Bit-Vector Typestate Analysis

Arslanagic, Alen; Subotić, Pavle; Pérez, Jorge A.

Published in:
Formal Aspects of Computing

DOI:
10.1145/3595299

Bit-Vector Typestate Analysis

ALEN ARSLANAGIĆ, University of Groningen, The Netherlands
PAVLE SUBOTIĆ, Microsoft, Serbia
JORGE A. PÉREZ, University of Groningen, The Netherlands

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* compositional 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 whose analysis uses efficient bit-vector operations.

CCS Concepts: • **Software and its engineering → Formal software verification;**

Additional Key Words and Phrases: Static analysis, code contracts, typestates

ACM Reference format:
Alen Arslanagić, Pavle Subotić, and Jorge A. Pérez. 2023. Bit-Vector Typestate Analysis. *Form. Asp. Comput.* 35, 3, Article 19 (September 2023), 36 pages.
https://doi.org/10.1145/3595299

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 [20]. 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 [13] 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* [23]. 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.

This work has been partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).
Authors' addresses: A. Arslanagić and J. A. Pérez, University of Groningen, The Netherlands, Groningen, Nijenborgh 9, 9747 AG; emails: alen.arslanagic@gmail.com, j.a.perez@rug.nl; P. Subotić, Microsoft, Serbia, Belgrade, Španskih Boraca 3, 11070; email: psubotic@gmail.com.

0934-5043/2023/09-ART19 $15.00
https://doi.org/10.1145/3595299

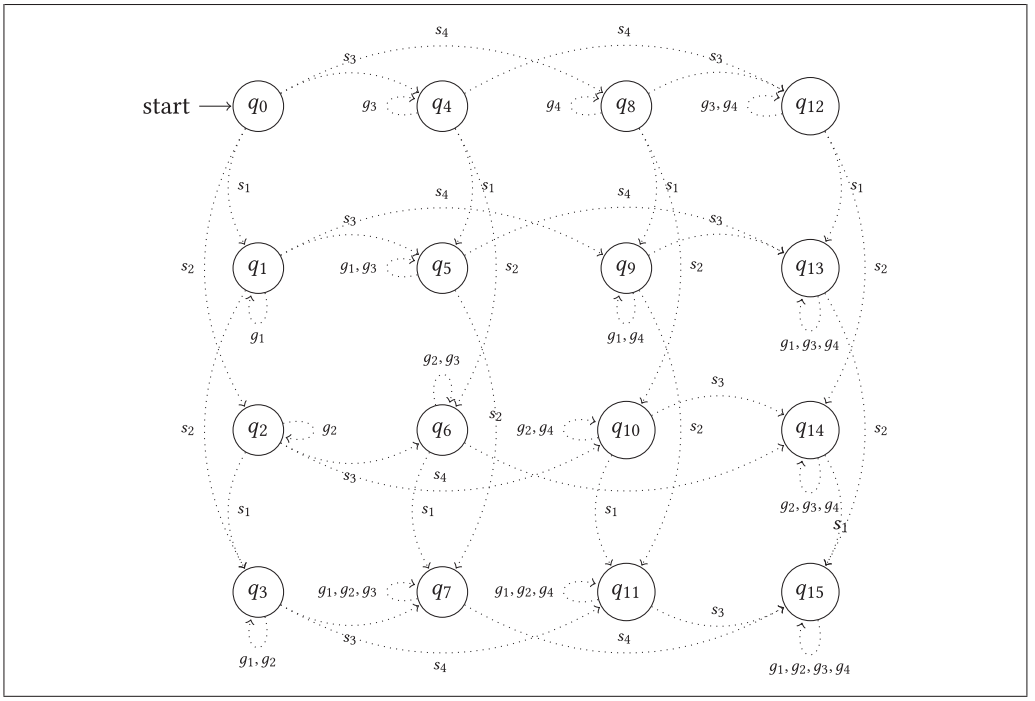


Fig. 1. State diagram of a DFA-based setter/getter contract (case $n = 4$, with 16 states and 64 transitions).

Given this, it is natural for static typestate checkers, such as Fugue [10], SAFE [26], and Infer's Topl checker [1], 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**) [24, 25]. In this context, to avoid noticeable disruptions in the users' workflow, the analysis should ideally run *under a second* [2]. However, relying on DFAs jeopardizes this goal, as it can lead to scalability issues.

To illustrate these limitations, consider the representative example of a class with four setter/getter method pairs, where each setter method *enables* a corresponding getter method and then *disables* itself; the intention is that values can be set up once and accessed multiple times. The associated DFA contract has 2^4 states, as any subset of getter methods can be available at a particular program point, depending on previous calls (cf. Figure 1). Additionally, the full DFA-based specification requires as many as 64 state transitions. To see this, each state has 4 transitions available, with complementary enabled setter and getter methods; this way, e.g., state q_3 has outgoing transitions with labels g_1, g_2, s_3, s_4 and state q_7 has outgoing transitions with labels g_1, g_2, g_3, s_4. In the general case (n methods), a DFA for this kind of contract can have 2^n states. Even with a small n, as in Figure 1, such contracts are impractical to codify manually and are likely to result in sub-par performance.

This kind of enable/disable properties are referred to as *may call properties*. Interestingly, the specification of common "must call" properties can also result in prohibitively large DFA

state-space. As an example, consider a class that has m pairs of methods for acquiring/releasing some resources. The contract should ensure that all acquired resources are released before the object is destructed. Because states would need to track unreleased resources, a DFA for this contract requires 2^m states.

Any DFA-based typestate analysis crucially depends on the number of states. Typically the analysis has a finite-state domain and a distributive transfer function; it falls into a category of so-called *distributive analysis* that admits precise interprocedural (compositional) analysis in polynomial time (see IFDS [21]). The number of states is critical: in the worst case, the analysis takes $|Q|^3$ operations per method invocation, where Q is the set of states of the underlying DFA. To see why this is the case, we may notice that a procedure can be invoked in any state—thus, we need to analyze a function with every state as a potential entry state. Furthermore, this per-state analysis must deal with subsets of states. Thus, contracts that induce a large state space can severely impact the performance of the compositional analysis.

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—an enabling/disabling relation concerns a pair of methods. In contrast, DFA-based approaches have by definition a *global standpoint*; 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.

Our Solution. Based on these observations, 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, and so on, 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. We establish the exact difference between DFAs and BFAs: a *context-independence* property, satisfied by the latter but not by the former. 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, we are able to abstract BFA states and transitions in such a way that our compositional analysis requires a *constant* number of bit-vector operations per method invocation. This makes our analysis *insensitive* to the number of states, which in turn ensures scalability with contract and program size.

Importantly, code contracts that are locally dependent allow efficient reasoning about contract subtyping, as required by class inheritance. Relying on DFAs can make reasoning and specifying contract subtyping a difficult task. Suppose c_2 is a sub-class of c_1 (i.e., c_1 is the super-class of c_2). Intuitively, a contract for c_2 must be *at least as permissive* as a contract for c_1. That is, a set of allowed sequences of method invocations for c_2 must subsume that of c_1. Locally-dependent contracts enable succinct specifications, which in turn enable an efficient subsumption checking algorithm, thereby making reasoning about subtyping an easy task. Indeed, by relying on our annotation language, we can check the subtyping relation simply by comparing annotations of the corresponding methods of super- and sub-classes; because this comparison operation is usual set inclusion, subtyping checking is insensitive to the number of states in a corresponding DFA.

We have implemented our lightweight typestate analysis in the industrial-strength static analyzer Infer [8]. 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 slowdowns due to state increase.

Contributions and Organization. We summarize our contributions as follows:

- A specification language for typestates based on *lightweight annotations*. Our language rests upon BFAs, a new sub-class of DFA based on bit-vectors (Section 2).
- A lightweight analysis technique for code contracts, implemented in Infer (Section 3). An associated artifact is publicly available [4].
- The specification language in Section 2 and the analysis technique in Section 3 concern "may call" properties, which involve methods that may be called at some program point. In Section 4, we extend our approach to consider also *"must call"* properties, which are useful to express that a method *requires* another one to be invoked in a code continuation.
- Extensive evaluations for our lightweight analysis technique, which demonstrate considerable gains in performance and usability (Section 5).

We review related work in Section 6 and collect some closing remarks in Section 7.

This article is an extended and revised version of our conference paper [3]. In this presentation, we consider a more general formalism of BFAs, which incorporates "must call" properties. Moreover, this article includes formal proofs for the extended formalism in Section 4 and an updated experimental evaluation in Section 5.

2 BIT-VECTOR TYPESTATE ANALYSIS

2.1 Annotation Language

We introduce BFA specifications, which succinctly encode temporal properties by describing *local* method dependencies, thus avoiding the need for a full 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.

Notation 2.1. We define some base sets and notations.

- We write *Classes* to denote the finite set of all classes under consideration. We use $c, c', \ldots$ to denote elements of *Classes*.
- The set $\Sigma_c = \{m^{\uparrow}, m_1, \ldots, m_n, m^{\downarrow}\}$ denotes the n methods of a class c. In Σ_c, both $m^{\uparrow}$ and $m^{\downarrow}$ are notations reserved for the constructor and destructor methods of the class, respectively. We assume a single constructor and destructor for simplicity and clarity; our formalism can be extended to support multiple constructors without difficulties.
- The set $\Sigma_c^{\bullet}$ is defined as $\Sigma_c \setminus \{m^{\uparrow}, m^{\downarrow}\}$. For convenience, we will assume a total ordering on $\Sigma_c^{\bullet}$; this will be useful when defining BFAs in the next section.

We will often use E and D to denote subsets of $\Sigma_c^{\bullet}$. Also, we shall write $\tilde{x}$ to denote finite sequences of elements $x_1, \ldots, x_k$ (with $k > 0$).

Definition. Following the above intuitions on "`@Enable`(n) m" and "`@Disable`(n) m", we define BFA annotations per method and a corresponding notion of valid method sequences:

Definition 2.1 (Annotation Language). Let $c \in \mathit{Classes}$ such that $\Sigma_c = \{m^{\uparrow}, m_1, \ldots, m_n, m^{\downarrow}\}$. We have:

— The constructor method $m^{\uparrow}$ is annotated by

$$\texttt{@Enable}(E^c)\ \texttt{@Disable}(D^c)\ m^{\uparrow}$$

where $E^c \cup D^c = \Sigma_c^{\bullet}$ and $E^c \cap D^c = \emptyset$;

— Each $m_i \in \Sigma_c^{\bullet}$ is annotated by

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

where $E_i \subseteq \Sigma_c^{\bullet}$, $D_i \subseteq \Sigma_c^{\bullet}$, and $E_i \cap D_i = \emptyset$.

Let $\tilde{x} = m^{\uparrow}, x_1, x_2, \ldots$ be a sequence where each $x_i \in \Sigma_c^{\bullet}$. We say that $\tilde{x}$ is *valid (w.r.t. annotations)* if for all subsequences $\tilde{x}' = x_i, \ldots, x_k$ of $\tilde{x}$ such that $x_k \in D_i$ there is j $(i < j \leq k)$ such that $x_k \in E_j$.

The formal semantics for these specifications is given in Section 2.2. We note that if E_i or D_i is $\emptyset$ then we omit the corresponding annotation.

Derived Annotations. The annotation language can be used to derive other useful annotations:

$$\texttt{@EnableOnly}(E_i)\ m_i \stackrel{\text{def}}{=} \texttt{@Enable}(E_i)\ \texttt{@Disable}(\Sigma_c^{\bullet} \setminus E_i)\ m_i$$

$$\texttt{@DisableOnly}(D_i)\ m_i \stackrel{\text{def}}{=} \texttt{@Disable}(D_i)\ \texttt{@Enable}(\Sigma_c^{\bullet} \setminus D_i)\ m_i$$

$$\texttt{@EnableAll}\ m_i \stackrel{\text{def}}{=} \texttt{@Enable}(\Sigma_c^{\bullet})\ m_i$$

This way, the annotation "`@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 $\Sigma_c^{\bullet}$. The annotation "`@DisableOnly`(D_i) m_i" is defined dually. Finally, the annotation "`@EnableAll` m_i" asserts that a call to method m_i enables all methods in a class; an annotation "`@DisableAll` m_i" can be defined similarly.

Examples. We illustrate the expressivity and usability of BFA annotations by means of examples. First, the complete setter/getter contract from Figure 1 can be specified with *only four BFA annotations*, namely:

$$\texttt{@Enable}(g_1)\ \texttt{@Disable}(s_1)\ s_1$$
$$\texttt{@Enable}(g_2)\ \texttt{@Disable}(s_2)\ s_2$$
$$\texttt{@Enable}(g_3)\ \texttt{@Disable}(s_3)\ s_3$$
$$\texttt{@Enable}(g_4)\ \texttt{@Disable}(s_4)\ s_4$$

Next, we consider the `SparseLU` class from `Eigen C++` library.[1] This class implements a **lower-upper (LU)** decomposition of a sparse matrix. We illustrate the expressivity and usability of BFA annotations by means of the following example. For brevity, we consider representative methods for a typestate specification (we also omit return types):

```
1 class SparseLU {
2     void analyzePattern(Mat a);
3     void factorize(Mat a);
4     void compute(Mat a);
5     void solve(Mat b);  }
```

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

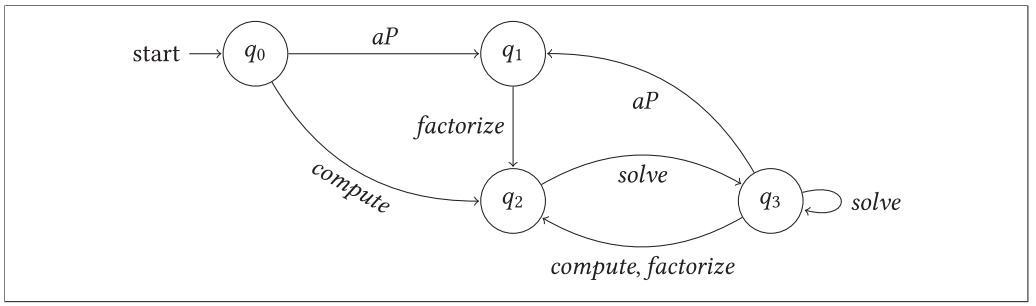


Fig. 2. DFA for the class SparseLU.

```
1  class SparseLU {
2      states q0, q1, q2, q3;
3      @Pre(q0) @Post(q1)
4      @Pre(q3) @Post(q1)
5      void analyzePattern(Mat a);
6      @Pre(q1) @Post(q2)
7      @Pre(q3) @Post(q2)
8      void factorize(Mat a);
9      @Pre(q0) @Post(q2)
10     @Pre(q3) @Post(q2)
11     void compute(Mat a);
12     @Pre(q2) @Post(q3)
13     @Pre(q3)
14     void solve(Mat b);  }
```

Listing 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 2. SparseLU BFA Contract.

Eigen's implementation of the class SparseLU 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.

Some method call sequences do not cause errors but have redundancies. For example, we can disallow consecutive calls to compute in sequences such as, e.g.,

"compute().compute().solve()"

as the result of the first call to compute is never used. Also, because compute is essentially implemented as "analyzePattern().factorize()", it is also redundant to call factorize after compute.

Figure 2 gives the corresponding DFA that substitutes dynamic checks and avoids redundancies. (In the figure, and in the following, we write *aP* to denote/abbreviate "analyzePattern".) Following the literature [10], this DFA can be annotated inside the definition of the class SparseLU as in Listing 1: States are listed in the class header and transitions are specified by *@Pre* and *@Post* conditions on methods. Already in this small example, this DFA specification is too low-level and presents high annotation overheads, which makes it unreasonable for software engineers to annotate their APIs.

The entire contract for the SparseLU class can be succinctly specified using BFA annotations as in Listing 2. In this case, the starting state is unspecified, as it is determined by annotations. In fact, methods that are not *guarded* by other methods (such as solve is guarded by compute), or

have weaker guards, are enabled in the starting state. We assume that `@EnableOnly` is a stronger guard than `@EnableAll`. Thus, here we infer that `analyzePattern()` and `compute()` are the only methods enabled upon object creation. This condition can be overloaded by specifying annotations on the constructor method. Remarkably, the contract can be specified with only four annotations; in contrast, the corresponding DFA requires eight annotations plus four states specified in the class header.

Another difference concerns the treatment of local method dependencies: a small change in BFA annotations can result in a substantial change of the corresponding DFA. To see this, let $\{m_1, m_2, m_3, \ldots, m_n\}$ be methods of some class with an associated DFA (with set of states Q), in which m_1 and m_2 are enabled in each state of Q. Adding an annotation such as "`@Enable(m2) m1`" doubles the number of states of the required 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 BFA (BFA, in the following): a class of DFAs that captures enabling/disabling dependencies between the methods of a class (cf. Definition 2.1) leveraging a bit-vector abstraction on typestates.

Definition 2.2 (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 ith 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 $\cup, \cap$, and $\setminus$).

We first define a BFA per class. We assume $c \in \mathit{Classes}$ and $\Sigma_c^\bullet = \{m_1, \ldots, m_n\}$ be as described in Notation 2.1. Given that c has n methods, we consider states q_b, where, following Definition 2.2, the bit-vector $b_A \in \mathcal{B}^n$ denotes the set of methods $A \subseteq \Sigma_c^\bullet$ enabled at that point. We assume that the bit-vector representation of the subset A is consistent with respect to the total ordering on $\Sigma_c^\bullet$, in the sense that bit $b[i]$ corresponds to $m_i \in \Sigma_c^\bullet$. 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 ith method is enabled (resp. disabled) in q_b.

Definition 2.3, given next, gives a mapping from methods to triples of bit-vectors, denoted $\mathcal{L}_c$. 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 is determined by E^c, the set of enabling annotations on the constructor.

Definition 2.3 (Mapping $\mathcal{L}_c$). Given a class c, we define $\mathcal{L}_c$ as a mapping from methods to triples of subsets of $\Sigma_c^\bullet$ as follows:

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

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 $\mathcal{L}_c(m_i)$. Similarly, we write E^c, D^c, and P^c to denote the elements of the triple $\mathcal{L}_c(m^\uparrow)$. 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 $\{m_i\}$. This singleton formulation is convenient to define the domain of the compositional analysis in Section 3.2: as we will see later, it allows us to uniformly treat method calls and procedure calls which can have more elements in pre-set P_i.

We impose some natural well-formedness conditions on the BFA mapping.

Definition 2.4 (well_formed($\mathcal{L}_c$)). Let c, Σ_c, and $\mathcal{L}_c$ be a class, its method set, and its BFA mapping, respectively. Then, well_formed($\mathcal{L}_c$) = **true** iff the following conditions hold:

— $\mathcal{L}_c(m^{\uparrow}) = \langle E^c, D^c, \emptyset \rangle$ such that $E^c \cup D^c = \Sigma_c^\bullet$ and $E^c \cap D^c = \emptyset$;
— for $m_i \in \Sigma_c^\bullet$ we have $\mathcal{L}_c(m_i) = \langle E_i, D_i, \{m_i\} \rangle$ such that $E_i, D_i \subseteq \Sigma_c^\bullet$ and $E_i \cap D_i = \emptyset$.

The first condition says that the constructor's enabling and disabling sets must be disjunctive and complementary with respect to $\Sigma_c^\bullet$; this will be convenient later when defining the compositional analysis algorithm in Section 3. The second condition ensures that every method's enabling and disabling sets are disjunctive. Furthermore, by taking $E_i, D_i \in \Sigma_c^\bullet$ we ensure that the annotations of method m_i cannot refer to the constructor nor the destructor (see Notation 2.1).

In a BFA, transitions between states $(q_b, q_{b'}, \ldots)$ are determined by $\mathcal{L}_c$. Given $m_i \in \Sigma_c$, we have $j \in E_i$ if and only if m_i enables m_j; similarly, we have $k \in D_i$ 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 serve to illustrate our approach and, in particular, the local nature of enabling/disabling dependencies between methods. The following definition makes them precise.

Definition 2.5 (BFA). Given a $c \in \mathit{Classes}$ with $n > 0$ methods, a BFA for c is defined as a tuple $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c}, \mathcal{L}_c)$ where:

— Q is a finite set of states $q_b, q_{b'}, \ldots$, where $b, b', \ldots \in \mathcal{B}^n$;
— $\Sigma_c^\bullet = \{m_1, \ldots, m_n\}$ is the alphabet (method identities);
— q_{E^c} is the starting state (recall that E^c is enabling set of a constructor);
— $\mathcal{L}_c$ is a BFA mapping (cf. Definition 2.3).
— $\delta : Q \times \Sigma_c \to Q$ 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.

Example 2.6 (SparseLU). We give the BFA derived from the annotations in the SparseLU example (Listing 2). We associate indices to methods:

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

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

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

The starting state is q_{1100}, as given by the annotations on the constructor. The set of states is

$$Q = \{q_{1100}, q_{0010}, q_{0001}, q_{1111}\}$$

Finally, the transition function δ is given by following eight transitions:

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

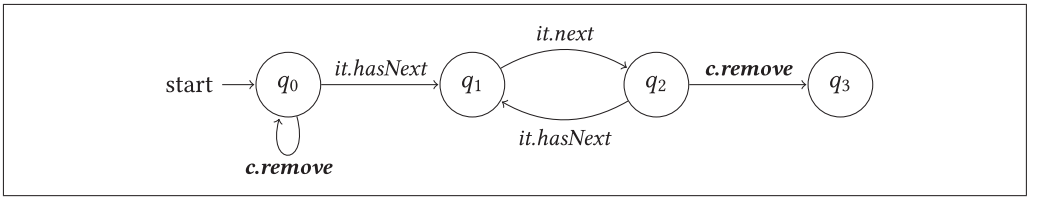


Fig. 3. State diagram of the DFA of an iterator.

Contrasting BFAs and DFAs. We have already seen the differences between BFAs and DFAs in the specification of a representative concrete example. We now compare BFAs and DFAs more formally, by identifying a property that distinguishes the two models.

The property, called *context-independence*, is satisfied by all BFAs but not by all DFAs. To state the property and prove this claim, we need some convenient notations. First, we use $\widetilde{m}$ to denote a finite sequence of method names in Σ. Also, we use "$\cdot$" to denote sequence concatenation, defined as expected. Furthermore, given a BFA M, we write $L(M)$ to denote the language accepted by M, defined as $\{\widetilde{m} : \hat{\delta}(q_{E^c}, \widetilde{m}) = q' \wedge q' \in Q\}$, where $\hat{\delta}(q_b, \widetilde{m})$ is the extension of the one-step transition function $\delta(q_b, m_i)$ to a sequence $\widetilde{m}$ of method calls.

BFAs determine a strict sub-class of DFAs. First, because all states in Q are accepting, BFA cannot encode *must call* properties (cf. Section 6). Next, we have the context-independence property:

Theorem 2.1 (Context-independence). *Let $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c}, \mathcal{L}_c)$ be a BFA. Then, for $m_n \in \Sigma_c^\bullet$ we have*

(1) If there is $\widetilde{p} \in L(M)$ and $m_{n+1} \in \Sigma_c^\bullet$ such that $\widetilde{p} \cdot m_{n+1} \notin L(M)$ and $\widetilde{p} \cdot m_n \cdot m_{n+1} \in L(M)$ then there is no $\widetilde{m} \in L(M)$ such that $\widetilde{m} \cdot m_n \cdot m_{n+1} \notin L(M)$.
(2) If there is $\widetilde{p} \in L(M)$ and $m_{n+1} \in \Sigma_c^\bullet$ such that $\widetilde{p} \cdot m_{n+1} \in L(M)$ and $\widetilde{p} \cdot m_n \cdot m_{n+1} \notin L(M)$ then there is no $\widetilde{m} \in L(M)$ such that $\widetilde{m} \cdot m_n \cdot m_{n+1} \in L(M)$.

Proof. We only consider the first item, as the second item is shown similarly. By $\widetilde{p} \cdot m_{n+1} \notin L(M)$ and $\widetilde{p} \cdot m_n \cdot m_{n+1} \in L(M)$ and Definition 2.5 we know that

$$m_{n+1} \in E_n \tag{1}$$

Furthermore, for any $\widetilde{m} \in (\Sigma_c^\bullet)^*$, let q_b be such that $\delta(q_{10^{n-1}}, \widetilde{m}) = q_b$ and $q_{b'}$ such that $\delta(q_b, m_n) = q_{b'}$. Now, by Definition 2.5 we have that $\delta(q_{b'}, m_{n+1})$ is defined, as by (1) we know $P_{n+1} = \{m_{n+1}\} \subseteq b'$. Thus, for all $\widetilde{m} \in L(M)$ we have $\widetilde{m} \cdot m_n \cdot m_{n+1} \in L(M)$. This concludes the proof. □

Informally, the above theorem says that the effect of a call to m_n to subsequent calls (m_{n+1}) is not influenced by previous calls (i.e., the context) $\widetilde{m}$. That is, Item 1. (resp. Item 2.) says that method m_n enables (resp. disables) the same set of methods in any context.

The context-independence property is not satisfied by all DFAs. Consider, for example, a DFA that disallows modifying a collection while iterating is not a BFA (as in [6, Figure 3]). Let *it* be a Java Iterator with its usual methods for a collection *c*. For the sake of illustration, we assume a single DFA relates the iterator and its collection methods; we give the associated state diagram in Figure 3. Then, the sequence

"`c.remove();it.hasNext()`"

```
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 3. Class Foo using SparseLU.

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

Listing 4. Client code for Foo.

should be allowed, whereas

"`it.hasNext();it.next();c.remove();it.hasNext()`"

should not. That is, "`c.remove`" disables "`it.hasNext`" *only if* "`it.hasNext`" has been previously called. Thus, the effect of calling "`c.remove`" depends on the calls that precede it.

BFA subtyping. The combination of (i) locally-dependent annotations and (ii) the context-independence property they satisfy enable us to check for contract subtyping by *independently* comparing annotations method-wise; importantly, this comparison boils down to usual set inclusion. Suppose M_1 and M_2 are BFAs for classes c_1 and c_2, respectively, with c_1 being the super-class of c_2. The class inheritance imposes the question: how to check that c_2 is a proper refinement of c_1? In other words, c_2 must subsume c_1: any valid sequence of calls to methods of c_1 must also be valid for c_2. 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 $m_j \in \mathcal{L}_c$ we have $E_1 \subseteq E_2$, $D_1 \supseteq D_2$, and $P_2 \subseteq P_1$ where $\langle E_i, D_i, P_i \rangle = \mathcal{L}_{c_i}(m_j)$ for $i \in \{1, 2\}$.

3 A COMPOSITIONAL ANALYSIS ALGORITHM

Since BFAs can be encoded as bit-vectors, standard data-flow analysis frameworks can be employed for the non-compositional case (e.g., intra-procedural analyses) [16]. Here, we address the case of member object methods being called: we present a compositional algorithm that is tailored to the Infer compositional static analysis framework.

3.1 Key Ideas

We motivate our compositional analysis technique with the example below.

Example 3.1. Let `Foo` be a class that has a member `lu` of class `SparseLU` (cf. Listing 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 on the *typestate* of an argument when returning from the procedure. A simple instance of a client is `wrongUseFoo` in Listing 4.

The central idea of our analysis is to accumulate enabling and disabling annotations. For this, the abstract domain maps object access paths to triples from the definition of $\mathcal{L}_{\text{SparseLU}}$ (cf. Definition 2.3). A *transfer function* interprets method calls in this abstract state. We illustrate the transfer function; the evolution of the abstract state is presented 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 triple 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 in this case). 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 (i.e., s_2 and s_3) at line 7. Intuitively, this 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. Definition 3.2). Accordingly, in line 8, we obtain the final state s_4, which is also a summary for the method `SetupLU1`.

Now, we illustrate the checking of the client code `wrongUseFoo()` (cf. Listing 4), with computed summaries:

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

Above, at line 2, the abstract state is initialized with annotations of the constructor `Foo`. Upon 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, in line 6, we can see that `aP` is in the pre-condition set in the summary for `setupLU2()` (sum_2), which is computed similarly as sum_1; however, it is 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, a class `Bar` can use two solvers, `SparseLU` and `SparseQR`:

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

where the class `SparseQR` has its own BFA contract. The implicit contract of `Bar` depends on the contracts of `lu` and `qr`. Moreover, a class such 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*.

3.2 The 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, which enable a field-sensitive data-flow analysis; see, e.g., [5, 18, 22] for more information on this subject. 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 would like to explicitly track states of class members; this, in turn, enables a precise compositional analysis. The abstract domain, denoted $\mathbb{D}$, maps access paths $\mathcal{AP}$ to BFA triples; below we write $Cod(\text{-})$ to denote the codomain of a mapping:

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

As the variables denoted by an access path in $\mathcal{AP}$ can be of any declared class $c \in Classes$, 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 computing summaries, as we will show in the remainder of the section.

Definition 3.2 (Join Operator). We define $\sqcup : Cod(\mathcal{L}_c) \times Cod(\mathcal{L}_c) \rightarrow Cod(\mathcal{L}_c)$ 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.

We now define some useful functions and predicates. First, we remark that our analysis is only concerned with three types of CFG nodes: a method call node, entry, and exit node of a method body; all other node types are irrelevant.

Notation 3.1. We introduce convenient notations for entry and method call nodes:

- `Entry-node`$[m_j(p_0, \ldots, p_n)]$ denotes a method entry node where m_j is a method name and $p_0, \ldots, p_n$ are formal arguments;
- `Call-node`$[m_j(p_0 : b_0, \ldots, p_n : b_n)]$ denotes a call to method m_j where $p_0, \ldots, p_n$ are formal arguments and $b_0, \ldots, b_n$ are actual arguments.

The following definitions concern CFG traversal, predecessor nodes, exit nodes, and actual parameters:

Definition 3.3 (forward(-)). Let G be a CFG. Then, $forward(G)$ enumerates nodes of G by traversing it in a breadth-first manner.

Definition 3.4 (pred(-)). Let G be a CFG and v a node of G. Then, $pred(v)$ denotes a set of predecessor nodes of v. That is, $pred(v) = W$ such that $w \in W$ if and only if there is an edge from w to v in G.

Definition 3.5 (warning(-)). Let G be a CFG and $\mathcal{L}_1, \ldots, \mathcal{L}_k$ be a collection of BFAs. We define $warning(G, \mathcal{L}_1, \ldots, \mathcal{L}_k) =$ **true** if there is a path in G that violates some of $\mathcal{L}_i$ for $i \in \{1, \ldots, k\}$.

Definition 3.6 (exit_node(-)). Let v be a method call node. Then, $exit\_node(v)$ denotes the exit node w of a method body corresponding to v.

Definition 3.7 (actual_arg(-, -)). Let $v =$ `Call-node`$[m_j(p_0 : b_0, \ldots, p_n : b_n)]$ be a call node. Suppose $p \in \mathcal{AP}$. We define $actual\_arg(p, v) = b_i$ if $p = p_i$ for $i \in \{0, \ldots, n\}$; otherwise $actual\_arg(p, v) = p$.

ALGORITHM 1: BFA Compositional Analysis

Data: G : A program's CFG, a collection of BFA mappings: $\mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k}$ over classes $c_1, \ldots c_k$ such that $well\_formed(\mathcal{L}_{c_i})$ for $i \in \{1, \ldots, k\}$
Result: $warning(G, \mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k})$

```
1 Initialize NodeMap : Node → 𝔻 as an empty map;
2 foreach v in forward(G) do
3     if pred(v) is empty then
4         Initialize σ : 𝔻 as an empty mapping;
5     else
6         σ = ⊔_{w∈pred(v)} w;
7     if guard(v, σ) then NodeMap[v] := transfer(v,σ); else return True;
8 return False
```

ALGORITHM 2: Guard Predicate

Data: v : CFG node, $\sigma : \mathbb{D}$
Result: False iff v is a method call that cannot be called in σ

```
1 Procedure guard (v, σ)
2     switch v do
3         case Call-node[m_j(p_0 : b_0, ..., p_n : b_n)] do
4             Let w = exit_node(v);
5             for i ∈ {0, ..., n} do
6                 if σ_w[p_i].P ∩ σ[b_i].D ≠ ∅ then return False;
7             return True
8         otherwise do
9             return True
```

For convenience, we use a *dot notation* to access elements of BFA triples:

Definition 3.8 (Dot Notation for BFA Triples). 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 $\sigma[p].P = P_\sigma$.

The compositional analysis is given in Algorithm 1. It expects a program's CFG and a series of contracts, expressed as BFAs annotation mappings (Definition 2.3). 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. Definition 3.5). In the actual implementation, we provide more elaborate error reporting.

The algorithm traverses the CFG nodes top-down in a for-loop (lines 2–7) as given by $forward(G)$ (cf. Definition 3.3). For each node v, we first check whether v has predecessors: if not, when $pred(v) = \emptyset$, we initialize domain σ as an empty mapping of type $\mathbb{D}$; otherwise, we collect information from its predecessors (as given by $pred(v)$) and join them as σ (line 6). Then, it uses predicate guard(-,-) (cf. Algorithm 2) to check whether a method can be called in the given abstract state σ. If the pre-condition is met, then the function transfer(-, -) (cf. Algorithm 3) is called on a node. We assume a collection of BFA contracts (given as $\mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k}$, the input for Algorithm 1) is accessible in Algorithm 3 to avoid explicit passing.

Guard Predicate. Predicate guard(v, σ) checks whether a pre-condition for method call node v in the abstract state σ is met (cf. Algorithm 2). We represent a call node as $m_j(p_0 : b_0, \ldots, p_n : b_n)$ where p_i and b_i (for $i \in \{0, \ldots, n\}$) are formal and actual arguments, respectively. Let σ_w be a post-state of an exit node of method m_j. A pre-condition is satisfied if for all b_i there are no elements in

ALGORITHM 3: Transfer Function

Data: v : CFG node, $\sigma : \mathbb{D}$
Result: Output abstract state $\sigma' : \mathbb{D}$

```
1  Procedure transfer (v, σ)
2      switch v do
3          case Entry-node[m_j(p_0, ..., p_n)] do
4              Let c_i be the class of method m_j(p_0, ..., p_n);
5              if There is L_{c_i} then return {this ↦ L_{c_i}(m_j)};
6              else return EmptyMap ;
7          case Call-node[m_j(p_0 : b_0, ..., p_n : b_n)] do
8              Let σ_w be an abstract state of exit_node(v);
9              Initialize σ' := σ;
10             if this not in σ' then
11                 for ap in dom(σ_w) do
12                     ap' = actual_arg(ap{b_0/this}, v);
13                     if ap' in dom(σ) then
14                         E' = (σ[ap'].E ∪ σ_w[ap].E) \ σ_w[ap].D;
15                         D' = (σ[ap'].D ∪ σ_w[ap].D) \ σ_w[ap].E;
16                         P' = σ[ap'].P ∪ (σ_w[ap].P \ σ[ap'].E);
17                         σ'[ap'] = ⟨E', D', P'⟩;
18                     else
19                         σ'[ap'] := σ_w[ap];
20             return σ'
21         otherwise do
22             return σ
```

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 $D = \Sigma^{\bullet}_{c_i} \setminus E$, where $\Sigma^{\bullet}_{c_i}$ is a set of methods for class c_i. This is ensured by condition $well\_formed(\mathcal{L}_{c_i})$ (Definition 2.4) and by the definition of transfer() (see below).

The Transfer Function. The transfer function, given in Algorithm 3, distinguishes between two types of CFG nodes:

Entry-node: (lines 3–6) This is a function entry node. As described in Notation 3.1, for simplicity, we represent it as $m_j(p_0, \ldots, p_n)$ where m_j is a method name and $p_0, \ldots, p_n$ are formal arguments. We assume p_0 is a reference to the receiver object (i.e., *this*). If method m_j is defined in a class c_i with user-supplied annotations $\mathcal{L}_{c_i}$, in line 5, we initialize the domain to the singleton map (i.e., *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 $m_j(p_0 : b_0, \ldots, p_n : b_n)$ (cf. Notation 3.1) where we assume actual arguments $b_0, \ldots, b_n$ are access paths for objects, with b_0 representing 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 an access path ap' given ap. We distinguish two cases: ap denotes (i) a member and (ii) a formal argument of m_j. In line 12, we handle both cases.

In the former case, we know ap has form $this.c_1.\ldots.c_n$. We then construct ap' as ap with $this$ substituted for b_0 (actual_arg(-) is the identity in this case, see Definition 3.7): 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 the formal argument p_i and actual_arg(-) returns the corresponding actual argument b_i (as $p_i\{b_0/this\} = p_i$).

Now, as ap' is determined, we construct its BFA triple. If ap' is not in the domain of σ (line 13) we copy a corresponding BFA triple from σ_w (line 19). Otherwise, we transfer elements of an BFA triple 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 $D = \Sigma^{\bullet}_{c_i} \setminus E$ 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 a constant number of bit-vector operations per argument. That is, our BFA analysis is insensitive to the number of states, as a set of states is abstracted as a single set. Next, we discuss the efficiency of our compositional analysis algorithm by comparing it to the DFA-based approach.

Analysis Complexity: Comparison to DFA-based algorithm. As already mentioned, the performance of a compositional DFA-based analysis depends on the number of states.

In DFA-based analyses, the analysis domain is given by $\mathcal{P}(Q)$, where Q is the set of states. In the intraprocedural analysis, at each method call, the transfer function would need to transition each state in the abstract state according to a given DFA. That is, the transfer function is the DFA's transition function lifted to a subset of states (with signature $\mathcal{P}(Q) \mapsto \mathcal{P}(Q)$). Clearly, the intraprocedural analysis depends linearly in the number of DFA states.

Even more prominently, the compositional interprocedural analysis is affected by the number of states. Each procedure has to be analyzed taking *each* state as an entry state: thus, effectively, we would need to run the intraprocedural analysis $|Q|$ times. Now, as a procedure body can contain branches, the analysis can result in *a set of states* for a given input state: the procedure summary is a mapping from a state into a set of states. For a procedure call, the transfer function would need to apply this mapping, thus taking $|Q|^2$ in the worst case. Overall, the compositional analysis takes $|Q|^3$ operations in the worst-case per a procedure call.

To sum up, taking BFAs as the basis for our analysis, an abstract domain is a set of bit-vectors; also, both transfer and join functions are bit-vector operations. The resulting intraprocedural analysis thus requires a *constant* number of operations per method invocation. More importantly, the compositional analysis also has a constant number of operations per method invocation. In fact, the bit-vector abstraction allows a uniform treatment of intraprocedural analysis and procedure summary computation. That is, our compositional analysis is insensitive to the number of states, which is in sharp contrast with DFA-based analyses.

Implementation. 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 simplifies the handling of branch statements. In principle, however, BFA can be implemented in other frameworks, such as, e.g., IFDS [21].

Correctness. In a BFA, we can abstract a set of states by an *intersection* of states in the set. Let M be a BFA, and Q be its state set. Then, for $S \subseteq Q$ every method call sequence accepted by M *starting* in each state of S is also accepted starting in a state that is an intersection of bits of states in S. Theorem 3.1 formalizes this property. First, we need an auxiliary definition:

Definition 3.9 ($[\![\text{-}]\!](\text{-})$). 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 3.1 (BFA $\cap$-Property). *Suppose $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c}, \mathcal{L}_c)$, $S \subseteq Q$, and $b_* = \bigcap_{q_b \in S} b$. Then we have:*

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

Proof. We show the two items:

(1) By Definition 2.5, for all $q_b \in S$ we know $\delta(q_b, m)$ is defined when $P \subseteq b$ with $\langle E, P, D\rangle = \mathcal{L}_c(m)$. So, we have $P \subseteq \bigcap_{q_b \in P} b = b_*$ and $\delta(q_{b_*}, m)$ is defined.
(2) By induction on $|S|$.
— $|S| = 1$. Follows immediately as $\bigcap_{q_b \in \{q_b\}} q_b = q_b$.
— $|S| > 1$. Let $S = S_0 \cup \{q_b\}$. Let $|S_0| = n$. By IH, we know

$$\bigcap_{q_b \in S_0} [\![\sigma]\!](b) = [\![\sigma]\!]\left(\bigcap_{q_b \in S_0} b\right). \tag{2}$$

We should show

$$\bigcap_{q_b \in (S_0 \cup \{q_{b'}\})} [\![\sigma]\!](b) = [\![\sigma]\!]\left(\bigcap_{q_b \in (S_0 \cup \{q_{b'}\})} b\right)$$

We have

$$\begin{aligned}
\bigcap_{q_b \in (S_0 \cup \{q_{b'}\})} [\![\sigma]\!](b) &= \bigcap_{q_b \in S_0} [\![\sigma]\!](b) \cap [\![\sigma]\!](b') \\
&= [\![\sigma]\!](b_*) \cap [\![\sigma]\!](b') && \text{(by (2))} \\
&= ((b_* \cup E) \setminus D) \cap ((b' \cup E) \setminus D) \\
&= ((b_* \cap b') \cup E) \setminus D && \text{(by set laws)} \\
&= [\![\sigma]\!](b_* \cap b') = [\![\sigma]\!]\left(\bigcap_{q_b \in (S_0 \cup \{q_{b'}\})} b\right)
\end{aligned}$$

where $b_* = [\![\sigma]\!](\bigcap_{q_b \in S_0} b)$. This concludes the proof. □

Our BFA-based algorithm (Algorithm 1) interprets method call sequences in the abstract state and joins them (using the join operator from Definition 3.2) 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 (Definition 3.10) and (2) the soundness of join operator (Definition 3.2). 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 3.10 ($\mathsf{dtransfer}_c(\text{-}, \text{-})$). Let $c \in Classes$ be a class, $\Sigma_c^\bullet$ be a set of methods of c, and $\mathcal{L}_c$ be a BFA mapping. Furthermore, let $m \in \Sigma_c^\bullet$ 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, we define

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

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 Theorem 3.1, we state the soundness of join:

Theorem 3.2 (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 Definition 3.9, Definition 3.2, and set laws we have:

$$\begin{aligned}
[\![\phi_1]\!](b) \cap [\![\phi_2]\!](b) &= ((b \cup E_1) \setminus D_1) \cap ((b \cup E_2) \setminus D_2) \\
&= ((b \cup E_1) \cap (b \cup E_2)) \setminus (D_1 \cup D_2) \\
&= (b \cup (E_1 \cap E_2)) \setminus (D_1 \cup D_2) \\
&= (b \cup (E_1 \cap E_2 \setminus (D_1 \cup D_2)) \setminus (D_1 \cup D_2) \\
&= [\![\phi_1 \sqcup \phi_2]\!](b)
\end{aligned}$$

□

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 3.3 (Correctness of $\mathsf{dtransfer}_c(\text{-}, \text{-})$). *Let $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c}, \mathcal{L}_c)$. Let $q_b \in Q$ and $\widetilde{m} = m_1, \ldots, m_n \in (\Sigma_c^\bullet)^*$. Then*

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

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

Proof. We show the two directions of the equivalence:

— ($\Rightarrow$, Soundness): By induction on n, the length of $\widetilde{m} = m_1, \ldots, m_n$.
– Case $n = 1$. In this case, we have $\widetilde{m} = m_1$. Let $\langle E^m, D^m, \{m_1\} \rangle = \mathcal{L}_c(m_1)$. By Definition 3.10, we have $E' = (\emptyset \cup E^m) \setminus D^m = E^m$ and $D' = (\emptyset \cup D^m) \setminus E^m = D^m$, as E^m and D^m are disjoint, and $P' = \emptyset \cup (\{m\} \setminus \emptyset)$. So, we have $b' = (b \cup E^m) \setminus D^m$. Further, we have $P' \subseteq b$. Finally, by the definition of $\delta(\cdot)$ (from Definition 2.5) we have $\hat{\delta}(q_b, m_1, \ldots, m_n) = q_{b'}$.
– Case $n > 1$. Let $\widetilde{m} = m_1, \ldots, m_n, m_{n+1}$. By IH we know

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E', D', P' \rangle \implies \hat{\delta}(q_b, m_1, \ldots, m_n) = q_{b'} \tag{3}$$

such that $b' = (b \cup E') \setminus D'$ and $P' \subseteq b$. Now, we assume $P'' \subseteq b$ and

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E'', D'', P'' \rangle$$

Then, we should show

$$\hat{\delta}(q_b, m_1, \ldots, m_n, m_{n+1}) = q_{b''} \tag{4}$$

where $b'' = (b \cup E'') \setminus D''$.

Let $\mathcal{L}_c(m_{n+1}) = \langle E^m, D^m, P^m \rangle$. We know $P^m = \{m_{n+1}\}$. By Definition 3.10, we have

$$\begin{aligned}&\mathsf{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle)\\ &\quad = \mathsf{dtransfer}_c(m_{n+1}, \mathsf{dtransfer}_c(m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle))\\ &\quad = \mathsf{dtransfer}_c(m_{n+1}, \langle E', D', P' \rangle)\end{aligned}$$

Further, we have

$$E'' = (E' \cup E^m) \setminus D^m \qquad D'' = (D' \cup D^m) \setminus E^m \qquad P'' = P' \cup (P^m \setminus E') \tag{5}$$

Now, by substitution and De Morgan's laws we have:

$$\begin{aligned}b'' &= (b \cup E'') \setminus D''\\ &= (b \cup ((E' \cup E^m) \setminus D^m)) \setminus ((D' \cup D^m) \setminus E^m)\\ &= ((b \cup (E' \cup E^m)) \setminus (D' \setminus E^m)) \setminus D^m\\ &= (((b \cup E') \setminus D') \cup E^m) \setminus D^m\\ &= (b' \cup E^m) \setminus D^m\end{aligned}$$

Now, by $P'' \subseteq b$, $P'' = P' \cup (P^m \setminus E')$, and $P^m \cap D' = \emptyset$, we have $P^m \subseteq (b \cup E') \setminus D' = b'$ (by (3)). Furthermore, by Definition 2.5, we have

$$\delta(q_{b'}, m_{n+1}) = q_{b''} \tag{6}$$

Now, by the definition of $\hat{\delta}(\cdot)$ we have

$$\hat{\delta}(q_b, m_1, \ldots, m_{n+1}) = \delta(\hat{\delta}(q_b, m_1, \ldots, m_n), m_{n+1})$$

By this, Equations (3), and (6) the goal Equation (4) follows. This concludes this case.

— ($\Leftarrow$, Completeness): By induction on n, the length of $\widetilde{m} = m_1, \ldots, m_n$.

– $n = 1$. In this case $\widetilde{m} = m_1$. Let $\langle E^m, D^m, \{m_1\} \rangle = \mathcal{L}_c(m_1)$. By Definition 2.5 we have $b' = (b \cup E^m) \setminus D^m$ and $\{m_1\} \subseteq b$. By Definition 3.10 we have $E' = E^m$, $D' = D^m$, and $P' = \{m_1\}$. Thus, as $\{m_1\} \cap \emptyset = \emptyset$ we have $b' = [\![\langle E', D', P' \rangle]\!](b)$.

– $n > 1$. Let $\widetilde{m} = m_1, \ldots, m_n, m_{n+1}$. By IH we know

$$\hat{\delta}(q_b, m_1, \ldots, m_n) = q'_b \Rightarrow \mathsf{dtransfer}_c(m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E', D', P' \rangle \tag{7}$$

where $b' = (b \cup E') \setminus D'$ and $P' \subseteq b$. Now, we assume

$$\hat{\delta}(q_b, m_1, \ldots, m_n, m_{n+1}) = q_{b''} \tag{8}$$

We should show that

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E'', D'', P'' \rangle$$

such that $b'' = (b \cup E'') \setminus D''$ and $P'' \subseteq b$. We know

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \mathsf{dtransfer}_c(m_{n+1}, \langle E', D', P' \rangle)$$

By Definition 2.5, we have:

$$\hat{\delta}(q_b, m_1, \ldots, m_n, m_{n+1}) = \delta(\hat{\delta}(q_b, m_1, \ldots, m_n), m_{n+1}) = q_{b''}$$

So by Equations (7) and (8) we have $\{m_{n+1}\} \subseteq b'$ and $b' = (b \cup E') \setminus D'$. It follows $\{m_{n+1}\} \cap D' = \emptyset$. That is, $\mathsf{dtransfer}_c(m_{n+1}, \langle E', D', P' \rangle)$ is defined. Finally, showing that $b'' = (b \cup E'') \setminus D''$ follows by the substitution and De Morgan's laws as in the previous case. This concludes the proof. □

Let us discuss the specialization of Theorem 3.3 for code checking. In this case, we know that a method sequence starts with the constructor method (i.e., the sequence is of the form $m^{\uparrow}, m_1, \ldots, m_n$) and q_{E^c} is the input state. By *well_formed*($\mathcal{L}_c$) (Definition 2.4), we know that if $\delta(q_{E^c}, m^{\uparrow}) = q_b$ and

$$\mathsf{dtransfer}_c(m^{\uparrow}, m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \sigma$$

then methods not enabled in q_b are in the disabling set of σ. Thus, for any sequence $m_1, \ldots, m_{k-1}, m_k$ such that m_k is disabled by the constructor and not enabled in substring $m_1, \ldots, m_{k-1}$, the condition $P \cap D_i \neq \emptyset$ correctly checks that a method is disabled. If *well_formed*($\mathcal{L}_c$) did not hold, the algorithm would fail to detect an error as it would put m_k in P since $m_k \notin E$.

Aliasing. We discuss how *aliasing information* can be integrated into our approach. In Example 3.1, member `lu` of object `foo` can be aliased. Thus, we keep track of BFA triples for all base members instead of constructing an explicit BFA contract for a composed class (e.g., `Foo`). Furthermore, we would need to generalize an abstract state to a mapping of *alias sets* to BFA triples. That is, given a set of access paths $\{a_1, \ldots, a_n\}$, the elements of the abstract state would be $\{a_1, \ldots, a_n\} \mapsto \langle E, D, P \rangle$. For example, when invoking method `setupLU1` we would need to apply its summary (sum_1) to triples 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 triples.

```
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 the bindings of S_1 and S_2 by applying a BFA triple for `this.foo` from sum_1 (that is t_3) to t_1 and t_2. The resulting abstract state d_2 is given at line 3. We remark that if a procedure does not alter aliases, we can soundly compute and apply summaries, as shown above.

4 ANALYZING "MUST CALL" PROPERTIES

Up to here, we have considered the specification of so-called *may call* properties—our BFA abstraction contains states that represent methods that *may* be called at some program point. It is natural to also consider *must call* properties, in which a method *requires* another method to be invoked in a code continuation. In this section, we show how the main ideas of our approach can be accommodated to support the analysis of contracts with "must call" properties, by relying on a conservative extension of our BFA formalism with a "require" annotation.

We note that local contracts involving only "must call" method dependencies also suffer from the state explosion problem. To illustrate this, consider a class that contains n pairs of methods such that one method requires another one to be invoked in a code continuation. Depending on the call history, at any given program point, any subset of n methods is required to be called in a code continuation. As this information must be encoded in states, the corresponding DFA would have 2^n reachable states.

Now we discuss how we refine our abstraction of states (set of states) in the presence of *require annotations*. In the case of enabling/disabling annotations, we showed that states only differ in a set of output edges. We leveraged this fact to abstract *a set of states* into a set of output edges. However, by having the additional "require" annotations there could be two *distinct* states with the same set of output edges where incoming paths of one state can satisfy the "require" annotation, whereas paths of the other state cannot. Furthermore, only states whose incoming paths satisfy all

"require" conditions can be accepting. Therefore, our abstraction of states must include information of required methods in addition to enabled methods. We remark that this refined abstraction still allow us to represent a set of states as a single state.

4.1 Annotation Language Extension

First, we extend the BFA specification language given in Section 2.1 with the following base annotation:

$$\texttt{@Require}(R_i)\ m_i$$

which asserts that invoking method m_i requires invocations of methods in R_i in a code continuation. In other words, a method call sequence starting with m_i is only valid if all methods in R_i are present in the sequence.

We extend the definition of annotation language from Definition 2.1 as follows:

Definition 4.1 (Annotation Language, Extended). Let $\Sigma_c = \{m^{\uparrow}, m_1, \ldots, m_n, m^{\downarrow}\}$ be a set of method names, where we have

— The constructor method $m^{\uparrow}$ is annotated by

$$\texttt{@Enable}(E^c)\ \texttt{@Disable}(D^c)\ \texttt{@Require}(R^c)\ m^{\uparrow}$$

where $E^c \cup D^c = \Sigma_c^{\bullet}$, $E^c \cap D^c = \emptyset$, and $R^c \subseteq E^c$;

— Each m_i for $m_i \in \Sigma_c^{\bullet}$ is annotated by

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

where $E_i \subseteq \Sigma_c^{\bullet}$, $D_i \subseteq \Sigma_c^{\bullet}$, $E_i \cap D_i = \emptyset$, and $R_i \subseteq E_i$.

Let $\tilde{x} = m^{\uparrow}, x_0, x_1, x_2, \ldots$ be a sequence where each $x_i \in \Sigma_c^{\bullet}$. We say that $\tilde{x}$ is *valid (w.r.t. annotations)* if the following holds:

— For all subsequences $\tilde{x}' = x_i, \ldots, x_k$ of $\tilde{x}$ such that $x_k \in D_i$ there is j $(i < j \leq k)$ such that $x_k \in E_j$;

— If $\tilde{x}' = x_i, \ldots$ is a subsequence of $\tilde{x}$ then for each $x_j \in R_i$ there is subsequence $x_i, \ldots, x_j$ in $\tilde{x}'$.

Analogously to $\texttt{@EnableOnly}(E_i)\ m_i$ we can derive $\texttt{@RequireOnly}(R_i)\ m_i$ as follows:

$$\texttt{@RequireOnly}(R_i)\ m_i \stackrel{\text{def}}{=} \texttt{@Enable}(R_i)\ \texttt{@Disable}(\Sigma_c^{\bullet} \setminus R_i)\ \texttt{@Require}(R_i)\ m_i$$

We illustrate the semantics of "$\texttt{@Require}(R_i)\ m_i''$ by appealing to our running example from Figure 2. We wish to refine the contract for class `SparseLU` in such a way that all computed resources *must* be used. For example, a call to method `compute` has to be followed by at least one invocation of method `solve`. The contract in Listing 6 makes use of "$\texttt{@RequireOnly}(R_i)\ m_i''$ to enforce that all computed resources are properly consumed. Compare this "must call" contract to its "may call" counterpart in Listing 5: the only difference is that occurrences of "$\texttt{@EnableOnly}(E_i)\ m_i''$ are substituted by "$\texttt{@RequireOnly}(R_i)\ m_i''$. Also, annotations for a constructor method are inferred similarly: methods that enabled upon an object's creation are those that are unguarded or have weaker annotation guards. Here, we assume that `@EnableOnly` and `@RequireOnly` are stronger guards than `@EnableAll` and `@RequireAll`. Thus, in both 'must call' and 'may call' contracts the only methods enabled in the starting state are `analyzePattern` and `compute`.

Observe that the "must call" contract induces an *extended* BFA (abbreviated BFA* in the following) in which *not all states are accepting* (differently from Figure 2). Such a BFA* is given in Figure 4: there, for instance, state q_2 is not an accepting state: calling `compute()` in q_1 cannot lead to an accepting state as it imposes a requirement to call `solve`. Hence, in order to reach an accepting state

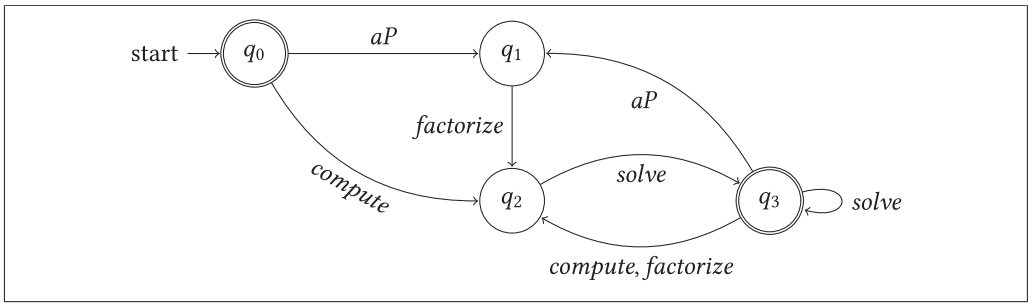


Fig. 4. SparseLU BFA* with Require annotation.

```
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 5. BFA* *May*-Contract for SparseLU.

```
class SparseLU {

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

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

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

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

Listing 6. BFA* *Must*-Contract for SparseLU.

from q_2 this requirement must be satisfied. In this case, a simple call to `solve` in q_2 leads to the accepting state q_3.

Our insight is that every state q should record the accumulated requirements for its outgoing paths, i.e., methods that must be invoked to reach accepting states. For example, the abstraction of state q_2 should contain information that method `solve()` must be an element of a path to an accepting state. Therefore, only states without any such requirements are accepting states. As we have seen, we abstract a state by a bit-vector b, which records enabled methods in a state. Now, our abstraction of a state should also include another bit-vector f that records the accumulated requirements of a state. We now proceed to make these intuitions formal.

4.2 Formalizing the "Must Call" Property

4.2.1 Extended BFA (BFA).* Following the intuition that a state must record requirements for outgoing paths, we extend the state bit-vector representation as follows:

$$q_{b,f}$$

where $b, f \in \mathcal{B}^n$ with n being the number of methods in a class. Here, b represents the enabled methods in a state, as before, and f accumulates require annotations: methods that must be elements of *every* path from $q_{b,f}$ to some accepting state.

Now, we define $\mathcal{L}_c^*$ as the extension of the mapping $\mathcal{L}_c$ from Definition 2.3 as follows:

Definition 4.2 (Mapping $\mathcal{L}_c^*$). Given a class c, we define $\mathcal{L}_c^*$ as a mapping from methods to tuple of subsets of Σ_c:

$$\mathcal{L}_c^* : \Sigma_c \to \left(\mathcal{P}(\Sigma_c^\bullet) \times \mathcal{P}(\Sigma_c^\bullet) \times \mathcal{P}(\Sigma_c^\bullet)\right) \times \left(\mathcal{P}(\Sigma_c^\bullet) \times \mathcal{P}(\Sigma_c^\bullet)\right)$$

Above, the first triple is as before: given $m_i \in \Sigma_c$ we write E_i, D_i, and P_i to denote first three elements of $\mathcal{L}_c^*(m_i)$. There is an additional pair in $\mathcal{L}_c^*(m_i)$, which collects information needed to encode the "must call" property. We shall write R_i and C_i to denote its elements.

Similarly as before, transitions between states $q_{b,f}, q_{b',f'}, \cdots$ are determined by $\mathcal{L}_c^*$. In addition to the semantics of E_i, D_i, and P_i on transitions, we give the following intuitions for R_i and C_i. The set of methods R_i adds the following requirements for subsequent transitions: given $m_i \in \Sigma_c$ we have $l \in R_i$ if and only if m_l must be called after m_i. Dually, C_i records the fulfillment of requirements for a transition. Similarly to P_i, C_i is a singleton set containing method m_i. Again, we define this is as a set to ease the definition of the domain of the compositional analysis algorithm in Section 4.3. We formalize these intuitions as an extension of BFA (Definition 2.5).

Well-formed mapping. We identify some natural well-formedness conditions on the mapping $\mathcal{L}_c^*$. First, we remark that a method cannot require a call to itself, as this would make a self-loop of requirements which cannot be satisfied by any finite sequence. Furthermore, in order to be able to satisfy requirements (i.e., to reach accepting states), we need a condition that require annotations are subset of enabling annotations. We incorporate these conditions in the extension of predicate $well\_formed(\text{-})$ (Definition 2.4):

Definition 4.3 ($well\_formed(\mathcal{L}_c^*)$). Let c, Σ_c, and $\mathcal{L}_c^*$ be a class, its method set, and its mapping, respectively. Then, well_formed($\mathcal{L}_c^*$) = **true** iff the following conditions hold:

— $\mathcal{L}_c^*(m^{\uparrow}) = \langle\langle E^c, D^c, \emptyset\rangle, \langle R^c, \emptyset\rangle\rangle$ such that $E^c \cup D^c = \Sigma_c^{\bullet}$, $E^c \cap D^c = \emptyset$, and $R^c \subseteq E^c$;
— For $m_i \in \Sigma_c$ we have $\mathcal{L}_c^*(m_i) = \langle\langle E_i, D_i, \{m_i\}\rangle, \langle R_i, \{m_i\}\rangle\rangle$ such that

$$E_i, D_i \subseteq \Sigma_c^{\bullet},\ E_i \cap D_i = \emptyset,\ m_i \notin R_i,\ \text{and } R_i \subseteq E_i.$$

We are now ready to extend the definition of BFA from Definition 2.5:

Definition 4.4 (BFA*). Given a $c \in$ *Classes* with $n > 0$ methods, an *extended BFA* (BFA*) for c is defined as a tuple $M = (Q, \Sigma_c^{\bullet}, \delta, q_{E^c,R^c}, \mathcal{L}_c^*, F)$ where:

— Q is a finite set of states $q_{b,f}, q_{b',f'}, \ldots$, where $b, b', \ldots, f, f', \ldots \in \mathcal{B}^n$
— $\Sigma_c^{\bullet} = \{m_1, \ldots, m_n\}$ is the alphabet (method identities);
— q_{E^c,R^c} is the starting state;
— $\delta : Q \times \Sigma_c^{\bullet} \to Q$ is the transition function, where

$$\delta(q_{b,f}, m_i) = q_{b',f'}$$

with $b' = (b \cup E_i) \setminus D_i$ if $P_i \subseteq b$, and is undefined otherwise. Also, $f' = f \setminus C_i \cup R_i$;
— $\mathcal{L}_c^*$ is an extended BFA mapping (cf. Definition 4.2) such that $well\_formed(\mathcal{L}_c^*)$ (cf. Definition 4.3);
— The set of accepting states F is defined as

$$F = \{q_{b,0^n} : q_{b,0^n} \in Q\}$$

The definition of F captures the intuition that a state is accepting only if it has no outstanding requirements, i.e., its bit-vector f is the zero-vector.

We now need to show that a well-formed $\mathcal{L}_c^*$ ensures that its induced BFA* has reachable accepting states. This boils down to showing that in each state, the required bit set f is contained in the enabled bit set b:

Lemma 4.5. *Let $M = (Q, \Sigma_c^{\bullet}, \delta, q_{E^c,R^c}, \mathcal{L}_c^*, F)$ be a BFA*. Then, for $q_{b,f} \in Q$ we have $f \subseteq b$.*

Proof. First, we can see that initial state q_{E^c,R^c} trivially satisfies $f \subseteq b$. Furthermore, let $q_{b,f} \in Q$ such that $f \subseteq b$. Then, for $m_i \in \Sigma_c^\bullet$ we have

$$\delta(q_{b,f}, m_i) = q_{b',f'}$$

with $b' = (b \cup E_i) \setminus D_i$ if $P_i \subseteq b$, and is undefined otherwise. Also, $f' = f \setminus C_i \cup R_i$. Now, the goal $f' \subseteq b'$ follows by this and conditions $E_i \cap D_i = \emptyset$ and $R_i \subseteq E_i$ ensured by well_formed($\mathcal{L}_c^*$) (Definition 4.3). □

We illustrate states and transitions of a BFA* given in Figure 4 in the following example:

Example 4.6 (SparseLU must-contract). The mapping $\mathcal{L}^*_{\text{SparseLU}}$ that corresponds to the contract given in Listing 6 is as follows:

$$\begin{aligned}\mathcal{L}^*_{\text{SparseLU}} = \big\{&0 \mapsto \langle\langle\{1,2\},\{3,4\},\emptyset\rangle,\langle\emptyset,\emptyset\rangle\rangle,\ 1 \mapsto \langle\langle\{3\},\{1,2,4\},\{1\}\rangle,\langle\{3\},\{1\}\rangle\rangle,\\ &2 \mapsto \langle\langle\{4\},\{1,2,3\},\{2\}\rangle,\langle\{4\},\{2\}\rangle\rangle,\ 3 \mapsto \langle\langle\{4\},\{1,2,3\},\{3\}\rangle,\langle\{4\},\{3\}\rangle\rangle,\\ &4 \mapsto \langle\langle\{1,2,3\},\emptyset,\{4\}\rangle,\langle\emptyset,\{4\}\rangle\rangle\big\}\end{aligned}$$

The starting state is $q_{1100,0000}$. The set of states is

$$Q = \{q_{1100,0000}, q_{0010,0010}, q_{0001,0001}, q_{1111,0000}\}$$

Differently from the contract given in Example 2.6, in which all states were accepting, here we have an explicit set of accepting states:

$$F = \{q_{1100,0000}, q_{1111,0000}\}.$$

The corresponding transition function δ(-) is as follows:

$$\begin{aligned}&\delta(q_{1100,0000}, aP) = q_{0010,0010} && \delta(q_{1100,0000}, compute) = q_{0010,0010}\\ &\delta(q_{0010,0010}, factorize) = q_{0001,0001} && \delta(q_{0001,0001}, solve) = q_{1111,0000}\\ &\delta(q_{1111,0000}, aP) = q_{0010,0010} && \delta(q_{1111,0000}, compute) = q_{0001,0001}\\ &\delta(q_{1111,0000}, factorize) = q_{0001,0001} && \delta(q_{1111,0000}, solve) = q_{1111,0000}\end{aligned}$$

Notice that the transformations of b-bits of states are as in Example 2.6. Additionally, transitions operate on f-bits to determine the accepting states. For example, the transition

$$\delta(q_{1111,0000}, compute) = q_{0001,0001}$$

adds the requirement to call `solve` by f-bits 0001. This is satisfied in transition $\delta(q_{0001,0001}, solve) = q_{1111,0000}$. As the f-bits of $q_{1111,0000}$ are all zeros, this state is accepting. ◁

BFA subtyping.* We now discuss the extension of the subtyping relation given in Section 2.2. In order to check that c_1 is a superclass of c_2, that is that M_2 subsumes M_1 ($M_2 \succeq M_1$), additionally to checking respective E, D, and P sets of $\mathcal{L}^*_{c_1}$ and $\mathcal{L}^*_{c_2}$ for each method, as given in Section 2.2, we need the following checks: $R_2 \subseteq R_1$ and $C_1 \subseteq C_2$. This follows the intuition that a superclass must be at least permissive as its subclasses: the subclass methods can only have less requirements.

4.3 An Extended Algorithm

We now present the extension of the compositional analysis algorithm to account for BFAs*. We illustrate the key ideas of the required extensions with an example.

Example 4.7. In Listing 7, we give class `Bar` that has a member `lu` of `SparseLU` and implements two methods that make calls on `lu`; Listing 8 contains a client code for class `Bar`. Now we illustrate how a summary is computed in the presence of a "require" annotation for `setupLU_must()` and `solveLU_must()`.

```
1  class Bar {
2      SparseLU lu; Matrix a, b;
3      void solveLU_must() {
4          this.lu.solve(this.b); }
5      void setupLU_must() {
6        if (?) {
7            this.lu.analyzePattern();
8            this.lu.factorize();
9        } else {
10           this.lu.compute(); } } }
```

Listing 7. Class Bar using SparseLU.

```
void useBar() {
    Bar bar;
    bar.setupLU_must();
    bar.solveLU_must();
}
```

Listing 8. Client code for Bar.

Analogously to how our original algorithm accumulates enabling annotations by traversing a program's CFG, in the extension, we will accumulate require annotations. We extend the abstract domain with a pair $\langle R, C \rangle$, where R and C are sets of methods in which we will appropriately accumulate require annotations. Intuitively, we use R to record call requirements for a code continuation and C to track methods that have been called up to a current code point.

First, we compute a summary for `solveLU_must()` as follows:

```
1 void solveLU_must() {
2     // s1 = this.lu -> ({}, {})
3     this.lu.solve();
4     // s2 = this.lu -> ({}, {solve})
5     // sum_solveLU = s2
6 }
```

At procedure entry, we initialize the abstract state as an empty pair (s_1). Next, on the invocation of `solve()`, we simply copy the corresponding annotations from $\mathcal{L}^*_{SparseLU}(solve)$. Therefore, the summary `sum_solveLU` essentially only records that `solve` is called within this procedure.

Next, we compute a summary for `setupLU_must`:

```
1  void setupLU_must(Matrix b) {
2      // s1 = this.lu -> ({}, {})
3      if (?) {
4          this.lu.analyzePattern();
5          // s2 = this.lu -> ({factorize}, {})
6          this.lu.factorize();
7          // s3 = this.lu -> ({solve}, {})
8      } else {
9          this.lu.compute();
10         // s4 = this.lu -> ({solve}, {})
11     }
12     // join s3 s4 = s5
13     // s5 = this.lu -> ({solve}, {})
14     // sum_setupLU = s5
15 }
```

In the first if-branch, on line 4, we copy the corresponding annotations from $\mathcal{L}^*_{SparseLU}(aP)$ to obtain s_2. Here, we remark that `factorize` is in the require set of s_2. Next, on line 6 on the invocation of `factorize()` we remove `factorize` from the require set of s_2 and add its requirements, i.e., `solve` to the require set of s_2 to obtain s_3. Similarly, we construct s_4 on line 9.

Now, on line 12, we should join the resulting sets of the two branches, that is, s_3 and s_4. For this we take the union of the require sets and the intersection of the called sets: this follows the intuition that a method must be called in a continuation if it is required within *any* branch; dually, a method call required prior to branching is satisfied only if it is invoked in *both* branches.

Once summaries for `solveLU_must()` and `setupLU_must()` are computed, we can check the client code `useBar`:

```
1  void useBar() {
2      Bar bar;
3      // b1 = bar -> ({}, {})
4      bar.setupLU_must();
5      // copy sum_setupL1 to get b2
6      // b2 = bar -> ({solve}, {})
7      bar.solveLU_must();
8      // apply sum_solveLU to b2 to get b3
9      // b3 = bar -> ({}, {})
10     bar.destructor(); // explicit call to a destructor
11     // bar can be destructed here as there are no requirements for it
12     // that is, b3[bar].R is the empty set
13 }
```

Here, on line 4, we simply copy the summary computed for method `setupLU_must()`. Next, on line 7, we apply the summary of `solveLU_must()` to the current abstract state b_1 to obtain b_2: the resulting require set of b_3 is obtained by taking an union of the current require set and the summary's require set (the first component of `sum_solveLU`) and by removing elements of the called set (the second component of `sum_solveLU`) from it. The resulting called set is the union of the current called set and the called set of the summary. Finally, when `destructor` method is called (line 10) we check if there are any outstanding requirements for object `bar`: i.e., if a required set of the current abstract state is empty. As the required set in b_3 is empty, there no warning is raised. ◁

We show how to extend our compositional analysis algorithm from Section 3 to incorporate analysis of "must call" properties.

Abstract Domain. First, we recall that our abstract domain $\mathbb{D}$ is a mapping from access paths to elements of mapping $\mathcal{L}_c$. Given the extended mapping $\mathcal{L}_c^*$, this is reflected on the abstract domain as follows:

$$\mathbb{D} : \mathcal{AP} \rightarrow \bigcup_{c \in Classes} Cod(\mathcal{L}_c^*)$$

The elements of the co-domain have now the following form:

$$\big\langle \langle E, D, P \rangle, \langle R, C \rangle \big\rangle$$

where $R, C \subseteq \Sigma_c^\bullet$. Intuitively, R is a set of methods that must be called in a code continuation, and C is a set of methods that have been called up to the current program point.

Algorithm. We modify the algorithm to work with an abstract domain extended with the pair $\langle R, C \rangle$. To this end, we extend (i) the join operator, (ii) the guard predicate (Algorithm 2), and (iii) the transfer function (Algorithm 3). Next, we discuss these extensions.

Join operator. The modified join operator has the following signature:

$$\bigsqcup : Cod(\mathcal{L}_c^*) \times Cod(\mathcal{L}_c^*) \rightarrow Cod(\mathcal{L}_c^*)$$

Its definition is conservatively extended as follows:

$$\begin{aligned}&\left\langle\langle E_1, D_1, P_1\rangle, \langle R_1, C_1\rangle\right\rangle \sqcup \left\langle\langle E_2, D_2, P_2\rangle, \langle R_2, C_2\rangle\right\rangle \\ &\quad = \left\langle\langle E_1 \ \cap \ E_2 \setminus (\ D_1 \cup D_2),\ D_1 \cup D_2,\ P_1 \cup \ P_2\rangle,\ \langle R_1 \cup R_2,\ C_1 \cap C_2\rangle\right\rangle\end{aligned}$$

Guard predicate. In Algorithm 2, in the body of case Call-node$[m_j(p_0 : b_0, \ldots, p_n : b_n)]$ we add the following check after line 4:

$$\textbf{if } m_j == \textit{destructor} \textbf{ and } \sigma_w[p_0].R \neq \emptyset \textbf{ then return } \textbf{\textit{False}};$$

In the case m_j is destructor, we additionally check whether its requirements are empty; if not we raise a warning.

Transfer function. In Algorithm 3, we add the following lines after line 16 to transfer the new elements $\langle R, C\rangle$:

$$\begin{aligned}R' &= \left(\sigma(ap).R \cup \sigma_w(ap).R\right) \setminus \sigma_w(ap).C \\ C' &= (\sigma(ap).C \cup \sigma_w(ap).C) \setminus \sigma_w(ap).R\end{aligned}$$

Then, the output abstract state σ' is constructed as follows:

$$\sigma'(ap') = \left\langle\langle E', D', P'\rangle, \langle R', C'\rangle\right\rangle$$

where E', D', and P' are constructed as in Algorithm 3.

4.4 Extended Proofs of Correctness

Here, we present the correctness guarantees for BFAs*. We describe the needed extensions to the definitions, theorems, and proofs we discussed in the case of BFA. As we will see, all the correctness properties that hold for "may call" contracts, hold for "must call" contracts as well. Hence, we confirm that the main ideas of our bit-vector abstraction of DFAs are not limited to "may call" properties that we initially focused on: the principles of our abstraction can be applied to "must call" properties too.

Context-independence. Here, we characterize context-independence property for required annotations. Recall that context-independence states that the effects of annotations on subsequent calls do not depend on previous calls. Similarly, in the case of enabling/disabling annotations, this property directly follows from the idempotence of the operation on f-bits in the extended definition of δ(-), that is, $f' = (f \setminus C_i) \cup R_i$. The effect of this operation is independent of bits in f, which are accumulated by preceding calls (i.e., they represent a context).

Now, we formalize the extension of the statement and proof. First, as not all states in a BFA* are accepting, the definition of $L(M)$ that denotes strings accepted by M is now as follows:

$$L(M) = \{\widetilde{m} : \hat{\delta}(q_{E^c,R^c}, \widetilde{m}) = q' \wedge q' \in F\}$$

Consequently, we need to reformulate statements of first two items to preserve their meanings, and add the item concerning require annotations. Thus, we extend Theorem 2.1 as follows:

Theorem 4.1 (Context-independence, Extended). *Let $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c,R^c}, \mathcal{L}_c^*, F)$ be a BFA*. Then, for $m_n \in \Sigma_c^\bullet$ we have*

(1) If there is $\widetilde{p}_1 \in (\Sigma_c^\bullet)^*$ and $m_{n+1} \in \Sigma_c^\bullet$ such that for any $\widetilde{s}_2 \in (\Sigma_c^\bullet)^*$ we have $\widetilde{p}_1 \cdot m_{n+1} \cdot \widetilde{s}_2 \notin L(M)$ and there is $\widetilde{p}_2 \in (\Sigma_c^\bullet)^*$ such that $\widetilde{p}_1 \cdot m_n \cdot m_{n+1} \cdot \widetilde{p}_2 \in L(M)$ then there is no $\widetilde{m} \in (\Sigma_c^\bullet)^*$ such that $\widetilde{m} \cdot m_n \cdot m_{n+1} \cdot \widetilde{s}_2 \notin L(M)$ for all $\widetilde{s}_2 \in (\Sigma_c^\bullet)^*$.

(2) If there is $\widetilde{p}_1, \widetilde{p}_2 \in (\Sigma_c^\bullet)^*$ and $m_{n+1} \in \Sigma_c^\bullet$ such that $\widetilde{p}_1 \cdot m_{n+1} \cdot \widetilde{p}_2 \in L(M)$ and $\widetilde{p} \cdot m_n \cdot m_{n+1} \cdot \widetilde{s}_2 \notin L(M)$ for all $\widetilde{s}_2 \in \Sigma_c^{\bullet *}$ then there is no $\widetilde{m}_1, \widetilde{m}_2 \in (\Sigma_c^\bullet)^*$ such that $\widetilde{m}_1 \cdot m_n \cdot m_{n+1} \cdot \widetilde{m}_2 \in L(M)$.

(3) If there are $\widetilde{p}_1, \widetilde{p}_2 \in L(M)$ *and* $m_n \in \Sigma_c^\bullet$ *such that* $\widetilde{p}_1 \cdot m_n \cdot \widetilde{p}_2 \in L(M)$ *then there is no* $\widetilde{m} \in L(M)$ *with* $\mathcal{L}_c^*(m).R = \emptyset$ *for* $m \in \widetilde{m}$ *such that* $\widetilde{m} \cdot m_n \cdot \widetilde{p}_2 \notin L(M)$.

Proof. This property follows directly by the definition of transition function δ(-) in BFAs* (Definition 4.4): that is, by the idempotence of b and f-bits transformation. More precisely, the effects of transformations $b' = (b \cup E_i) \setminus D_i$ (resp. $f' = (f \setminus C_i) \cup R_i$) do not depend on input bits b (resp. f).

The first two items are shown similarly as in the proof of Theorem 2.1. We remark that additional sequences are only introduced in order to properly use the definition of $L(M)$ for BFAs*.

Now we show item (3). We prove directly by the extended definition of the transition function δ(-), that is by $f' = (f \setminus C_i) \cup R_i$. First, let $q_{b,f}$ be defined as follows:

$$\delta(q_{E^c,R^c}, \widetilde{p}_1 \cdot m_n) = q_{b,f}$$

Further, by $\widetilde{p}_1 \cdot m_n \cdot \widetilde{p}_2 \in L(M)$ we have $\widetilde{p}_2 \supseteq f$, as $q_{b',0^n} \in F$ by the definition of F. By this we have $\widetilde{p}_2 \supseteq R_n$ as $R_n \subseteq f$. Finally, as $\mathcal{L}_c^*(m).R = \emptyset$ for $m \in \widetilde{m}$ we have

$$\delta(q_{E^c,R^c}, \widetilde{m} \cdot m_n) = q_{b',R_n}$$

Using this and $\widetilde{p}_2 \supseteq R_n$ we have $\widetilde{m} \cdot m_n \cdot \widetilde{p}_2 \in L(M)$. □

BFA ∩-Property. We first extend ⟦-⟧(-) from Definition 3.9 to operate on both b-bits and f-bits:

Definition 4.8 (⟦-⟧(-) Extended). Let $\langle\langle E, D, P\rangle, \langle R, C\rangle\rangle \in Cod(\mathcal{L}_c^*)$, $b, f \in \mathcal{B}^n$. We define

$$[\![\langle\langle E, D, P\rangle, \langle R, C\rangle\rangle]\!](b, f) = b', f'$$

where $b' = (b \cup E) \setminus D$ if $P \subseteq b$, and is undefined otherwise; and $f' = (f \setminus C) \cup R$.

Now, to abstract the set of states of a BFA* we also need to handle f-bits of states. Complementary to b^*, we define f^* as *the union* of f-bits. Now, we extend Theorem 3.1 by incorporating f-bits in states and also item (3), which shows that a union of f-bits is the right way to abstract set of states P into a single state: intuitively, set of states P can be abstracted into the accepting state only if all states in P are accepting.

Theorem 4.2 (BFA* ∩-Property). *Suppose* $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c,R^c}, \mathcal{L}_c^*, F)$, $S \subseteq Q$, $b_* = \bigcap_{q_b \in P} b$, *and* $f^* = \bigcup_{q_{b,f} \in P} f$. *Then we have:*

(1) For $m \in \Sigma_c^\bullet$, *it holds:* $\delta(q_{b,f}, m)$ *is defined for all* $q_{b,f} \in S$ *iff* $\delta(q_{b_*,f_*}, m)$ *is defined.*
(2) Let $\sigma = \mathcal{L}_c^*(m)$. *If* $S' = \{\delta(q_{b,f}, m) : q_{b,f} \in S\}$ *then* $\bigcap_{q_{b,f} \in S'} b, \bigcup_{q_{b,f} \in S'} f = [\![\sigma]\!](b_*, f_*)$.
(3) $S \subseteq F$ *if and only if* $f^* = 0^n$.

Proof. The first item is only concerned with b-bits, thus it is shown as in Theorem 3.1.

Now, we discuss the proof for item (2). Here, we can separately prove the part for b-bits and for f-bits. The former proof is the same as in the corresponding case of Theorem 3.1. Moreover, the proof concerning f-bits follows the same lines as for b-bits (by induction on the cardinality of S and set laws): it again directly follows by the idempotence of transformation of f-bits (i.e., $f' = (f \setminus C_i) \cup R_i$); we remark that difference here is that we use the union (in the definition of f^* bits) instead of the intersection.

Finally, the proof of item (3) follows directly from the definition of accepting states, that is $F = \{q_{b,0^n} : q_{b,0^n} \in Q\}$. Thus, we know $S \subseteq F$ if and only if for all $q_{b,f} \in S$ we have $f = 0^n$. The right-hand side is equivalent to $f^* = 0^n$. □

Soundness of join operator. We extend Theorem 3.2 with f-bits in the state representation, $\langle R_i, C_i\rangle$ in ϕ_i, and using the extended $[\![\text{-}]\!](\text{-})$ from Definition 4.8. We note that this theorem again relies on Theorem 4.2: we abstract set of reachable states by the union of f-bits.

For convenience, we will use "projections" of $[\![\text{-}]\!](\text{-})$ to b and f-bits. Let $\phi = \langle\langle E, D, P\rangle, \langle R, C\rangle\rangle$, then will use $[\![\phi]\!]_b(b) = b'$ and $[\![\phi]\!]_f(f) = f'$, where b' and f' are defined as in Definition 4.8.

Theorem 4.3 (Soundness of Extended $\sqcup$). *Let $q_{b,f} \in Q$ and $\phi_i = \langle\langle E_i, D_i, P_i\rangle, \langle R_i, C_i\rangle\rangle$ for $i \in \{1, 2\}$. Then, $[\![\phi_1]\!]_b(b) \cap [\![\phi_2]\!]_b(b) = [\![\phi_1 \sqcup \phi_2]\!]_b(b)$ and $[\![\phi_1]\!]_f(f) \cup [\![\phi_2]\!]_f(f) = [\![\phi_1 \sqcup \phi_2]\!]_f(f)$.*

Proof. The proof concerning b-bits is the same as in Theorem 3.2. Now, we show the part concerning f-bits, that is

$$[\![\phi_1]\!]_f(f) \cup [\![\phi_2]\!]_f(f) = [\![\phi_1 \sqcup \phi_2]\!]_f(f)$$

The proof follows by the extended definition of $[\![\text{-}]\!](\text{-})$ from Definition 4.8 and set laws as follows:

$$\begin{aligned}[\![\phi_1]\!](f) \cup [\![\phi_2]\!](f) &= ((f \setminus C_1) \cup R_1) \cup ((f \setminus C_2) \cup R_2)\\ &= (f \setminus (C_1 \cap C_2)) \cup (R_1 \cup R_2) = [\![\phi_1 \sqcup \phi_2]\!](f)\end{aligned}$$ □

Correctness of $\mathsf{dtransfer}_c(\text{-}, \text{-})$. We extend $\mathsf{dtransfer}_c(\text{-}, \text{-})$ from Definition 3.10 to account for the extended transfer function as follows:

Definition 4.9 ($\mathsf{dtransfer}_c(\text{-}, \text{-})$*).* Let $c \in$ *Classes* be a class, $\Sigma_c^\bullet$ be a set of methods of c, and $\mathcal{L}_c^*$ be a BFA*. Furthermore, let $m \in \Sigma_c^\bullet$ be a method, $\langle\langle E^m, D^m, P^m\rangle, \langle R^m, C^m\rangle\rangle = \mathcal{L}_c^*(m)$, and $\langle\langle E, D, P\rangle, \langle R, C\rangle\rangle \in Cod(\mathcal{L}_c^*)$. Then,

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

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. Also, $R' = (R \cup R^m) \setminus C^m$ and $C' = (C \cup C^m) \setminus R^m$.

Let $m_1, \ldots, m_n, m_{n+1}$ be a method sequence and $\phi = \big\langle\langle E, D, P\rangle, \langle R, C\rangle\big\rangle$, then

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

We now extend Theorem 3.3 to show the correctness of the extended $\mathsf{dtransfer}_c(\text{-}, \text{-})$ as follows:

Theorem 4.4 (Correctness of $\mathsf{dtransfer}_c(\text{-}, \text{-})$). *Let $M = (Q, \Sigma_c^\bullet, \delta, q_{E^c,R^c}, \mathcal{L}_c^*, F)$. Let $q_{b,f} \in Q$ and $m_1, \ldots, m_n \in (\Sigma_c^\bullet)^*$. Then*

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, \big\langle\langle \emptyset, \emptyset, \emptyset\rangle,\ \langle \emptyset, \emptyset\rangle\big\rangle) = \phi' \iff \hat{\delta}(q_{b,f}, m_1, \ldots, m_n) = q_{b',f'}$$

such that $b', f' = [\![\phi']\!](b, f)$ where $\phi' = \big\langle\langle E', D', P'\rangle,\ \langle R', C'\rangle\big\rangle$.

Proof. The proof concerning b-bits is as in Theorem 3.3. Now, we will prove the part concerning transformation of f-bits.

We show only the Soundness ($\Rightarrow$) direction as the other direction is shown similarly. The proof is by induction. We strengthen the induction hypothesis with the following invariant: $R' \cap C' = \emptyset$.

- Case $n = 1$. We have $\widetilde{m} = m_1$. Let $R^m = \mathcal{L}_c^*(m_1).R$ and $C^m = \mathcal{L}_c^*(m_1).C$. First by the definition of $\mathsf{dtransfer}_c(\text{-})$ we have $R' = (\emptyset \cup R^m) \setminus C^m = R^m$ and $C' = (\emptyset \cup C^m) \setminus R^m = C^m$. Thus, we have $f' = [\![\phi']\!]_f(f) = (f \setminus C^m) \cup R^m$. Thus, directly by the definition of $\delta(\text{-})$ we have $\delta(q_{b,f}, m_1) = q_{b',f'}$.
- Case $n > 1$. Let $\widetilde{m} = m_1, \ldots, m_n, m_{n+1}$. By IH we know

$$\mathsf{dtransfer}_c(m_1, \ldots, m_n, \big\langle\langle \emptyset, \emptyset, \emptyset\rangle,\ \langle \emptyset, \emptyset\rangle\big\rangle) = \phi' \Rightarrow \hat{\delta}(q_{b,f}, m_1, \ldots, m_n) = q_{b',f'} \tag{9}$$

such that $b', f' = [\![\phi']\!](b, f)$ and $f' \subseteq b'$ where $\phi' = \langle\langle E', D', P'\rangle, \langle R', C'\rangle\rangle$. As we focus only on f' bits, we can infer $f' = (f \setminus C') \cup R'$. Now, we assume

$$\text{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle\langle \emptyset, \emptyset, \emptyset\rangle, \langle \emptyset, \emptyset\rangle\rangle) = \phi'' \tag{10}$$

such that $\phi'' = \langle\langle E'', D'', P''\rangle, \langle R'', C''\rangle\rangle$. We should show

$$\hat{\delta}(q_{b,f}, m_1, \ldots, m_n, m_{n+1}) = q_{b'',f''} \tag{11}$$

such that $f'' = (f \setminus C'') \cup R''$ and $f'' \subseteq b''$. Let $\mathcal{L}_c^*(m_{n+1}) = \langle R^m, C^m\rangle$. We know $C^m = \{m_{n+1}\}$. By Definition 3.10 we have

$$\text{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle\langle \emptyset, \emptyset, \emptyset\rangle, \langle \emptyset, \emptyset\rangle\rangle) = \text{dtransfer}_c(m_{n+1}, \phi')$$

Furthermore, by Equations (9), (10) and Definition 3.10 we have:

$$R'' = (R' \cup R^m) \setminus C^m$$
$$C'' = (C' \cup C^m) \setminus R^m$$

Here, we remark that the invariant $R'' \cap C'' = \emptyset$ holds as $R^m \cap C^m = \emptyset$ by $well\_formed(\mathcal{L}_c^*)$ (Definition 4.3). Now, by substitution and De Morgan's laws we have:

$$\begin{aligned} f'' &= (f \setminus C'') \cup R'' \\ &= (f \setminus (C' \cup C^m)) \cup ((R' \cup R^m) \setminus C^m) \\ &= (((f \setminus C') \cup R') \setminus C^m) \cup R^m \\ &= ((f' \setminus C^m) \cup R^m \end{aligned}$$

where the third equivalence holds by invariant $R' \cap C' = \emptyset$ and $R^m \cap C^m = \emptyset$. Furthermore, by the definition of $\delta(\text{-})$ (from Definition 2.5) we have $\delta(q_{b',f'}, m_{n+1}) = q_{b'',f''}$. This concludes this case. □

Summing up, we presented BFAs*, the extension to BFAs that allow us to specify both "may call" and "must call" properties, while enabling the bit-vector representation of states and transitions in the underlying BFA*. The bit-vector abstraction provides noticeable scalability benefits in terms of both specification and the code analysis. Next, we present the usability and performance evaluations that substantiate the claim of the smaller annotation overhead, as well as theoretical discussions of the algorithm performance improvements over DFA-based techniques.

5 EVALUATION

To evaluate our technique, 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 analysis technique introduced in this article.
(2) DFA: A lightweight, DFA-based typestate analyzer implemented in Infer. We translate BFA* annotations to a minimal DFA and perform the analysis.
(3) Topl: An industrial typestate analyzer, implemented in Infer [1].

We remark that Topl 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 that we could hypothetically use. We note our benchmarks do not require aliasing and in theory Pulse is not required.

Goals and Considered Contracts. Our evaluation aims to validate the following two claims:

Claim-I: Reduced annotation overhead. The BFA* contract annotation overheads are smaller in terms of atomic annotations (e.g., @Post(...), @Enable(...)) than the two 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.

We analyzed a benchmark of 22 contracts that specify common patterns of locally dependent contract annotations for a class. Of these, 18 are may contracts and 4 are must contracts. We identified common patterns of locally dependent contracts, such as the setter/getter example given in Figure 1, and generated variants of them (e.g., by varying annotations and number of methods) such that we have contract samples that are *almost* linearly distributed in the number of (DFA) states. This can be seen in Figure 7, which outlines key features of these contracts (such as number of methods and number of states). The annotations for BFA* are varied; from them, we generated minimal DFA representations in the DFA annotation format and Topl annotation format. This allows us to clearly show how the performance of the analyzers under consideration is impacted by the increase of the state space.

Moreover, we self-generated 122 client programs that follow the compositional patterns we described in Example 3.1 (this kind of patterns are also considered in, e.g., [14]). The pattern defines a composed class, as the class `Bar` illustrated at the end of Section 3.1, that has an object member of classes that have declared contracts (recall that we refer to those as base classes). Each of the methods of the composed class invokes methods on its members. Thus, a compositional analysis computes procedure summaries of these methods; this way, it effectively infers a contract of the composed class based on those of its class members. We remark that a composed class can itself be a member of another composed class, as expected. This pattern depends on important parameters, namely, number of composed classes, lines of code (i.e., number of method invocations), if-branches, and loops. The self-generation that follows this pattern allows us to precisely vary those parameters and measure their impact on the analysis performance. Note, the code is such that it does not appeal to aliasing (as we do not support it yet in our BFA* implementation).

5.1 Experimental Setup

We used an Intel(R) Core(TM) i9-9880H CPU at 2.3 GHz with 16 GB 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.

Our use case is to integrate static analyses in interactive IDEs e.g., Microsoft Visual Studio Code [24], 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 [2].

5.2 Usability Evaluation

Figure 7 outlines the key features of the 22 contracts we considered, called CR-1 – CR-22. Among these, CR-12, CR-14, CR-17, and CR-22 are *must* contracts. For each contract, we specify the number of methods, the number of DFA states the contract corresponds to, and number of atomic

```
class SparseLU {
    states q0, q1, q2, q3, q4;
    @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)
    @Pre(q4) @Post(q3)
    void solve(Mat b);
    @Pre(q2) @Post(q4)
    @Pre(q3) @Post(q4)
    void transpose();}
```

Listing 9. DFA CR4 contract for SparseLU

```
class SparseLU {
  SparseLU();

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

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

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


  @EnableAll
  void solve(Mat b);

  @Disable(transpose)
  void transpose(); }
```

Listing 10. BFA CR4 contract for SparseLU

```
property SparseLU
  prefix "SparseLU"
  start -> start: *
  start -> q0: SparseLU() => x := RetFoo
  q1 -> q2: analyzePattern(SparseLU, IgnoreRet) when SparseLU == x
  q3 -> q2: analyzePattern(SparseLU, IgnoreRet) when SparseLU == x
  q1 -> q2: factorize(SparseLU, IgnoreRet) when SparseLU == x
  q3 -> q2: factorize(SparseLU, IgnoreRet) when SparseLU == x
  q0 -> q2: compute(SparseLU, IgnoreRet) when SparseLU == x
  q3 -> q2: compute(SparseLU, IgnoreRet) when SparseLU == x
  q2 -> q3: solve(SparseLU, IgnoreRet) when SparseLU == x
  q2 -> q4: transpose(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> q2: transpose(SparseLU, IgnoreRet) when SparseLU == x
  q2 -> error: analyzePattern(SparseLU, IgnoreRet) when SparseLU == x
  q3 -> error: analyzePattern(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> error: analyzePattern(SparseLU, IgnoreRet) when SparseLU == x
  q0 -> error: factorize(SparseLU, IgnoreRet) when SparseLU == x
  q2 -> error: factorize(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> error: factorize(SparseLU, IgnoreRet) when SparseLU == x
  q1 -> error: compute(SparseLU, IgnoreRet) when SparseLU == x
  q2 -> error: compute(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> error: compute(SparseLU, IgnoreRet) when SparseLU == x
  q1 -> error: solve(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> error: solve(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> error: solve(SparseLU, IgnoreRet) when SparseLU == x
  q0 -> error: transpose(SparseLU, IgnoreRet) when SparseLU == x
  q1 -> error: transpose(SparseLU, IgnoreRet) when SparseLU == x
  q4 -> error: transpose(SparseLU, IgnoreRet) when SparseLU == x
```

Listing 11. TOPL CR4 contract for SparseLU

Fig. 5. DFA, BFA*, and TOPL specifications of CR4 contract for SparseLU class.

annotation terms in BFA*, DFA, and Topl. An atomic annotation term is a standalone annotation in the given annotation language. In Figures 5 and 6, we detail CR-4 as an example.

Figure 7 shows that as the contract sizes increase in number 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 four 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.

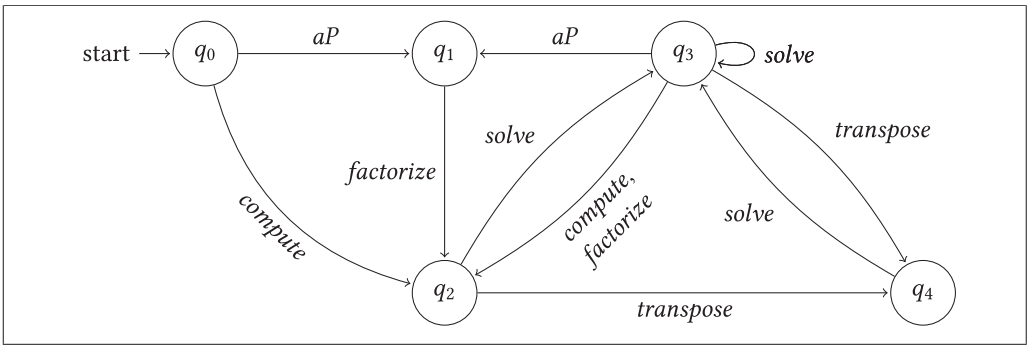


Fig. 6. DFA for the class SparseLU (CR-4 contract). This contract extends the SparseLU contract from Example 2.6 with an additional method (transpose). The intention is to capture the fact that consecutive calls to transpose are redundant. In Figure 5, we can see how this extension is specified in the three specification languages under consideration (DFA, TOPL, and BFA*).

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

Contract	#methods	#states	#BFA	#DFA	#TOPL
CR-12*	14	998	24	8185	20295
CR-13	14	1044	32	7766	20704
CR-14*	14	1358	24	10669	27265
CR-15	14	1628	21	13558	33740
CR-16	14	2104	25	17092	43305
CR-17*	14	2322	21	15529	47068
CR-18	14	2644	24	26014	61846
CR-19	16	3138	29	38345	88134
CR-20	18	3638	23	39423	91120
CR-21	18	4000	27	41092	101185
CR-22*	14	4510	27	36947	93615

Fig. 7. Details of the 22 contracts in our evaluation. Contracts marked with "*" include Require annotations.

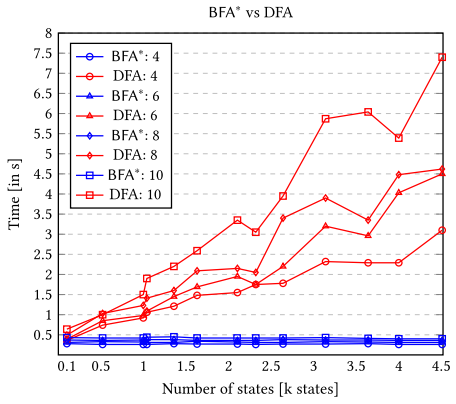
5.3 Performance Evaluation

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 Figure 8, 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).

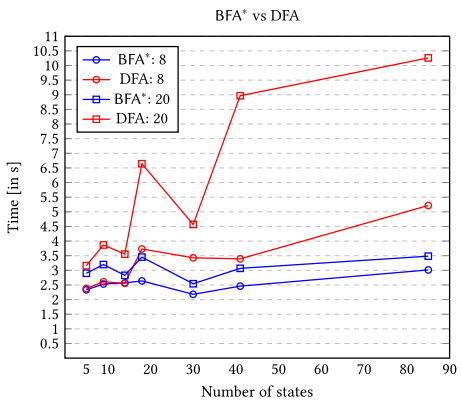
Comparing BFA and DFA analyses.* The comparison is presented in Figure 8(a) and 8(b):

— Figure 8(a) 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 5.7×.

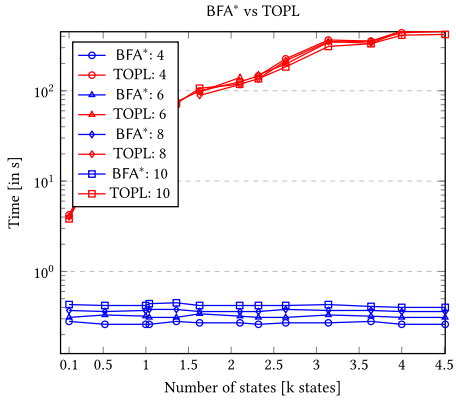
— Figure 8(b) compares various class compositions for client programs of 15K LoC. Both implementations fail to meet the SLA; however, the BFA* is close and exhibits constant behavior regardless of the number of states in the contract. The DFA implementation is rather erratic, tending to sharply increase in execution-time as the number of states increases. Overall, BFA* produces a geometric mean speedup over DFA of 1.5×. We note that *must* contracts do not have noticeable performance differences from *may* contracts.



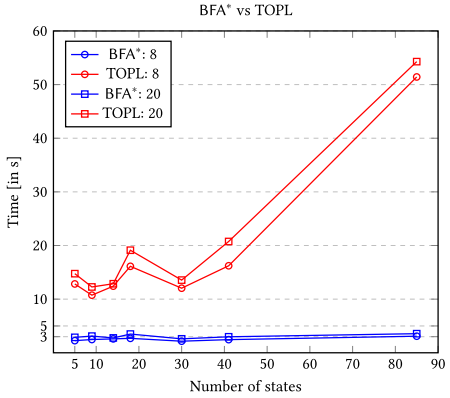
(a) DFA vs BFA* comparison on composed contracts (500-1k LoC)



(b) DFA vs BFA* 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. 8. Performance evaluation. Each line represents a different number of base classes composed in a client code.

Comparing BFA-based analysis vs TOPL typestate implementation (Execution time).* Here again, client programs do not require aliasing. The comparison is presented in Figure 8(c) and 8(d):

— Figure 8(c) 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×.

— Figure 8(d) 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 301.65×.

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. Again, we note that *must* contracts do not have noticeable performance differences from *may* contracts.

6 RELATED WORK

We focus on comparisons with restricted forms of typestate contracts. We refer to the typestate literature [7, 9, 10, 17, 23] for a more general treatment. The work [15] proposes a restricted form of typestates, tailored to use-cases of 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, [12] 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 [11] 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 [9] 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. The annotations in Fugue can specify "must call" properties (e.g., "must call a release method"). In this version of our article, we propose BFA extended with must logic that can express similar contracts. JaTyC [19] is a recent tool that supports Java inheritance. Our formalism can also handle inheritance, which we discuss in this article as BFA subsumption (cf. Section 2).

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 Section 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 [21].

7 CONCLUDING REMARKS

In this article, 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. They allow for succinct annotations that can describe a range of may and must call contracts; on the other hand, they exhibit more scalable performance compared to DFA based approaches. We have implemented our typestate analysis in the industrial-strength static analyzer Infer, which is publicly available and open source.

Future Work. There are several interesting research directions for the future work. First, it is worth investigating how BFA and DFA-based analyses can be bundled into a single analysis, thus inhering the benefits of both. Furthermore, we plan to integrate aliasing in our approach, leveraging the fact that Infer already comes with aliasing checkers. This would enable an investigation to verify our conjecture that our BFA-based analysis performance gains will be preserved, or perhaps more prominently displayed, in the presence of aliasing information.

Moreover, it would be interesting to explore whether our BFA formalism can be effectively used in settings where BFA-based methods are typically used, such as, for example, automata learning, code synthesis, and automatic program repair. Finally, understanding the usability gains of moving from DFAs to BFAs is definitely interesting and it deserves a separate user study investigation.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers for their constructive remarks.

REFERENCES

[1] 2021. Infer TOPL. (2021). Retrieved from https://fbinfer.com/docs/checker-topl/

[2] 2021. RAIL model. (2021). Retrieved from https://web.dev/rail/ Accessed: 2021-09-30.

[3] Alen Arslanagic, Pavle Subotic, and Jorge A. Pérez. 2022. Scalable typestate analysis for low-latency environments. In *Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings (Lecture Notes in Computer Science)*, Maurice H. ter Beek and Rosemary Monahan (Eds.), Vol. 13274. Springer, 322–340. DOI: https://doi.org/10.1007/978-3-031-07727-2_18

[4] Alen Arslanagić, Pavle Subotić, and Jorge A. Pérez. 2022. LFA checker: Scalable typestate analysis for low-latency environments. (Mar 2022). DOI: https://doi.org/10.5281/zenodo.6393183

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices* 49, 6 (06 2014), 259–269. DOI: https://doi.org/10.1145/2594291.2594299

[6] Kevin Bierhoff and Jonathan Aldrich. 2007. Modular typestate checking of aliased objects. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*. Association for Computing Machinery, New York, NY, 301–320. DOI: https://doi.org/10.1145/1297027.1297050

[7] Eric Bodden and Laurie Hendren. 2012. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer* 14, 3 (jun 2012), 307–326.

[8] Cristiano Calcagno and Dino Distefano. 2011. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, Berlin, 459–465.

[9] Robert Deline and Manuel Fähndrich. 2004. *The Fugue Protocol Checker: Is Your Software Baroque?* Technical Report MSR-TR-2004-07. Microsoft Research.

[10] Robert DeLine and Manuel Fähndrich. 2004. Typestates for objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings (Lecture Notes in Computer Science)*, Martin Odersky (Ed.), Vol. 3086. Springer, 465–490. DOI: https://doi.org/10.1007/978-3-540-24851-4_21

[11] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2021. 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 (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, 1416–1426. DOI: https://doi.org/10.1145/3468264.3473934

[12] Manuel Fahndrich and Rustan Leino. 2003. Heap monotonic typestate. In *Proceedings of the 1st International Workshop on Alias Confinement and Ownership (IWACO)* (proceedings of the first international workshop on alias confinement and ownership (iwaco) ed.). Retrieved from https://www.microsoft.com/en-us/research/publication/heap-monotonic-typestate/

[13] Manuel Fähndrich and Francesco Logozzo. 2010. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software (FoVeOOS'10)*. Springer-Verlag, Berlin, 10–30.

[14] Mathias Jakobsen, Alice Ravier, and Ornela Dardha. 2021. Papaya: Global typestate analysis of aliased objects. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP'21)*. Association for Computing Machinery, New York, NY, Article 19, 13 pages. DOI: https://doi.org/10.1145/3479394.3479414

[15] Martin Kellogg, Manli Ran, Manu Sridharan, Martin Schäf, and Michael D. Ernst. 2020. Verifying object construction. In *ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering*. Seoul, Korea.

[16] U. Khedker, A. Sanyal, and B. Sathe. 2017. *Data Flow Analysis: Theory and Practice*. CRC Press. Retrieved from https://books.google.rs/books?id=9PyrtgNBdg0C

[17] Patrick Lam, Viktor Kuncak, and Martin Rinard. 2004. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Not.* 39, 3 (March 2004), 46–55. DOI: https://doi.org/10.1145/981009.981016

[18] Johannes Lerch, Johannes Späth, Eric Bodden, and Mira Mezini. 2015. Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE Press, 619–629. DOI: https://doi.org/10.1109/ASE.2015.9

[19] João Mota, Marco Giunti, and António Ravara. 2021. Java typestate checker. In *Coordination Models and Languages*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer International Publishing, Cham, 121–133.

[20] Rajshakhar Paul, Asif Kamal Turzo, and Amiangshu Bosu. 2021. Why security defects go unnoticed during code reviews? A case-control study of the chromium OS project. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1373–1385. DOI: https://doi.org/10.1109/ICSE43902.2021.00124

[21] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. Association for Computing Machinery, New York, NY, 49–61. DOI: https://doi.org/10.1145/199448.199462

[22] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 48 (jan 2019), 29 pages. DOI: https://doi.org/10.1145/3290361

[23] Robert E. Strom and Shaula Yemini. 1986. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering* 12, 1 (1986), 157–171. DOI: https://doi.org/10.1109/TSE.1986.6312929

[24] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A static analysis framework for data science notebooks. In *Proceedings of the 44th International Conference on Software Engineering*.

[25] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. Association for Computing Machinery, New York, NY, 320–331. DOI: https://doi.org/10.1145/2970276.2970298

[26] Eran Yahav and Stephen Fink. 2011. *The SAFE Experience*. Springer, Berlin, 17–33. DOI: https://doi.org/10.1007/978-3-642-19823-6_3

Received 14 October 2022; revised 15 March 2023; accepted 12 April 2023