzePattern' is in pre of sum2, but not enabled in d2
  foo.solve(b); }
```

Above, at line 2 the abstract state is initialized with annotations of constructor `Foo`. At the invocation of `setupLU1()` (line 4) we apply sum_1 in the same way as user-entered annotations are applied to transfer s_2 to s_3 above. Next, at line 6 we can see that `aP` is in the pre-condition set in the summary for `setupLU2()` (sum_2), computed similarly as sum_1, but not in the enabling set of the current abstract state d_2. Thus, a warning is raised: `foo.lu` set up by `foo.setupLU1()` is never used and overridden by `foo.setupLU2()`.

Class Composition In the above example, the allowed orderings of method calls to an object of class `Foo` are imposed by the contracts of its object members (`SparseLU`) and the implementation of its methods. In practice, a class can have multiple members with their own BFA contracts. For instance, class `Bar` can use two solvers `SparseLU` and `SparseQR`:

```
class Bar {
  SparseLU lu; SparseQR qr; /* ... */ }
```

where class `SparseQR` has its own BFA contract. The implicit contract of `Bar` depends on contracts of both `lu` and `qr`. Moreover, a class as `Bar` can be a member of some other class. Thus, we refer to those classes as *composed* and to classes that have declared contracts (as `SparseLU`) as *base classes*.

Integrating Aliasing Now, we discuss how *aliasing information* can be integrated with our technique. In Ex. 2 member `lu` of object `foo` can be aliased. Thus, we keep track of BFA triplets for all base members instead of constructing an explicit BFA contract for a composed class (e.g., `Foo`). Further, we would need to generalize an abstract state to a mapping of *alias sets* to BFA triplets. That is, the elements of abstract state would be $\{a_1, a_2, \ldots, a_n\} \mapsto \langle E, D, P\rangle$ where $\{a_1, a_2, \ldots, a_n\}$ is a set of access paths. For example, when invoking method `setupLU1` we would need to apply its summary (sum_1) to triplets of each alias set that contains `foo.lu` as an element. Let $d_1 = \{S_1 \mapsto t_1, S_2 \mapsto t_2, \ldots\}$ be an abstract state where S_1 and S_2 are the only keys such that `foo.lu` $\in S_i$ for $i \in \{1, 2\}$ and t_1 and t_2 are some BFA triplets.

```
1 // d1 = S1 -> t1, S2 -> t2, ...
2 foo.setupLU1(); // apply sum1 = {this.lu -> t3}
3 // d2 = S1 -> apply t3 to t1, S2 -> apply t3 to t2, ...
```

Above, at line 2 we would need to update bindings of S_1 and S_2 (.resp) by applying an BFA triplet for `this.foo` from sum_1, that is t_3, to t_1 and t_2 (.resp). The resulting abstract state d_2 is given at line 4. We remark that if a procedure does not alter aliases, we can soundly compute and apply summaries, as shown above.

Algorithm We formally define our analysis, which presupposes the control-flow graph (CFG) of a program. Let us write $\mathcal{AP}$ to denote the set of access paths. Access paths model heap locations as paths used to access them: a program variable followed by a finite sequence of field accesses (e.g., $foo.a.b$). We use access paths as we want to explicitly track states of class members. The abstract domain, denoted $\mathbb{D}$, maps access paths $\mathcal{AP}$ to BFA triplets:

$$\mathbb{D} : \mathcal{AP} \to \bigcup_{c \in \mathcal{C}} Cod(\mathcal{L}_c)$$

As variables denoted by an access path in $\mathcal{AP}$ can be of any declared class $c \in \mathcal{C}$, the co-domain of $\mathbb{D}$ is the union of codomains of $\mathcal{L}_c$ for all classes in a program. We remark that $\mathbb{D}$ is sufficient for both checking and summary computation, as we will show in the remaining of the section.

Definition 6 (Join Operator). *We define* $\bigsqcup : Cod(\mathcal{L}_c) \times Cod(\mathcal{L}_c) \rightarrow Cod(\mathcal{L}_c)$ *as follows:* $\langle E_1, D_1, P_1\rangle \sqcup \langle E_2, D_2, P_2\rangle = \langle E_1 \cap E_2 \setminus (D_1 \cup D_2),\ D_1 \cup D_2,\ P_1 \cup\ P_2\rangle$.

The join operator on $Cod(\mathcal{L}_c)$ is lifted to $\mathbb{D}$ by taking the union of un-matched entries in the mapping.

The compositional analysis is given in Alg. 1. It expects a program's CFG and a series of contracts, expressed as BFAs annotation mappings (Def. 3). If the program violates the BFA contracts, a warning is raised. For the sake of clarity we only return a boolean indicating if a contract is violated (cf. Def. 8). In the actual implementation we provide more elaborate error reporting. The algorithm traverses the CFG nodes top-down. For each node v, it first collects information from its predecessors (denoted by $\mathsf{pred}(v)$) and joins them as σ (line 3). Then, the algorithm checks whether a method can be called in the given abstract state σ by predicate $\mathsf{guard}()$ (cf. Alg. 2). If the pre-condition is met, then the $\mathsf{transfer}()$ function (cf. Alg. 3) is called on a node. We assume a collection of BFA contracts (given as $\mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k}$), which is input for Alg. 1, is accessible in Alg. 3 to avoid explicit passing. Now, we define some useful functions and predicates. For the algorithm, we require that the constructor disabling set is the complement of the enabling set:

Definition 7 ($well\_formed(\mathcal{L}_c)$)**.** *Let c be a class, Σ methods set of class c, and $\mathcal{L}_c$. Then,* $\mathsf{well\_formed}(\mathcal{L}_c) = \textbf{\textit{true}}$ *iff* $\mathcal{L}_c(constr) = \langle E, \Sigma \setminus E, P\rangle$.

Definition 8 ($warning(\cdot)$)**.** *Let G be a CFG and $\mathcal{L}_1, \ldots, \mathcal{L}_k$ be a collection of BFAs. We define* $\mathsf{warning}(G, \mathcal{L}_1, \ldots, \mathcal{L}_k) = \textbf{\textit{true}}$ *if there is a path in G that violates some of $\mathcal{L}_i$ for $i \in \{1, \ldots, k\}$.*

Definition 9 ($exit\_node(\cdot)$)**.** *Let v be a method call node. Then,* $\mathsf{exit\_node}(v)$ *denotes exit node w of a method body corresponding to v.*

Definition 10 ($actual\_arg(\cdot)$)**.** *Let $v = Call - node[m_j(p_0 : b_0, \ldots, p_n : b_n)]$ be a call node where $p_0, \ldots, p_n$ are formal and $b_0, \ldots, b_n$ are actual arguments and let $p \in \mathcal{AP}$. We define* $\mathsf{actual\_arg}(p, v) = b_i$ *if $p = p_i$ for $i \in \{0, \ldots, n\}$, otherwise* $\mathsf{actual\_arg}(p, v) = p$.

For convinience, we use *dot notation* to access elements of BFA triplets:

Definition 11 (Dot notation for BFA triplets). *Let $\sigma \in \mathbb{D}$ and $p \in \mathcal{AP}$. Further, let $\sigma[p] = \langle E_\sigma, D_\sigma, P_\sigma\rangle$. Then, we have $\sigma[p].E = E_\sigma$, $\sigma[p].D = D_\sigma$, and $\sigma[p].P = P_\sigma$.*

Guard Predicate Predicate $\mathsf{guard}(v, \sigma)$ checks whether a pre-condition for method call node v in the abstract state σ is met (cf. Alg. 2). We represent a call node as $m_j(p_0 : b_0, \ldots, p_n : b_n)$ where p_i are formal and b_i are actual arguments

Algorithm 1: BFA Compositional Analysis

```
  Data: G : A program's CFG, a collection of BFA mappings: $\mathcal{L}_{c_1}, \dots, \mathcal{L}_{c_k}$ over
        classes $c_1, \dots c_k$ such that $well\_formed(\mathcal{L}_{c_i})$ for $i \in \{1, \dots, k\}$
  Result: $warning(G, \mathcal{L}_{c_1}, \dots, \mathcal{L}_{c_k})$
1 Initialize $NodeMap : Node \to \mathbb{D}$ as an empty map;
2 foreach $v$ in forward(G)) do
3     $\sigma = \bigsqcup_{w \in pred(v)} w$;
4     if guard(v, σ) then $NodeMap[v] :=$ transfer($v$,$\sigma$); else return True;
5 return False
```

Algorithm 2: Guard Predicate

```
  Data: $v$ : CFG node, $\sigma$ : Domain
  Result: False iff $v$ is a method call that cannot be called in $\sigma$
1 Procedure guard $(v, \sigma)$
2     switch $v$ do
3         case Call-node[$m_j(p_0 : b_0, \dots, p_n : b_n)$] do
4             Let $w = exit\_node(v)$;
5             for $i \in \{0, \dots, n\}$ do
6                 if $\sigma_w[p_i].P \cap \sigma[b_i].D \neq \emptyset$ then return False;
7             return True
8         otherwise do
9             return True
```

(for $i \in \{0, \dots, n\}$). Let σ_w be a post-state of an exit node of method m_j. The pre-condition is met if for all b_i there are no elements in their pre-condition set (i.e., the third element of $\sigma_w[b_i]$) that are also in disabling set of the current abstract state $\sigma[b_i]$. For this predicate we need the property $D = \Sigma_{c_i} \setminus E$, where Σ_{c_i} is a set of methods for class c_i. This is ensured by condition $well\_formed(\mathcal{L}_{c_i})$ (Def. 7) and by definition of transfer() (see below).

Transfer Function The transfer function is given in Alg. 3. It distinguishes between two types of CFG nodes:

Entry-node: (lines 3–6) This is a function entry node. For simplicity we represent it as $m_j(p_0, \dots, p_n)$ where m_j is a method name and $p_0, \dots, p_n$ are formal arguments. We assume p_0 is a reference to the receiver object (i.e., *this*). If method m_j is defined in class c_i that has user-supplied annotations $\mathcal{L}_{c_i}$, in line 5 we initialize the domain to the singleton map (*this* mapped to $\mathcal{L}_{c_i}(m_j)$). Otherwise, we return an empty map meaning that a summary has to be computed.

Call-node: (lines 7–20) We represent a call node as $m_j(p_0 : b_0, \dots, p_n : b_n)$ where we assume actual arguments $b_0, \dots, b_n$ are access paths for objects and b_0 represents a receiver object. The analysis is skipped if *this* is in the domain (line 10): this means the method has user-entered annotations. Otherwise, we transfer an abstract state for each argument b_i, but also for each *class member* whose state is updated by m_j. Thus, we consider all access paths in the domain of σ_w, that is $ap \in dom(\sigma_w)$ (line 11). We construct access path ap' given ap. We distinguish two cases: ap denotes (i) a member and (ii) a formal argument of m_j. By line 12 we handle both cases as follows. In the former case we know ap has

Algorithm 3: Transfer Function

```
Data: $v$ : CFG node, $\sigma$ : Domain
Result: Output abstract state $\sigma'$ : $Domain$
Procedure transfer $(v, \sigma)$
    switch $v$ do
        case $Entry\text{-}node[m_j(p_0, \ldots, p_n)]$ do
            Let $c_i$ be the class of method $m_j(p_0, \ldots, p_n)$;
            if $There\ is\ \mathcal{L}_{c_i}$ then return $\{this \mapsto \mathcal{L}_{c_i}(m_j)\}$;
            else return $EmptyMap$ ;
        case $Call\text{-}node[m_j(p_0 : b_0, \ldots, p_n : b_n)]$ do
            Let $\sigma_w$ be an abstract state of $exit\_node(v)$;
            Initialize $\sigma' := \sigma$;
            if $\mathtt{this}\ not\ in\ \sigma'$ then
                for $ap\ in\ dom(\sigma_w)$ do
                    $ap' = actual\_arg(ap\{b_0/\mathtt{this}\}, v)$;
                    if $ap'\ in\ dom(\sigma)$ then
                        $E' = (\sigma[ap'].E \cup \sigma_w[ap].E) \setminus \sigma_w[ap].D$;
                        $D' = (\sigma[ap'].D \cup \sigma_w[ap].D) \setminus \sigma_w[ap].E$;
                        $P' = \sigma[ap'].P \cup (\sigma_w[ap].P \setminus \sigma[ap'].E)$;
                        $\sigma'[ap'] = \langle E', D', P' \rangle$;
                    else
                        $\sigma'[ap'] := \sigma_w[ap]$;
            return $\sigma'$
        otherwise do
            return $\sigma$
```

form $this.c_1.\ldots.c_n$. Thus, we construct ap' as ap with $this$ substituted for b_0 ($\mathsf{actual\_arg}(\cdot)$ is the identity in this case, see Def. 10): e.g., if receiver b_0 is $this.a$ and ap is $this.c_1.\ldots.c_n$ then $ap' = this.a.c_1.\ldots.c_n$. In the latter case ap denotes formal argument p_i and $\mathsf{actual\_arg}(\cdot)$ returns corresponding actual argument b_i. Now, as ap' is determined we construct its BFA triplet. If ap' is not in the domain of σ (line 13) we copy a corresponding BFA triplet from σ_w (line 19). Otherwise, we transfer elements of an BFA triplet at $\sigma[ap']$ as follows. The resulting enabling set is obtained by (i) adding methods that m_j enables ($\sigma_w[ap].E$) to the current enabling set $\sigma[ap'].E$, and (ii) removing methods that m_j disables ($\sigma_w[ap].D$), from it. The disabling set D' is constructed in a complementary way. Finally, the pre-condition set $\sigma[ap'].P$ is expanded with elements of $\sigma_w[ap].P$ that are not in the enabling set $\sigma[ap'].E$. We remark that the property $D = \Sigma_{c_i} \setminus E$ is preserved by the definition of E' and D'. Transfer is the identity on σ for all other types of CFG nodes. We can see that for each method call we have constant number of bit-vector operations per argument. That is, BFA analysis is insensitive to the number of states, as a set of states is abstracted as a single set.

Note, in our implementation we use several features specific to INFER: (1) INFER's summaries which allow us to use a single domain for intra and inter procedural analysis; (2) scheduling on CFG top-down traversal which simplify the handling of branch statements. In principle, BFA can be implemented in other frameworks e.g., IFDS [19].

Correctness In a BFA, we can abstract a set of states by the *intersection* of states in the set. That is, for $P \subseteq Q$ all method call sequences accepted by each state in P are also accepted by the state that is the intersection of bits of states in the set. Theorem 2 formalizes this property. First we need an auxiliary definition; let us write $Cod(\cdot)$ to denote the codomain of a mapping:

Definition 12 **($[\![\cdot]\!](\cdot)$).** *Let $\langle E, D, P\rangle \in Cod(\mathcal{L}_c)$ and $b \in \mathcal{B}^n$. We define $[\![\langle E, D, P\rangle]\!](b) = b'$ where $b' = (b \cup E) \setminus D$ if $P \subseteq b$, and is undefined otherwise.*

Theorem 2 **(BFA $\cap$-Property).** *Let $M = (Q, \Sigma_c, \delta, q_{10^{n-1}}, \mathcal{L}_c)$, $P \subseteq Q$, and $b_* = \bigcap_{q_b \in P} b$, then*

1. *For $m \in \Sigma_c$, it holds: $\delta(q_b, m)$ is defined for all $q_b \in P$ iff $\delta(q_{b_*}, m)$ is defined.*
2. *Let $\sigma = \mathcal{L}_c(m)$. If $P' = \{\delta(q_b, m) : q_b \in P\}$ then $\bigcap_{q_b \in P'} b = [\![\sigma]\!](b_*)$.*

Proof. By induction on cardinality of P and Def. 4. See [3] for details.

Our BFA-based algorithm (Alg. 1) interprets method call sequences in the abstract state and joins them (using join from Def. 6) following the control-flow of the program. Thus, we can prove its correctness by separately establishing: (1) the correctness of the interpretation of call sequences using a *declarative* representation of the transfer function (Def. 13) and (2) the soundness of join operator (Def. 6). For brevity, we consider a single program object, as method call sequences for distinct objects are analyzed independently. We define the *declarative* transfer function as follows:

Definition 13 **($\mathsf{dtransfer}_c(\cdot)$).** *Let $c \in \mathcal{C}$ be a class, Σ_c be a set of methods of c, and $\mathcal{L}_c$ be a BFA. Further, let $m \in \Sigma_c$ be a method, $\langle E^m, D^m, P^m\rangle = \mathcal{L}_c(m)$, and $\langle E, D, P\rangle \in Cod(\mathcal{L}_c)$. Then,*

$$\mathsf{dtransfer}_c(m, \langle E, D, P\rangle) = \langle E', D', P'\rangle$$

where $E' = (E \cup E^m) \setminus D^m$, $D' = (D \cup D^m) \setminus E^m$, and $P' = P \cup (P^m \setminus E)$, if $P^m \cap D = \emptyset$, and is undefined otherwise. Let $m_1, \ldots, m_n, m_{n+1}$ be a method sequence and $\phi = \langle E, D, P\rangle$, then

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \phi) = \mathsf{dtransfer}_c(m_{n+1}, \mathsf{dtransfer}_c(m_1, \ldots, m_n, \phi))$$

Relying on Thm. 2, we state the soundness of join:

Theorem 3 **(Soundness of $\sqcup$).** *Let $q_b \in Q$ and $\phi_i = \langle E_i, D_i, P_i\rangle$ for $i \in \{1, 2\}$. Then, $[\![\phi_1]\!](b) \cap [\![\phi_2]\!](b) = [\![\phi_1 \sqcup \phi_2]\!](b)$.*

Proof. By definitions Def. 6 and Def. 12, and set laws. See [3] for details.

With these auxiliary notions in place, we show the correctness of the transfer function (i.e., summary computation that is specialized for the code checking):

Theorem 4 **(Correctness of $\mathsf{dtransfer}_c(\cdot)$).** *Let $M = (Q, \Sigma, \delta, q_{10^{n-1}}, \mathcal{L}_c)$. Let $q_b \in Q$ and $m_1 \ldots m_n \in \Sigma^*$. Then*

$$\mathsf{dtransfer}_c(m_1 \ldots m_n, \langle \emptyset, \emptyset, \emptyset, \rangle) = \langle E', D', P'\rangle \iff \hat{\delta}(q_b, m_1 \ldots m_n) = q_{b'}$$

where $b' = [\![\langle E', D', P'\rangle]\!](b)$.

Proof. By induction on the length of the method call sequence. See [3] for details.

4 Evaluation

We evaluate our technique to validate the following two claims:

Claim-I: Smaller annotation overhead. The BFA contract annotation overheads are smaller in terms of atomic annotations (e.g., @Post(...), @Enable(...)) than both competing analyses.

Claim-II: Improved scalability on large code and contracts. Our analysis scales better than the competing analyzers for our use case on two dimensions, namely, caller code size and contract size.

Experimental Setup We used an Intel(R) Core(TM) i9-9880H CPU at 2.3 GHz with 16GB of physical RAM running macOS 11.6 on the bare-metal. The experiments were conducted in isolation without virtualization so that runtime results are robust. All experiments shown here are run in single-thread for Infer 1.1.0 running with OCaml 4.11.1.

We implement two analyses in Infer, namely BFA and DFA, and use the default Infer typestate analysis Topl as a baseline comparison. More in details: (1) BFA: The Infer implementation of the technique described in this paper. (2) DFA: A lightweight DFA-based typestate implementation based on an DFA-based analysis implemented in Infer. We translate BFA annotations to a minimal DFA and perform the analysis. (3) Topl: An industrial typestate analyzer, implemented in Infer [2]. This typestate analysis is designed for high precision and not for low-latency environments. It uses Pulse, an Infer memory safety analysis, which provides it with alias information. We include it in our evaluation as a baseline state-of-the-art typestate analysis, i.e., an off-the-shelf industrial strength tool we could hypothetically use. We note our benchmarks do not require aliasing and in theory Pulse is not required.

We analyze a benchmark of 18 contracts that specify common patterns of locally dependent contract annotations for a class. Moreover, we auto-generate 122 client programs parametrized by lines of code, number of composed classes, if-branches, and loops. Note, the code is such that it does not invoke the need for aliasing (as we do not support it yet in our BFA implementation). Client programs follow the compositional patterns we described in Ex. 2; which can also be found in [13]. The annotations for BFA are manually specified; from them, we generate minimal DFAs representations in DFA annotation format and Topl annotation format.

Our use case is to integrate static analyses in interactive IDEs e.g., Microsoft Visual Studio Code [21], so that code can be analyzed at coding time. For this reason, our use case requires low-latency execution of the static analysis. Our SLA is based on the RAIL user-centric performance model [1].

Usability Evaluation Fig. 4 outlines the key features of the 18 contracts we considered, called CR-1 – CR-18. In [3] we detail CR-4 as an example. For each contract, we specify the number of methods, the number of DFA states the contract corresponds to, and number of atomic annotation terms in BFA, DFA, and Topl. An atomic annotation term is a standalone annotation in the given annotation language. We can observe that as the contract sizes increase in number

Contract	#methods	#states	#BFA	#DFA	#TOPL
CR-1	3	2	3	5	9
CR-2	3	3	5	5	14
CR-3	3	5	4	8	25
CR-4	5	5	5	10	24
CR-5	5	9	8	29	71
CR-6	5	14	9	36	116
CR-7	7	18	12	85	213
CR-8	7	30	10	120	323
CR-9	7	41	12	157	460
CR-10	10	85	18	568	1407
CR-11	14	100	17	940	1884
CR-12	14	1044	32	7766	20704
CR-13	14	1628	21	13558	33740
CR-14	14	2322	21	15529	47068
CR-15	14	2644	24	26014	61846
CR-16	16	3138	29	38345	88134
CR-17	18	3638	23	39423	91120
CR-18	18	4000	27	41092	101185

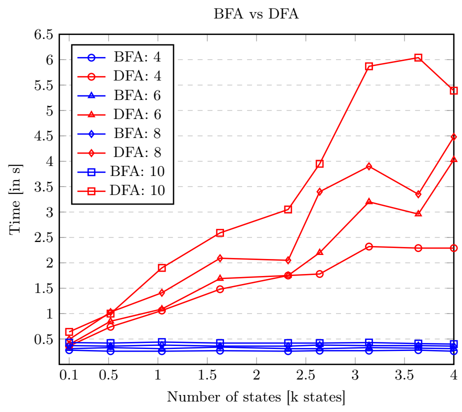
Fig. 4: Details of the 18 contracts in our evaluation.

of states, the annotation overhead for DFA and Topl increase significantly. On the other hand, the annotation overhead for BFA remain largely constant wrt. state increase and increases rather proportionally with the number of methods in a contract. Observe that for contracts on classes with 4 or more methods, a manual specification using DFA or Topl annotations becomes impractical. Overall, we validate Claim-I by the fact that BFA requires less annotation overhead on all of the contracts, making contract specification more practical.

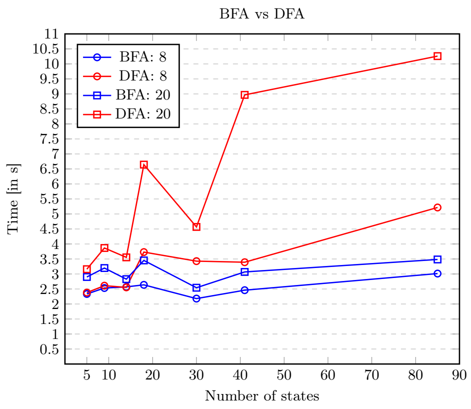
Performance Evaluation Recall that we distinguish between *base* and *composed* classes: the former have a user-entered contract, and the latter have contracts that are implicitly inferred based on those of their members (that could be either base or composed classes themselves). The total number of base classes in a composed class and contract size (i.e., the number of states in a minimal DFA that is a translation of a BFA contract) play the most significant roles in execution-time. In Fig. 5 we present a comparison of analyzer execution-times (y-axis) with contract size (x-axis), where each line in the graph represents a different number of base classes composed in a given class (given in legends).

Comparing BFA analysis against DFA analysis. **Fig. 5a** compares various class compositions (with contracts) specified in the legend, for client programs of 500-1K LoC. The DFA implementation sharply increases in execution-time as the number of states increases. The BFA implementation remains rather constant, always under the SLA of 1 seconds. Overall, BFA produces a geometric mean speedup over DFA of 5.52×. **Fig. 5b** compares various class compositions for client programs of 15K LoC. Both implementations fail to meet the SLA; however, the BFA is close and exhibits constant behaviour regardless of the number of states in the contract. The DFA implementation is rather erratic, tending to sharply increase in execution-time as the number of states increases. Overall, BFA produces a geometric mean speedup over DFA of 1.5×.

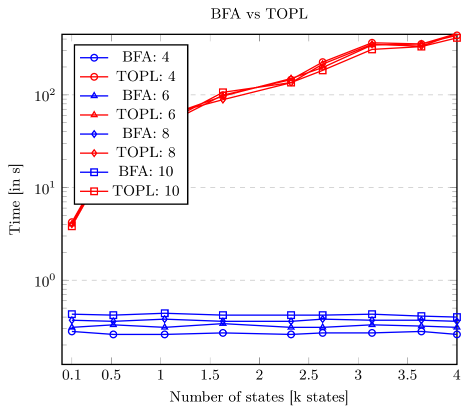
Comparing BFA-based analysis vs TOPL typestate implementations (Execution time). Here again client programs do not require aliasing. **Fig. 5c** compares various class compositions for client programs of 500-1K LoC. The Topl implementation sharply increases in execution-time as the number of states increases, quickly missing the SLA. In contrast, the BFA implementation remains constant always under the SLA. Overall, BFA produces a geometric mean speedup over Topl of 6.59×. **Fig. 5d** compares various class compositions for client programs of 15K LoC. Both implementations fail to meet the SLA. The Topl implementation remains constant until ∼30 states and then rapidly increases in execution time. Overall, BFA produces a geometric mean speedup over Topl of 287.86×.



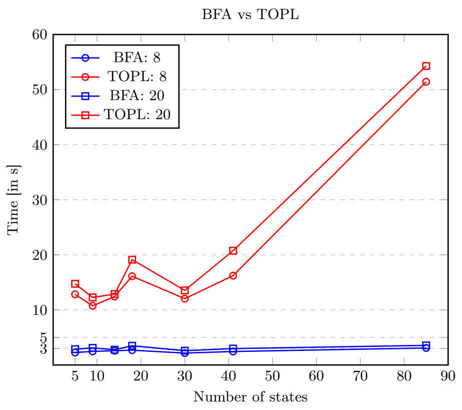
(a) DFA vs BFA execution comparison on composed contracts (500-1k LoC)



(b) DFA vs BFA execution comparison on composed contracts (15k LoC)



(c) Topl vs BFA comparison on composed contracts (500-1k LoC)



(d) Topl vs BFA comparison on composed contracts (15k LoC)

Fig. 5: Runtime comparisons. Each line represents a different number of base classes composed in a client code.

Overall, we validate Claim-II by showing that our technique removes state as a factor of performance degradation at the expense of limited but suffice contract expressively. Even when using client programs of 15K LoC, we remain close to our SLA and with potential to achieve it with further optimizations.

5 Related Work

We focus on comparisons with restricted forms of typestate contracts. We refer to the typestate literature [20,16,9,6,8] for a more general treatment. The work [14] proposes restricted form of typestates tailored for use-case of the object construction using the builder pattern. This approach is restricted in that it only accumulates called methods in an abstract (monotonic) state, and it does not require aliasing for supported contracts. Compared to our approach, we share the idea of specifying typestate without explicitly mentioning states. On the other hand, their technique is less expressive than our annotations. They cannot express various properties we can (e.g., the property "cannot call a method"). Similarly, [11] defines heap-monotonic typestates where monotonicity can be seen as a restriction. It can be performed without an alias analysis.

Recent work on the RAPID analyzer [10] aims to verify cloud-based APIs usage. It combines *local* type-state with global value-flow analysis. Locality of type-state checking in their work is related to aliasing, not to type-state specification as in our work. Their type-state approach is DFA-based. They also highlight the state explosion problem for usual contracts found in practice, where the set of methods has to be invoked prior to some event. In comparison, we allow more granular contract specifications with a very large number of states while avoiding an explicit DFA. The FUGUE tool [8] allows DFA-based specifications, but also annotations for describing specific *resource protocols* contracts. These annotations have a *locality* flavor—annotations on one method do not refer to other methods. Moreover, we share the idea of specifying typestate without explicitly mentioning states. These additional annotations in FUGUE are more expressive than DFA-based typestates (e.g. "must call a release method"). We conjecture that "must call" property can be encoded as bit-vectors in a complementary way to our BFA approach. We leave this extension for future work.

Our annotations could be mimicked by having a local DFA attached to each method. In this case, the DFAs would have the same restrictions as our annotation language. We are not aware of prior work in this direction. We also note that while our technique is implemented in INFER using the algorithm in §2, the fact that we can translate typestates to bit-vectors allows typestate analysis for local contracts to be used in distributive dataflow frameworks, such as IFDS [19], without the need for modifying the framework for non-distributive domains [17].

6 Concluding Remarks

In this paper, we have tackled the problem of analyzing code contracts in low-latency environments by developing a novel lightweight typestate analysis. Our technique is based on BFAs, a sub-class of contracts that can be encoded as bit-vectors. We believe BFAs are a simple and effective abstraction, with substantial potential to be ported to other settings in which DFAs are normally used.

Acknowledgements We are grateful to the anonymous reviewers for their constructive remarks. This work has been partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

References

1. RAIL model. `https://web.dev/rail/`, accessed: 2021-09-30
2. Infer TOPL. `https://fbinfer.com/docs/checker-topl/` (2021)
3. Arslanagić, A., Subotić, P., Pérez, J.A.: Scalable Typestate Analysis for Low-Latency Environments (Extended Version). CoRR **abs/2201.10627** (2022), `https://arxiv.org/abs/2201.10627`

4. Arslanagić, A., Subotić, P., Pérez, J.A.: Lfa checker: Scalable typestate analysis for low-latency environments (Mar 2022). https://doi.org/10.5281/zenodo.6393183
5. Bierhoff, K., Aldrich, J.: Modular Typestate Checking of Aliased Objects. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. p. 301–320. OOPSLA '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1297027.1297050, https://doi.org/10.1145/1297027.1297050
6. Bodden, E., Hendren, L.: The Clara framework for Hybrid Typestate Analysis. Int. J. Softw. Tools Technol. Transf. **14**(3), 307–326 (jun 2012)
7. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 459–465. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
8. Deline, R., Fähndrich, M.: The Fugue protocol checker: Is your software Baroque? Tech. Rep. MSR-TR-2004-07, Microsoft Research (04 2004)
9. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming. pp. 465–490. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
10. Emmi, M., Hadarean, L., Jhala, R., Pike, L., Rosner, N., Schäf, M., Sengupta, A., Visser, W.: RAPID: Checking API Usage for the Cloud in the Cloud. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1416–1426. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3473934, https://doi.org/10.1145/3468264.3473934
11. Fahndrich, M., Leino, R.: Heap Monotonic Typestate. In: Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO) (July 2003), https://www.microsoft.com/en-us/research/publication/heap-monotonic-typestate/
12. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software. pp. 10–30. FoVeOOS'10, Springer-Verlag, Berlin, Heidelberg (2010)
13. Jakobsen, M., Ravier, A., Dardha, O.: Papaya: Global Typestate Analysis of Aliased Objects. In: 23rd International Symposium on Principles and Practice of Declarative Programming. PPDP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3479394.3479414, https://doi.org/10.1145/3479394.3479414
14. Kellogg, M., Ran, M., Sridharan, M., Schäf, M., Ernst, M.D.: Verifying Object Construction. In: ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering. Seoul, Korea (May 2020)
15. Khedker, U., Sanyal, A., Sathe, B.: Data Flow Analysis: Theory and Practice. CRC Press (2017), https://books.google.rs/books?id=9PyrtgNBdg0C
16. Lam, P., Kuncak, V., Rinard, M.: Generalized Typestate Checking Using Set Interfaces and Pluggable Analyses. SIGPLAN Not. **39**(3), 46–55 (Mar 2004). https://doi.org/10.1145/981009.981016, https://doi.org/10.1145/981009.981016
17. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical Extensions to the IFDS algorithm. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction. pp. 124–144. CC'10/ETAPS'10, Springer-Verlag, Berlin, Heidelberg

(2010). https://doi.org/10.1007/978-3-642-11970-5_8, https://doi.org/10.1007/978-3-642-11970-5_8
18. Paul, R., Turzo, A.K., Bosu, A.: Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS project. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 1373–1385. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00124, https://doi.org/10.1109/ICSE43902.2021.00124
19. Reps, T., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61. POPL '95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199462, https://doi.org/10.1145/199448.199462
20. Strom, R.E., Yemini, S.: Typestate: A Programming Language Concept for Enhancing Software Reliability. IEEE Trans. Software Eng. **12**(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929, https://doi.org/10.1109/TSE.1986.6312929
21. Subotić, P., Milikić, L., Stojić, M.: A Static analysis Framework for Data Science Notebooks. In: ICSE'22: The 44th International Conference on Software Engineering (May 21-May 29 2022)
22. Szabó, T., Erdweg, S., Voelter, M.: IncA: A DSL for the Definition of Incremental Program Analyses. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 320–331. ASE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2970276.2970298, https://doi.org/10.1145/2970276.2970298
23. Yahav, E., Fink, S.: The SAFE Experience, pp. 17–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19823-6_3, https://doi.org/10.1007/978-3-642-19823-6_3