Scalable Typestate Analysis using Bit-Vector Machines

Alen Arslanagić
University of Groningen, The Netherlands

Pavle Subotić
Microsoft, Serbia

Jorge A. Pérez
University of Groningen and CWI, The Netherlands

Abstract

Static analyses based on typestates are important in certifying correctness of industrial code contracts. At their heart, such analyses rely on finite-state machines (FSMs) to specify important properties of an object. Unfortunately, many useful contracts are impractical to encode as FSMs and/or their associated FSMs have a prohibitively large number of states, which leads to sub-par performance for low-latency environments. To address this bottleneck, we present a lightweight typestate analyzer, based on a specification language that can succinctly specify code contracts with significant expressivity. A central idea in our analysis is that using a class of FSMs that can be expressed and analyzed as bit-vectors can unlock substantial performance improvements. We validate this idea by implementing our lightweight typestate analyzer in the industrial-strength static analyzer Infer. We show how our lightweight approach exhibits considerable performance and usability benefits when compared to existing techniques, including industrial-scale static analyzers.

2012 ACM Subject Classification
Theory of computation → Program analysis; Software and its engineering → Integrated and visual development environments

Keywords and phrases
Static analysis, Code contracts, Typestate

1 Introduction

Industrial-scale software is generally composed of multiple interacting components. Typically, each component is produced by separate developers. As a result, software integration is a major source of bugs [23]. Many of these 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 have been proposed [13]; this is a particularly effective solution, 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 [24]. Typestate analysis is a technique used to enforce temporal code contracts. In object-oriented programs, where objects change state over time, typestates denote the valid sequences of method calls for a given object. The behavior of the object is prescribed by the collection of typestates, and each method call can potentially change the object’s typestate.

Given this, it is natural for static typestate checkers, such as FUGUE [9], SAFE [28], and Infer’s TOPL checker [3], to define the analysis property using finite-state machines, in particular 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 analyses based on typestates.
In this paper, we are interested in the problem of analyzing potentially large contracts in low-latency environments such as, e.g., Integrated Development Environments (IDEs). In such environments, to avoid noticeable disruptions in the users’ workflow, the analysis should take at most 1 second; ideally it should run under 500ms. However, the reliance on DFAs jeopardizes this goal, and it can result in scalability issues. Consider, e.g., a class with \( n \) methods in which each method enables another one and then disables itself. In this common scenario, the contract can result in a DFA with \( 2^n \) states. Given a large \( n \), such a contract is likely to result in sub-par performance and is cumbersome to specify manually.

Interestingly, a large number of practical contracts do not require the full expressivity of a DFA. In the enable/disable example above, the analysis of method dependencies is essentially local to a single state of the DFA: in enabling/disabling a method, only its associated state matters; the remaining \( 2^n - 1 \) states play no rôle. This suggests that using DFAs to express contracts that specify dependencies that are local to each method (or to a few methods) is redundant and/or prone to inefficient implementations. This observation triggers two intertwined questions: (i) how can we soundly restrict DFAs to obtain efficient and scalable program analyses?, and (ii) how can we succinctly specify such a restricted DFA?

To answer (i) and (ii), in this paper we develop a lightweight typestate analyzer, based on Lightweight Finite Automata (LFAs), a new sub-class of DFAs that can be expressed and analyzed using bit-vectors. Our work develops two key insights:

1. **LFAs suffice to express a wide range of code contracts.** We show that in many practical scenarios, LFAs suffice to capture information about the enabled and disabled methods at any given point. Because this information can be codified using bit-vectors, the analysis of properties can be performed efficiently; in particular, our analysis technique is not sensitive to the number of states in the associated LFA, which in turn ensures that our analysis scales well with contract and program size.

2. **Allowed and disallowed sequences of method calls for objects can be succinctly specified without resorting to DFAs.** To unburden the task of specifying typestates, we introduce LFA annotations to specify method dependencies as annotations on methods. LFA annotations can specify code contracts for usage scenarios commonly encountered when using libraries such as File, Stream, Socket, etc. in considerably fewer lines of code than DFAs.

We have implemented our LFA-based typestate analysis in the industrial-strength static analyzer Infer. Our analysis exhibits concrete usability and performance advantages and is adequately expressive to encode the vast majority of relevant typestate properties in the literature. On average, compared to state-of-the-art typestate analyses, our approach requires \( 273 \times \) less annotations and exhibits \( 45 \times \) analysis speedups.

To the best of our knowledge, ours is the first work to propose and validate in practice a specialization of DFA for typestate analysis. We envisage our LFA model is one of potentially many sub-classes of DFA for performing specialized computations (e.g., static analysis, compiler optimization, automata learning, etc.) that are traditionally implemented using DFA and yet result in scalability issues for large DFAs.

**Contributions and Organization** We summarise our contributions as follows:

- We introduce the LFA specification language, which is as expressive as an LFA, a new sub-class of DFA based on bit-vectors.
- We provide new lightweight analysis techniques based on LFAs, implemented in Infer.
- We validate our approach via extensive evaluations that demonstrate the gains in performance and usability due to our lightweight analysis technique.
class Transaction {} {
    void setId();
    void setAmount();
    void setDescription();
    // ...
    String getId();
    float getAmount();
    String getDescription();
}

Listing 1 Getter/Setter Example

class Converter {
    void setConversionRate(float);
    void setFeeRate(float);
    void setSaleTax(float);
    // ...
    float convert() {...};
    float applyFee() {...};
    float applyTax() {...};
}

Listing 2 Getter/Setter Example 2

Next, we further motivate the need for an LFA-based analysis (§2). In §3, we precisely define the LFA model. Then, we describe a concrete algorithm for implementing an LFA-based static analysis (§4). In §5, we evaluate an implementation of our LFA-based analysis in the Infer static analyzer, comparing it to several other typestate approaches. §6 outlines related work and §7 discusses the limitations of our approach. Finally, §8 concludes.

2 Motivation

We motivate our techniques by highlighting the differences between our approach and DFA-based typestate analyses.

In Listing 1, class Transaction represents a transaction with three getter and setter pairs. We want to enforce that a ‘get’ method is called only after its corresponding ‘set’ method. Therefore, an object of class Item can be in a state in which any subset of \{getId(), getAmount(), getDescription()\} is enabled. If we were to specify this contract with a DFA, this pattern would result in a DFA with \(2^3\) states, as shown in Fig. 1, where, for brevity, a self-loop with the \{Id\} label denotes on both setter and getter transitions (setId() and getId()).

In general, in classes with \(n\) pairs of setters and getters, the number of states in a corresponding DFA would be \(2^n\). We have found ample code and contracts that can result in DFA states numbering in approximately 30K states. The number of states impacts the DFA analysis, which has a complexity of \(O(n \times c)\), where \(n\) is the number of states and \(c\) is the number of calls to object methods. Industrial case studies such as [11, 19, 18], attest to similar experiences, and highlight the fact that the number of states can quickly become large for contracts. As we show in Section 5, such DFAs do not adequately scale for larger values of \(n\) for our use case both in terms of annotations and performance.

The state explosion problem apparent with DFAs is further exacerbated in the presence of
composition, a common scenario in real-world programs. Consider Listing 3 where the class `ProcessTransaction` uses instances of `Transaction` and `Converter` (given in Listing 2), which joins a transaction data and a conversion logic to process a row transaction. Let us assume the following client code for class `ProcessTransaction`:

```java
void useTransaction () {
    ProcessTransaction pt = new ProcessTransaction ();
    pt.setTransaction (...);
    pt.setRates (...);
    pt.getNetAmount ();
}
```

The allowed orderings of calls to methods of `ProcessTransaction` depend on the DFA specification of both `Transaction` and `Converter` classes. That is, the DFA for class `ProcessTransaction` is induced by those of `Transaction` and `Converter` as well as by implementations of methods using objects of these two classes. To check whether a call to method `getNetAmount` is valid after `setTransaction` and `setRates` (without inlining their code), we need to compute procedure summaries of these three methods.

Let us initially consider a procedure summary for a method with a single argument. Note, this definition also hold for a class member object, since the two are interchangeable. A procedure summary for a method, maps each each typestate from before the method was invoked to a typestate after the method was invoked. Since a method can contain branches, it could transition the typestate to different set of typestates. Thus, the summary computation for a method with a single argument whose DFA consists of n states has a worst-case complexity of $O(n^2 \times c)$, where c is the number of method calls. That is, for each method call and each input typestate, we need to transition n typestates in the worst case. Moreover, when composing summaries, each transition can give us a set of typestates: this requires a union operation in the number of typestates, so the worst-case complexity of composing summaries is $O(n^3 \times c)$. Since the summary has to be computed for each method
class ProcessTransaction {
    Transaction t; Converter c;
    // ...
    void setTransaction(float amount) { t.setAmount(amount); }
    void setRates(float convRate, float fee, float tax) {
        c.setConversionRate(convRate);
        c.setFee(fee).setTax(tax);
    }
    // ...
    float getNetAmount() {
        float net = c.convert(t.getAmount());
        net = c.applyFee(net);
        return c.applyTax(net);
    }
}

Listing 3 Composition Example

argument that has an associated DFA, the worst-case complexity of the DFA summary computation for methods with multiple arguments is $O(p \times n^3 \times c)$, where $p$ is a maximum number of arguments for a method and $n$ is a maximum number of typestates in a program DFA contract.

In contrast, our LFA-based analysis exploits the following fact: because dependencies are local to each method, it is possible to specify allowed and disallowed sequences of method calls for objects without explicitly specifying a DFA. Thus, LFA specifies method dependencies as annotations on methods. Furthermore, the nature of LFA allows the abstract domain to be implemented as a bit-vector, thus the complexity of the analysis remains constant, irrespective of how many states the equivalent DFA has and irrespective of compositions. Thus the LFA analysis exhibits a worst-case complexity of $O(p \times c)$. In the following sections we will provide a precise definition of LFA and our an LFA-based analysis is computed.

3 Lightweight Finite Automata Analysis

3.1 LFA Annotations

We introduce LFA specifications, which succinctly encode temporal properties by only describing local method dependencies, thus avoiding an explicit DFA specification. LFA specifications define code contracts by using atomic combinations of annotations ‘@Enable($n$)’ and ‘@Disable($n$)’, where $n$ is a set of method names. Intuitively, ‘@Enable($n$) $m$’ asserts that invoking method $m$ makes calling methods in $n$ valid in a continuation. Dually, ‘@Disable($n$) $m$’ asserts that a call to $m$ disables calls to all methods in $n$ in the continuation.

More concretely, we give semantics for LFA annotations by defining valid method sequences:

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

@Enable($E_i$) @Disable($D_i$) $m_i$

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

The formal semantics for these specification is given in §3.2. We note, if \( E_i \) or \( D_i \) is \( \emptyset \) then we omit the corresponding annotation. Moreover, the LFA language can be used to derive other useful annotations:

- ‘\(@\text{EnableOnly}(E_i)\ m_i\)’ asserts that a call to method \( m_i \) enables only calls to methods in \( E_i \) while disabling all other methods in \( C \). Thus, it is encoded as follows:

\[
@\text{EnableOnly}(E_i)\ m_i \equiv @\text{Enable}(E_i) \oplus @\text{Disable}(C \setminus E_i)\ m
\]

where \( C \) is a set of method identities of a class where \( m_i \) belongs to.

- ‘\(@\text{DisableOnly}(D_i)\ m_i\)’ is dual to ‘\(@\text{EnableOnly}\)’, and can be encoded as follows:

\[
@\text{DisableOnly}(D_i)\ m_i \equiv @\text{Disable}(D_i) \oplus @\text{Enable}(C \setminus E_i)\ m
\]

- ‘\(@\text{EnableAll} m_i\)’ asserts that a call to method \( m_i \) enables all methods in a class, and thus it is encoded as:

\[
@\text{EnableAll} m_i \equiv @\text{Enable}(C)\ m_i
\]

- ‘\(@\text{DisableAll} m_i\)’ is dual to ‘\(@\text{EnableAll} m_i\)’.

We illustrate the expressivity and usability of LFA annotations by means of an example. We consider the \texttt{SparseLU} class from Eigen C++ library\(^1\). For brevity, we consider representative methods for a typestate specification (we also omit return types):

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

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

We notice that there are allowed method call sequences that do not necessarily cause errors, but have redundant computations. For example, we can disallow consecutive calls to \texttt{compute} as in, e.g., sequences like ‘\texttt{compute().compute().solve()}’ as the result of first compute is never used. Furthermore, we know that \texttt{compute} is essentially implemented as follows:

\[
\texttt{compute()} = \texttt{analyzePattern().factorize()}
\]

Thus, for example, it is also redundant to call \texttt{factorize} after \texttt{compute}. The DFA that substitutes dynamic checks and avoids redundancies is given in Figure 2. Following the literature \([9]\), this DFA can be annotated inside a class definition as follows:

\[\text{https://eigen.tuxfamily.org/dox/classEigen_1_1SparseLU.html}\]
In this annotation, states are listed in the class header and transitions are specified as @Pre and @Post conditions on methods. However, this code contract specification is too low-level and unreasonable for software engineers to annotate their APIs with, due to high annotation overheads. In contrast, using LFA annotations the entire SparseLU class contract can be succinctly specified as follows:

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

Here, the starting state is unspecified. The starting state is determined by annotations on methods. In fact, methods that are not guarded by other methods (like solve is guarded by compute) are enabled in the starting state. We remark that this can be overloaded by...
specifying annotations on constructor method. We can see that we are able to specify the the contract with only 4 annotations, whereas the corresponding DFA requires 8 annotations and 4 states specified in the class header.

3.2 Lightweight Finite Automata

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

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

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

An LFA for a class with \( n \) methods considers states \( q_b \), where, following Def. 2, the bit-vector \( b_A \) in \( \mathcal{B}^n \) denotes the set \( A \subseteq \Sigma_c \) enabled at that point. We often write ‘\( b \)’ (and \( q_b \)) rather than ‘\( b_A \)’ (and ‘\( q_b \)’), for simplicity. As we will see, the intent is that if \( m_i \in b \) (resp. \( m_i \not\in b \)), then the \( i \)-th method is enabled (resp. disabled) in \( q_b \). Def. 3 will give a mapping from methods to triples of bit-vectors.

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

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

Definition 3 (Mapping \( \mathcal{L}_c \)). Given a class \( c \), we define \( \mathcal{L}_c \) as a mapping from methods to triples of subsets of \( \Sigma_c \): \( \mathcal{L}_c : \Sigma_c \to \mathcal{P}(\Sigma_c) \times \mathcal{P}(\Sigma_c) \times \mathcal{P}(\Sigma_c) \).

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

In an LFA, 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, \( k \in D_i \) if and only if \( m_i \) disables \( m_k \). A transition from \( q_b \) labeled by method \( m_i \) leads to state \( q_{b'} \), where \( b' \) is determined by \( \mathcal{L}_c \) using \( b \). Such a transition is defined only if a pre-condition for \( m_i \) is met in state \( q_b \), i.e., \( P \subseteq b \). In that case, \( b' = (b \cup E_i) \setminus D_i \).

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

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

- \( Q \) is a finite set of states \( q_{10^n-1}, q_b, q_{b'}, \ldots \), where \( b, b', \ldots \in \mathcal{B}^n \);
- \( q_{0^{10^n-1}} \) is the initial state;
- \( \Sigma_c = \{ m_1, \ldots, m_n \} \) is the alphabet (method identities);
\[ L_C \text{ is an LFA mapping (cf. Def. 3);} \]

\[ \delta : Q \times \Sigma_C \rightarrow Q \text{ is the transition function, where} \]

\[ \delta(q_0, m_i) = q_{b'} \text{ (with } b' = (b \cup E_i) \setminus D_i) \]

if \( P_i \subseteq b \), and is undefined otherwise.

\[ \text{Example 5 (Refined SparseLU).} \] We give the LFA derived from the code annotations in the refined SparseLU example (Listing 5). First, we associate an index to a method by taking its position in the alphabetically sorted list of method names:

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

Index 0 is reserved for the constructor. Its annotations are implicit: it enables methods that are not guarded by annotations on other methods (in this case, \( aP \) and \( \text{compute} \)). The mapping \( L_{\text{SparseLU}} \) is as follows:

\[ L_{\text{SparseLU}} = \{0 \mapsto (\{1, 2\}, \{}), 1 \mapsto (\{3\}, \{1, 2, 4\}, \{}), 2 \mapsto (\{4\}, \{1, 2, 3\}, \{}), 3 \mapsto (\{4\}, \{1, 2, 3\}, \{}), 4 \mapsto (\{1, 2, 3\}, \{}), \{4\})\} \]

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

\[ \delta(q_{1000}, 0 : \text{constructor}) = q_{1100} \]

\[ \delta(q_{1100}, 1 : aP) = q_{0010} \quad \delta(q_{1100}, 2 : \text{compute}) = q_{0010} \]

\[ \delta(q_{0010}, 3 : \text{factorize}) = q_{0001} \quad \delta(q_{0001}, 4 : \text{solve}) = q_{1111} \]

\[ \delta(q_{1111}, 1 : aP) = q_{0010} \quad \delta(q_{1111}, 2 : \text{compute}) = q_{0001} \]

\[ \delta(q_{1111}, 3 : \text{factorize}) = q_{0001} \quad \delta(q_{1111}, 4 : \text{solve}) = q_{1111} \]

\[ \textbf{3.3 The LFA for Procedure Summaries} \]

We extend LFAs to account for procedure summaries. Let \( m(p) \) be a procedure signature with \( p \) acting as an actual argument for \( m \), with associated LFA \( M_p \). We can model a summary for \( m \) as an LFA transition. Differently from the transition function \( \delta \) from Def. 4 a procedure can branch; it could lead \( p \) to more than one state for a given input state. We generalize \( \delta \) as follows:

\[ \delta : Q_p \times \bigcup_{c \in C} \Sigma_c \rightarrow \mathcal{P}(Q_p) \]

where \( Q_p \) is the state set for \( M_p \), and \( \Sigma_c \) is the alphabet (method identities) for class \( c \), as before. This way, given an input state and a procedure name, the transition function returns a set of states.

In an LFA, we can abstract a set of states by a single state, given by the intersection of all states in the set. More concretely, for \( P \subseteq Q \) all method call sequences that are accepted by every state in \( P \) are also accepted by the state that is the intersection of bits of states in the set (i.e. \( q_b \), where \( b = \bigcap_{c \in P} b_c \)). Theorem 3 formalizes this property of LFA. First we need an auxiliary definition: let us write \( \text{Cod}(\cdot) \) to denote the codomain of a mapping:

\[ \textbf{Definition 6} \quad \text{[\text{[} \}} \text{\text{\text{]}]}\).

Let \( (E, D, P) \in \text{Cod}(\mathcal{L}_c) \) and \( b \in B^n \). We define \((E, D, P)](b) = b'\) where \( b' = (b \cup E) \setminus D \) if \( P \subseteq b \), and is undefined otherwise.
Theorem 1 (LFA States Union Property). Let $M = (Q, \Sigma_c, \delta, q_{10^{n-1}}, \mathcal{L}_c)$ and $P \subseteq Q$.
1. For $m \in \Sigma$ we have $\delta(q_b, m)$ is defined for all $q_b \in P$ iff $\delta(q_b, m)$ is defined, where $b = \bigcap_{q_b \in P} b$.
2. Let $\sigma = \mathcal{L}_c(m)$. If $P' = \{ \delta(q_b, m) : q_b \in P \}$ and $b_* = \bigcap_{q_b \in P'} b$ then $\bigcap_{q_b \in P'} b = [\sigma](b_*)$.

Proof. Item 1. is shown by the definition of $\delta(\cdot)$ (Def. 4). Item 2. is shown by induction on cardinality of $P$ using Def. 4. We remark that by Def. 6 and Def. 4 for $q \in Q$ and $m \in \Sigma_c$ we have $\delta(q_b, m) = q_b$ if $[\mathcal{L}_c(m)](b) = b'$. See App. A for details.

We introduce the relation $\vdash^m$ to capture transitions of states intersections as follows:

$$q' = \bigcap \delta(q, m)$$

where we write $q_A \cap q_B$ the operation on bit-vectors, i.e., $b_A \cap b_B$ (cf. Def. 2).

This idea generalizes easily to consider method calls with more than one argument, i.e., $m(p_1, \ldots, p_n)$, where we assume that each $p_i$ has a corresponding LFA $M_i$. To account for this, we extend $\vdash^m$ with state configurations. A state configuration is $n$-tuple of type $Q_1 \times \ldots \times Q_n$ where $Q_i$ is the state set of $M_i$. For given method $m$ the summary is of the following type:

$$\vdash^m : Q_1 \times \ldots \times Q_n \rightarrow Q_1 \times \ldots \times Q_n$$

We can break down this relation element-wise as summaries for arguments are independent as follows:

$$\prod_{i \in \{1, \ldots, n\}} q_i \vdash^m q_i'$$

$$\vdash^m (q_1', \ldots, q_n')$$

3.4 LFA subsumption

Since LFA are tied to classes, the class inheritance imposes a question on how we check that subclass LFA is a proper refinement of its superclass LFA. Proper refinement should mean that any valid sequence of calls to superclass methods must also be valid for its subclass. In other words, subclass’ LFA must subsume its parent’s LFA. Using LFAs, we can verify this by simply checking annotations method-wise. Let $M_1$ and $M_2$ be LFAs for classes $c_1$ and $c_2$. We can check whether $M_2$ subsumes $M_1$ only by considering their respective annotation mappings $\mathcal{L}_{c_2}$ and $\mathcal{L}_{c_1}$. Then, we have

$$M_2 \succeq M_1$$

iff for all $m_j \in \mathcal{L}_{c_1}$ we have $E_1 \subseteq E_2$, $D_1 \supseteq D_2$, and $P_1 \subseteq P_2$ where $(E_i, D_i, P_i) = \mathcal{L}_{c_i}(m_j)$ for $i \in \{1, 2\}$. Thus, we see that LFA subsumption boils down to set inclusion. The worst case complexity is $O(n \times n)$ where $n$ is a number of methods in the parent class, (i.e. $|\mathcal{L}_1|$).

4 Compositional Analysis Algorithm

We present an algorithm for compositional analysis based on LFAs. Given in the style of an Abstract Interpretation framework, the analysis algorithm interprets programs in an abstract domain exploiting a join operator on elements of the domain and a transfer function.
We formally define our analysis, which presupposes the control-flow graph (CFG) of a program. Let us write \( V \) to denote the set of program variables. The abstract domain for analysis, denoted \( D \), is defined similarly as the mapping \( L_c \) (Def. 3). The difference is that \( D \) maps program variables \( V \) rather than method identities:

\[
D : V \rightarrow \bigcup_{c \in C} \text{Cod}(L_c)
\]

As variables in \( V \) can be of any declared class \( c \in C \), which in turn has an associated LFA (given by \( L_c \)), the co-domain of \( D \) is the union of codomains of \( L_c \) for all classes in a program.

**Definition 8 (Join Operator).** We define \( \sqcup : \text{Cod}(L_c) \times \text{Cod}(L_c) \rightarrow \text{Cod}(L_c) \) as follows:

\[
(E_1, D_1, P_1) \sqcup (E_2, D_2, P_2) = (E_1 \cap E_2 \setminus (D_1 \cup D_2), D_1 \cup D_2, P_1 \cup P_2)
\]

The join operator on \( \text{Cod}(L_c) \) is lifted to \( D \) by taking the union of un-matched entries in the mapping: if a key \( a \) is defined in both mappings then \( \sqcup \) is applied to its value; otherwise, we take the union of entries with keys that are present in only one map.

We remark that \( D \) is sufficient for both analysis and summary computation, as we will show in the remaining of the section.

### 4.1 Algorithm

The algorithm for compositional analysis is given in Alg. 1. It expects a program’s CFG and a series of contracts, expressed as LFAs annotation mappings (Def. 3). If the program violates the LFA contracts, a warning is raised. The algorithm traverses the CFG nodes in a forward manner. For each node in the CFG, it first collects information from its predecessors (denoted by \( \text{pred}(v) \)) and joins them as \( \sigma \) (line 3). Then, the algorithm checks whether a method can be called in the given abstract state \( \sigma \) by calling predicate \( \text{guard()} \) (cf. Alg. 2). If the pre-condition is met, then the \( \text{transfer()} \) function (cf. Alg. 3) is called on a CFG node.

We assume a collection of LFA contracts (given as \( L_{c_1}, \ldots, L_{c_k} \)), which is input for Alg. 1, is accessible in Alg. 3 to avoid explicit passing.

First, we define some useful functions and predicates. For the algorithm, we require that the constructor method’s disabling set is the complement of the enabling set:

**Definition 9 (Well-formed \( L_c \)).** Let \( c \) be a class, \( \Sigma \) set method identities of class \( c \), and \( L_c \). Then, predicate \( \text{well-formed}(L_c) \) holds iff \( D = \Sigma \setminus E \) with \( L_c(\text{constructor}) = \langle E, D, P \rangle \).

**Definition 10 (\( \text{warning}() \)).** Let \( G \) be a CFG and \( L_1, \ldots, L_k \) be a collection of LFAs. We define

\[
\text{warning}(G, L_1, \ldots, L_k) = \text{true}
\]

if there is a path in \( G \) that violates some of \( L_i \) for \( i \in \{1, \ldots, k\} \).

**Definition 11 (\( \text{exit}_\text{-node}() \)).** Let \( v \) be a method call node. Then, \( \text{exit}_\text{-node}(v) \) denotes exit node \( w \) of a method body corresponding to \( v \).

#### 4.1.1 Guard Predicate

Predicate \( \text{guard}(v, \sigma) \) checks whether a pre-condition for method call node \( v \) in the abstract state \( \sigma \) is met (cf. Alg. 2). For convenience we represent a call node as follows:

\[
\text{Call} = \text{node}[m_j(p_0:b_0, \ldots, p_n:b_n)]
\]
Algorithm 1 LFA Compositional Analysis
Data: G: A program’s CFG, a collection of LFA mappings: \( \mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k} \) over classes \( c_1, \ldots, c_k \) such that \( \text{well}_{\text{formed}}(\mathcal{L}_{c_i}) \) for \( i \in \{1, \ldots, k\} \)
Result: \( \text{warning}(G, \mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k}) \)
1 Initialize \( \text{NodeMap}: \text{Node} \rightarrow D \) as an empty map;
2 foreach \( v \) in \( \text{forward}(G) \) do
3 \( \sigma = \bigsqcup_{w \in \text{pred}(v)} w; \)
4 if \( \text{guard}(v, \sigma) \) then
5 \( \text{NodeMap}[v] := \text{transfer}(v, \sigma); \)
6 else
7 \( \text{return True} \)
8 end
9 end
10 \text{return False}

where \( p_i \) are formal and \( b_i \) are actual parameters (for \( i \in \{0, \ldots, n\} \)). Let \( \sigma_w \) be a post-state of an exit node of method \( m_j \). The pre-condition is met if for all \( b_i \) there are no elements in their pre-condition set (i.e., the third element of \( \sigma[w][b_i] \)) that are also in disabling set of the current abstract state \( \sigma \). This way, we uniformly check if there is an error for code checking and for computing summaries. That is, for code checking we could have checked if a pre-condition is a subset of the enabling set of \( \sigma \), but this would not be a correct guard for the procedure summary computation. In this case, method calls inside a procedure body do not have to be enabled within the procedure as this can be satisfied at a program point where a procedure is called. On the other hand, calling methods which are previously disabled within the procedure (i.e. methods in the disabling set of \( \sigma \)) is an error. Moreover, for code checking we need the property that \( D = \Sigma_{c_i} \setminus E \) for all variables in abstract state, where \( \Sigma_{c_i} \) is a set of method identities for class \( c_i \), in order for \( \text{guard}() \) to correctly detect an error. This is ensured by condition \( \text{well}_{\text{formed}}(\mathcal{L}_{c_i}) \) (Def. 9) and definition of \( \text{transfer}() \) (see below).

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

**Entry-node:** (lines 3-11) This is a function entry node. 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 that the first formal argument \( p_0 \) is a reference to the receiver object (i.e., \( \text{this} \)). We initialize the domain based on the user-supplied LFA (annotations). If method \( m_j \) is defined in class \( c_i \) (with \( \mathcal{L}_{c_i} \)), we initialize the domain to the singleton map, where \( \text{this} \) is mapped to the corresponding annotations in \( \mathcal{L}_{c_i} \). The pre-condition in this case is simply the singleton \( \{m_j\} \). On the other hand, if there is no user-supplied annotations for class \( C_i \), we return an empty map meaning that a summary has to be computed. Here we remark that although the remaining arguments \( p_1, \ldots, p_n \) could have their own LFAs, for simplicity we assume that user-supplied annotations for \( m_j \) override them.

**Call-node:** (lines 12-25) This is a method call node. As in \( \text{guard}() \), we represent it as \( m_j(p_0 : b_0, \ldots, p_n : b_n) \) where \( b_0, \ldots, b_n \) are actual arguments. We assume arguments \( b_0, \ldots, b_n \) are values that are references to objects. This is the most interesting case as it is the crux of the analysis. We first check if \( \text{this} \) is in the domain (line 15). This way we skip over analysis if we are analyzing a method which has user-supplied annotations. If
Algorithm 2  Guard Predicate

Data: $v$: CFG node, $\sigma$: Domain
Result: False iff $v$ is a method call that cannot be called in $\sigma$

Procedure $guard$ ($v, \sigma$)

switch $v$ do
  case $Call-node[m_j(p_0 : b_0, \ldots, p_n : b_n)]$ do
    Let $w = \text{exit\_node}(v)$;
    for $i \in \{0, \ldots, n\}$ do
      if $\sigma_w[p_i].P \cap \sigma[b_i].D \neq \emptyset$ then return False;
    end
    return True
  end
  otherwise do
    return True
  end
end

there are no user-supplied annotations for a method under analysis, we compute a summary for each formal argument. In line 17, we first gather annotations for an argument as a summary which as a post-state of the function exit node (i.e., $\sigma_w$). Here, we can see that we uniformly treat computed summaries and user-supplied annotations. In lines 18 and 19 we appropriately accumulate enabling and disabling sets. The resulting enabling set is obtained by (i) adding methods that $m_j$ enables ($E^m_i$) to the current enabling set $E_i$, and (ii) subtracting methods that $m_j$ disables ($D^m_i$), from it. Similarly, the resulting disabling set is obtained by (i) taking the union of the current disabling set $D_i$ and the set of methods that $m_j$ disables ($D^m_i$), and (ii) subtracting the methods that $m_j$ enables ($E^m_i$). Finally, the current set $P_i$ is expanded with elements of $P^m_i$ that are not in the enabling set $E_i$. We remark that property $D = \Sigma_{\alpha_i \setminus E}$ needed by $guard()$ for code checking is first established by interpreting the call to constructor method (by condition $well\_formed(L_i)$) and then preserved by the definition of $E_i'$ and $D_i'$.

Transfer is the identity on $\sigma$ for all other types of CFG nodes.

We can see that for each method call we have constant number of bit-vector operations (lines 15-21) per argument. Thus, the worst-case complexity of the compositional algorithm is $O(p \times c)$ where $p$ is the maximum number of arguments per method and $c$ is a number of method calls in a program.

4.2 To LFA summary

Now we discuss how the results of Alg. 1 can be phrased in terms of relation $\vdash^m$ (Def. 7).

For a method $m(p_1, \ldots, p_n)$ the algorithm computes a summary of the following form:

$$\{ p_1 \mapsto (E_1, D_1, P_1), \ldots, p_n \mapsto (E_n, D_n, P_n) \}$$

We remark that by applying $(E, D, P)$ we immediately get a state that is the intersection of states of LFA corresponding to $p_i$. Thus, for each argument $p_i$ we can construct $\vdash^m$ (Def. 7) by using $(E_i, D_i, P_i)$ in the same manner as $\delta$ is defined in an LFA (Def. 4). That is,

$q_b \vdash^m q_{b'} \quad (\text{if } [(E_i, D_i, P_i)](b) = b')$
Algorithm 3 Transfer Function

Data: \( v \): CFG node, \( \sigma \): Domain
Result: Output abstract state \( \sigma' \): Domain

1. Procedure \( \text{transfer}(v, \sigma) \)

   switch \( v \) do
   
   case \( \text{Entry-node}[m_j(p_0, \ldots, p_n)] \) do
   
   // Initialize domain based on LFA contract;
   
   Let \( c_i \) be the class of method \( m_j(p_0, \ldots, p_n) \);
   
   if \( \text{There is } \mathcal{L}_c \) then
   
   return \( \{ \text{this } \mapsto \mathcal{L}_c(m_j) \} \)
   
   else
   
   return Empty map
   
   end
   
   end
   
   case \( \text{Call-node}[m_j(p_0 : b_0, \ldots, p_n : b_n)] \) do
   
   Let \( w = \text{exit\_node}(v) \);
   
   Initialize \( \sigma' := \sigma \);
   
   if \( \text{this not in } \sigma' \) then
   
   for \( i = 1 \rightarrow n \) do
   
   \( \langle E_i, D_i, P_i \rangle = \sigma[b_i]; \langle E^m_i, D^m_i, P^m_i \rangle = \sigma_w[p_i]; \)
   
   \( E'_i = (E_i \cup E^m_i) \setminus D^m_i; \)
   
   \( D'_i = (D_i \cup D^m_i) \setminus E^m_i; \)
   
   \( P'_i = P_i \cup (P^m_i \setminus E_i); \)
   
   \( \sigma'[b_i] = \langle E'_i, D'_i, P'_i \rangle; \)
   
   end
   
   end
   
   return \( \sigma' \)
   
   end
   
   otherwise do
   
   return \( \sigma \)
   
   end
   
end

The only difference is that in \( \delta(\cdot) \) tuples \( \langle E, D, P \rangle \) are stored in \( \mathcal{L}_c \) (user-entered annotations), while in this case we get it as a result of a summary computation. This shows how our technique is insensitive to the number of states.

4.3 Correctness

Our LFA-based algorithm (Alg. 1) works by interpreting method call sequences in the abstract state and joins them appropriately (using join from Def. 8) following the control-flow of the program. Thus, we can prove its correctness by separately establishing (1) correctness of the interpretation of method sequences using a declarative representation of the algorithm (Def. 12) and (2) soundness of join operator (Def. 8). Furthermore, the algorithm works for a collection of LFA contracts of different classes and analyze method calls for all program variables. For the sake of proof we can isolate a method calls to a single program variable,
as checking method call sequences for different variables are independent of each other: in Alg. 3 we can see that when interpreting \(m_j(p_0 : b_0, \ldots, p_n : b_n)\), each variable \(b_i\) is updated separately in the abstract state (on line 21) using only information from its corresponding pre-state (line 17).

Thus, we define the declarative transfer function as follows:

\[ \text{dtransfer}_c(m, \langle E, D, P \rangle) = \langle E', D', P' \rangle \]

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, then

\[
\text{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle E, D, P \rangle) = \text{dtransfer}_c(m_{n+1}, \text{dtransfer}_c(m_1, \ldots, m_n, \langle E, D, P \rangle))
\]

By Thm. 2 we have \(\bigcup_{i \in I} \{q_{i}\} = q_{\emptyset}\) where \(b' = \bigcap_{i \in I} b_i\) for some index set \(I\). So, the statement for the soundness of the join operator is as follows:

**Theorem 2 (Soundness of Join Operator).** Let \(q_b \in Q\) and \(\phi_i = \langle E_i, D_i, P_i \rangle\) for \(i \in \{1, 2\}\). Then, \(\llbracket \phi_1 \rrbracket(b) \cap \llbracket \phi_2 \rrbracket(b) = \llbracket \phi_1 \cup \phi_2 \rrbracket(b)\).

**Proof.** By definitions Def. 8 and Def. 9 and set laws. See App. A for details.

With auxiliary notions in place, we present the theorem that shows the correctness of summary computation and it can be specialized for the correctness of code checking:

**Theorem 3 (Correctness of Declarative Transfer).** Let \(M = (Q, \Sigma, \delta, q_{10^{n-1}}, \mathcal{L}_c)\). Let \(q_b \in Q\) and \(\overline{m} = m_1, \ldots, m_n\) be a method call sequence where \(m_i \in \Sigma\) for \(i \in \{1, \ldots, n\}\).

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

where \(b' = \llbracket \langle E', D', P' \rangle \rrbracket(b)\).

**Proof.** By induction on length of method call sequence. See App. A for details.

Now we discuss specialization of Thm. 3 for the code checking. In the case of code checking, we know that a method sequence must start with the constructor method (i.e., the sequence is of the form \(\text{constr}, m_1, \ldots, m_n\)) and that only the input state can be \(q_{10^{n-1}}\). By the algorithm’s requirement \textit{well\_formed} (Def. 8) we know that if \(\hat{\delta}(q_{10^{n-1}}, \text{constr}) = q_b\) and \(\text{dtransfer}_c(\text{constr}, m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \sigma\) then all methods that are not enabled in \(q_b\) will be in the disabling set of the abstract state \(\sigma\). So for any \(m_1, \ldots, m_{k-1}, m_k\) in the sequence that is disabled by the constructor and not enabled in substring \(m_1, \ldots, m_{k-1}\), the condition \(P \cap D_i \neq \emptyset\) will correctly check that method is disabled. If \textit{well\_formed} (Def. 8) did not hold, the algorithm would fail to detect an error as it would put \(m_k\) in \(P\) set since \(m_k \notin E\).
5 Evaluation

We evaluate our LFA analysis on a set of code benchmarks and code contracts. The goal of our evaluation is to validate the following two claims:

Claim-I: Smaller annotation overhead. The LFA contract annotation overheads are significantly smaller in terms of lines of code (LoC) than both competing analyses.

Claim-II: Improved scalability on large code and contracts. Our analysis scales better than the competing analyzers on two dimensions, namely, caller code size and contract size, allowing us to check more contracts given the service level agreement (SLA) of 1 second.

5.1 Use Case and SLA

Our use case is to integrate static analyses in interactive IDEs e.g., Microsoft Visual Studio Code, Jupyter Notebooks [26], so that code can be analyzed at coding time. For this reason, our use case requires low latency execution of the static analysis. Our SLA is based on the RAIL user-centric performance model [1]. We aim for our analysis to be less than 1 second long and ideally under 500ms. Any delays beyond 1 second result in noticeable disruption in the users workflow and delays above 10 seconds result in user frustration, causing them to likely abandon tasks. Moreover, we want to minimize the annotation overhead required by users. We aim to have an annotation overhead of less than 2 annotations per method.

5.2 Experimental Setup

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

Implementations Under Comparison

We implement two analyses, namely, LFA and DFA in the Infer static analyzer and use the default Infer typestate analysis TOPL as a baseline comparison. More in details:

- LFA: The Infer implementation of the technique described in this paper.
- DFA: A lightweight DFA-based typestate implementation based on an optimized DFA-based analysis developed previously specifically for our low-latency use case. Implemented in Infer, DFA is our previous attempt to scale typestate analysis for our use case. We translate LFA annotations to a minimal DFA and perform the analysis.
- TOPL: An industrial-strength typestate analyzer, implemented in Infer [3]. This typestate analysis is designed for high precision and not for low-latency environments. It uses Pulse, an Infer memory safety analysis, which provides it with alias information. We include it in our evaluation as a baseline state-of-the-art typestate analysis, i.e., an off-the-shelf industrial strength tool we could hypothetically use.

Benchmark Characteristics

We analyze a benchmark of 32 contracts that specify annotations for a class. Moreover, we auto-generate 228 client programs that vary in lines of code and number of composed classes. The annotations for LFA are manually specified and  

\[\text{removed due to double blind}\]

\[\text{removed due to double blind}\]
| Contract | #methods | #states | LFA LoC | DFA LoC | TOPL LoC |
|----------|----------|---------|---------|---------|----------|
| CR-1     | 3        | 2       | 3       | 5       | 9        |
| CR-2     | 3        | 3       | 5       | 5       | 14       |
| CR-3     | 3        | 4       | 4       | 7       | 25       |
| CR-4     | 5        | 5       | 5       | 10      | 24       |
| CR-5     | 5        | 4       | 6       | 10      | 30       |
| CR-6     | 5        | 10      | 11      | 31      | 83       |
| CR-7     | 5        | 14      | 9       | 36      | 116      |
| CR-8     | 7        | 15      | 12      | 85      | 213      |
| CR-9     | 7        | 30      | 10      | 120     | 323      |
| CR-10    | 7        | 41      | 12      | 157     | 460      |
| CR-11    | 9        | 80      | 15      | 425     | 1168     |
| CR-12    | 9        | 100     | 17      | 940     | 1884     |
| CR-13    | 11       | 156     | 20      | 1079    | 2828     |
| CR-14    | 11       | 292     | 20      | 1881    | 5108     |
| CR-15    | 13       | 403     | 24      | 2188    | 8032     |
| CR-16    | 15       | 522     | 19      | 4237    | 10423    |
| CR-17    | 15       | 663     | 28      | 5128    | 14307    |
| CR-18    | 15       | 845     | 29      | 7194    | 19857    |
| CR-19    | 15       | 991     | 26      | 8121    | 23113    |
| CR-20    | 15       | 1044    | 32      | 7766    | 20704    |
| CR-21    | 15       | 1628    | 21      | 13558   | 33740    |
| CR-22    | 15       | 2322    | 21      | 15529   | 47068    |
| CR-23    | 17       | 2044    | 24      | 26014   | 61346    |
| CR-24    | 19       | 3138    | 23      | 35345   | 85134    |
| CR-25    | 19       | 3638    | 23      | 39123   | 91120    |
| CR-26    | 19       | 4900    | 27      | 41092   | 101185   |
| CR-27    | 19       | 4588    | 27      | 46976   | 117828   |
| CR-28    | 19       | 5012    | 27      | 58206   | 133035   |
| CR-29    | 19       | 5531    | 27      | 55640   | 143227   |
| CR-30    | 21       | 6272    | 32      | 80264   | 179568   |
| CR-31    | 21       | 7626    | 32      | 101907  | 224187   |
| CR-32    | 21       | 7958    | 25      | 91347   | 223802   |

**Figure 3** Details of the 32 contracts considered in our evaluation, with LoC comparison. App. B gives details for CR-4.

From them, we generate minimal DFAs representation in DFA annotation format and TOPL annotation format.

### 5.3 Usability Evaluation

Figure 3 outlines the key features of the 32 contracts we considered, called CR-1–CR-32. To give an idea of these contracts, App. B gives the details for CR-4, one of the smallest contracts. For each contract, we specify the number of methods, the number of DFA states for the contract, and the lines of code for annotations in LFA, DFA and TOPL. 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 LFA remains constant wrt. state increase and increases rather proportionally with the number of methods in a contract. Observe that for contracts on classes with 4 or more methods, a manual specification using DFA or TOPL annotations becomes impractical. Overall, we validate Claim-I by the fact that LFA requires less annotation overhead on all of the contracts, on average requiring 273× less overheads than DFA and TOPL.
Scalable Typestate Analysis using Bit-Vector Machines

5.4 Performance Evaluation

We compare LFA against TOPL and DFA considering both execution time (Figures 4 and 5) and memory usage (Figures 6 and 7) on our client programs and the contracts from Figure 3. We discuss details of their evaluation.
(a) DFA vs LFA execution comparison: LoC from 100 to 1000

(b) DFA vs LFA execution comparison on composed contracts

Figure 5 Execution Time: DFA vs LFA. The dotted line (in red) denotes the 1s SLA.

**Execution Time** Figure 4 compares the execution time of our LFA-based analysis vs the TOPL typestate implementations. In Figure 4a, we compare single class contracts with varying clients (caller code) of different code sizes. Here we see that as the number of states increases in the contract, the TOPL execution-time increases accordingly. On the other hand, our LFA-based analysis remains constant with respect to the increase in state, only varying
Scalable Typestate Analysis using Bit-Vector Machines

(a) TOPL vs LFA memory comparison: LoC from 100 to 1000

(b) TOPL vs LFA memory comparison on composed contracts

Figure 6 Memory Usage: TOPL vs LFA.

with different client code sizes. In Figure 6b we compare LFA to TOPL for class compositions (4 to 10 classes). Composed contracts represent common scenarios, as typically functions may operate on several objects and classes contain several member objects, each with their own contracts. Here we see that as the number compositions increases the execution time of TOPL quickly increases, in the other hand, the LFA based analysis remains constant.
Compared to TOPL, LFA exhibits speedups on $273 \times$ on average. We conjecture TOPLs memory modeling likely accounts for some of the slowdown, even though the contracts and client programs do not require aliasing. Moreover, TOPL is unable to meet the SLA of our use cases, a testament to the need for alias-free analysis and the LFA technique.

Figure 5 compares our LFA analysis to a lightweight DFA analysis, designed specifically
for our use cases. In Figure 5a, we compare single class contracts with varying clients (caller code) of different code sizes. Here we see, as with TOPL, that for DFA execution-time increases as the number of states increase. Our LFA-based analysis remains constant. In Figure 5b, we compare LFA to DFA for class compositions. Here we see that as with TOPL, for DFA the number compositions increases the execution time while the LFA based analysis remains constant. Compared to DFA, LFA exhibits speedups of $4\times$ on average, allowing us to support large contracts within our SLA.

Overall, we validate Claim-II by showing that with an overall speedup of $45\times$, LFA can scale to contracts larger than 8K states on client programs over 1000 lines of code. Moreover it can scale to contracts with over 4K states with compositions over 10 classes (e.g., CR-26 to CR-32). This is a significant improvement on both DFA and TOPL in both categories.

### Memory Usage

In Figure 6 we compare the memory usage of our LFA-based analysis and TOPL. Our LFA-based analysis remains at approx. 35 MB of memory usage, regardless of number of lines of code or number of composed objects. On the other hand, TOPL increases with the number of states, peaking at approx. 370 MB. In Figure 7 we compare the memory usage of our LFA-based analysis and DFA. DFA scales poorly compared to the other approaches, peaking at approx 1.2 GB.

Overall, we further strengthen Claim-II as LFA uses under 50 MB of memory in all configurations. In contrast, DFA peaks at approx. 1.2 GB of memory and TOPL at approx. 400 MB of memory, which in some environments e.g., browsers, IDEs can contribute to degraded performance.

### 6 Related Work

Typestates were originally introduced to track value initialisation [25]; nowadays, there is a plethora of typestate techniques mainly focusing on object-oriented programs [21, 9]. These techniques provide the basis for analysis tools like Clara [7], Fugue [10], Infer [3] (TOPL), Plaid [16], SAFE [28], Checker [2] among many others. In such tools, DFAs are commonly used both as a specification language and as computational model [15, 22, 25, 17, 20, 6, 9, 10].

Restricted forms of typestates have recently been proposed to improve scalability. The work [18] proposes restricted form of typestates tailored for use-case of the object construction using the builder pattern. This approach has restrictions in that it only accumulates called methods in abstract state so it is monotonic and it does not require aliasing. 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 LFA and cannot express important practical properties in our use cases (e.g., the property “cannot call a method”).

The work [11] defines heap-monotonic typestates. This typestate analysis is monotonic and can be seen as a restriction. Moreover, it can also be performed without an alias analysis.

The recently proposed Rapid analyzer [11] aims to verify cloud-based APIs usage. It exploits the nature of clients code of these APIs for precise analysis during code-review or CI. 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 is the case in our technique. Their type-state approach is DFA-based and, interestingly, they describe state explosion problem for usual contracts found in practice, where the set of methods have to invoked prior to some event (i.e., where the ordering of calls to these methods is irrelevant). At high-level, their solution for state explosion is to factorize states in less granular control states (e.g., states that say that all methods are called, not enough, or none), which is suitable for
contracts for builder patterns they describe. On the other hand, our approach remedies the state-explosion problem by restricting DFA such that the set of states can be represented as a bit-vector. Therefore, we allow more granular contract specifications with a very large number of states while avoiding explicit specification of the DFA: thus we can specify the setter/getter example, which would not be possible using the RAPID approach.

The Fugue tool [10] allows DFA-based specifications, but also annotations for describing specific resource protocols contracts. Here special attributes are used to specify which methods create and releases resources for an object. These annotations have a notion of locality as annotations on one method do not refer to other methods. Moreover, we share the idea of specifying typestate without explicitly mentioning states. These additional annotations in Fugue are more expressive than DFA-based typestates (e.g., “must call a release method”). Using LFA we could only specify things like “nothing can be called after release” and “create must be called before release”.

7 Discussion on Limitations

The most obvious limitation of our LFA-based technique is that it assumes that enable/disable is performed with no context. Therefore, contracts that are conditional on context (e.g., abstract state) cannot be defined using LFA. An example of a contract that LFA cannot model can be found in [5] (cf. Figure 3). In this example, a contract enables or disables based on whether another method was previously called. While this limitation limits the contracts we can encode, it allows us to encode the abstract state as a bit-vector and gives us properties such as idempotence such that a call to a method can only alter the abstract state once and any immediate subsequent calls to the method do not change the abstract state. This helps us avoid issues with convergence in loops and recursion.

Concerning the current implementation of our LFA-based analysis, we do not perform alias analysis and our analysis is not path sensitive. This is due to the nature of our use case. We see no fundamental obstacle to support aliasing and path sensitivity in our approach. We believe that the integration of LFA with alias analysis can follow known approaches: (1) two-phase analysis by performing LFA analysis on top of results from the compositional aliasing analysis (for example, utilizing Infer’s Pulse checker) or (2) coupling alias analysis with LFA analysis. It is shown in [14] that two-phase analysis can lead to precision loss and that typestate analysis should be coupled with alias analysis. In both cases, instead of mapping a single program variable (i.e. \( p \in V \)), a subset of program variables (i.e. \( P \subseteq P(V) \)) has to be mapped to DFA state. Thus, aliasing should only impact the mapping of subset of program variables to states, and it should not impact the mechanics of our LFA approach. That is, at a method call \( p.foo() \) instead of only transitioning the typestate of \( p \), we would need to make a typestate transition on alias sets for which \( p \) belongs to. We conjecture that the performance improvements resulting from LFA insensitivity to the number of contract states can only multiplied in the presence of aliasing due to the increase in the DFA states that require transitions on method calls.

8 Conclusion

In this paper, we have presented a novel lightweight typestate analysis based on LFAs, a sub-class of DFAs based on bit-vectors. We believe LFAs are a simple and effective abstraction, with substantial potential to be ported and adapted to other settings. Compared to DFA-based analyses, LFAs are able to scale both in terms of annotation overhead and analysis
Scalable Typestate Analysis using Bit-Vector Machines

performance for large scale contracts. We have implemented our technique as an open source analysis in the Infer static analyzer and have demonstrated both its usability and performance improvements compared to state-of-the-art typestate analyzers.

Our LFA-based approach raises several interesting avenues of future research. One area we are interested in investigating concerns the potential advantages that LFA may provide in other algorithms such as automata learning [4]. Other areas we would like to further explore include integrating alias analysis alongside LFA and experimenting with restrictions and additions to the expressiveness of LFA.

References

1. Rail model. https://web.dev/rail/ Accessed: 2021-09-30.
2. Checker framework. https://github.com/typetools/checker-framework/ 2021.
3. Infer topl. https://fbinfer.com/docs/checker-topl/ 2021.
4. Dana Angluin. Learning regular sets from queries and counterexamples. Information and Computation, 75(2):87–106, 1987. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526 doi:https://doi.org/10.1016/0890-5401(87)90052-6
5. Kevin Bierhoff and Jonathan Aldrich. Modular typestate verification of aliased objects. page 51, 03 2007.
6. Eric Bodden. Efficient and precise typestate analysis by determining continuation-equivalent states, 2009.
7. Eric Bodden and Laurie Hendren. The clara framework for hybrid typestate analysis. Int. J. Softw. Tools Technol. Transf., 14(3):307–326, jun 2012.
8. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, NASA Formal Methods, pages 459–465, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
9. Robert DeLine and Manuel Fähndrich. Typestates for objects, In Martin Odersky, editor, ECOOP 2004 – Object-Oriented Programming, pages 465–490, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
10. Robert Deline and Manuel Fähndrich. The fugue protocol checker: Is your software baroque? 04 2004.
11. Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. Rapid: Checking api usage for the cloud in the cloud. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, page 1416–1426, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3473934
12. Manuel Fähndrich and Rustan Leino. Heap monotonic typestate. In Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO), July 2003. URL: https://www.microsoft.com/en-us/research/publication/heap-monotonic-typestate/
13. Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software, FoVeOOS'10, page 10–30, Berlin, Heidelberg, 2010. Springer-Verlag.
14. J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. Science of Computer Programming, 58(1):57–82, 2005. Special Issue on the Static Analysis Symposium 2003. URL: https://www.sciencedirect.com/science/article/pii/S0167642305000444 doi:https://doi.org/10.1016/j.scico.200502.004
15. Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol., 17(2), May 2008. doi:10.1145/1348250.1348255.
16 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4), oct 2014. doi:10.1145/2629609

17 Emmanuel Geay, Eran Yahav, and Stephen Fink. Continuous code-quality assurance with safe. pages 145–149, 01 2006.

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

19 Martin Kellogg, Narges Shadab, Manu Sridharan, and Michael D. Ernst. Lightweight and modular resource leak verification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 181–192, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3468264.3468576

20 Dimitrios Kouzapas, Ornella Dardha, Roly Perera, and Simon J. Gay. Typechecking protocols with mungo and stmungo. In *Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming*, PPDP ’16, page 146–159, New York, NY, USA, 2016. Association for Computing Machinery.

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

22 Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA ’12, page 997–1016, New York, NY, USA, 2012. Association for Computing Machinery.

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

24 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. doi:10.1109/TSE.1986.6312929

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

26 Pavle Subotić, Lazar Milikić, and Milan Stojić. A static analysis framework for data science notebooks, 2021. arXiv:2110.08339

27 Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, page 320–331, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2970276.2970298

28 Eran Yahav and Stephen Fink. *The SAFE Experience*, pages 17–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-19823-6_3
A Proofs

► **Theorem 1** (LFA States Union Property). Let \( M = (Q, \Sigma, \delta, q_{010^{-}}, L_c) \) and \( P \subseteq Q \).

1. For \( m \in \Sigma \) we have \( \delta(q_b, m) \) is defined for all \( q_b \in P \) iff \( \delta(q_b, m) \) is defined, where \( b_* = \bigcap_{q_b \in P} b \).
2. Let \( \sigma = L_c(m) \). If \( P' = \{ \delta(q_b, m) : q_b \in P \} \) and \( b_* = \bigcap_{q_b \in P} b \) then \( \bigcap_{q_b \in P} b = \| \sigma \|(b_*) \).

**Proof.** We show two items:

1. By Def. 4 for all \( q_b \in P \) we know \( \delta(q_b, m) \) is defined when \( P \subseteq b \) with \( \langle E, P, D \rangle = L_c(m) \).

So, we have \( P \subseteq \bigcap_{q_b \in P} b = b_* \) and \( \delta(q_b, m) \) is defined.

2. By induction on \( |P| \).
   - \( |P| = 1 \). Follows immediately as \( \bigcap_{q_b \in \{ q_b \}} q_b = q_b \).
   - \( |P| > 1 \). Let \( P = P_0 \cup \{ q_b \} \). Let \( |P_0| = n \). By IH we know
     \[
     \bigcap_{q_b \in P_0} [\| \sigma \|(b) = [\| \sigma \|(b)]_{q_b \in P_0} \] (1)

We should show
\[
\bigcap_{q_b \in (P_0 \cup \{ q_b \})} [\| \sigma \|(b) = [\| \sigma \|(b)]_{q_b \in (P_0 \cup \{ q_b \})}
\]

We have
\[
\bigcap_{q_b \in (P_0 \cup \{ q_b \})} [\| \sigma \|(b) = \bigcap_{q_b \in P_0} [\| \sigma \|(b) \cap [\| \sigma \|(b')]
\]
\[
= \bigcap_{q_b \in P_0} [\| \sigma \|(b) \cap [\| \sigma \|(b')]
\]
\[
= \bigcap_{q_b \in P_0} [\| \sigma \|(b) \cap [\| \sigma \|(b')]
\]
 \[
= ((b_* \cap E) \setminus D) \cap ((b' \cap E) \setminus D)
\]
\[
= ((b_* \cap E) \setminus D) \cap (b' \cap E) \setminus D
\]
\[
= [\| \sigma \|(b) \cap [\| \sigma \|(b')]_{q_b \in (P_0 \cup \{ q_b \})}
\]

where \( b_* = \| \sigma \|(\bigcap_{q_b \in P} b) \). This concludes the proof.

► **Theorem 2** (Soundness of Join Operator). Let \( q_b \in Q \) and \( \phi_i = \langle E_i, D_i, P_i \rangle \) for \( i \in \{1, 2\} \). Then, \( \| \phi_1 \|(b) \cap [\| \phi_2 \|(b) = \| \phi_1 \cup \phi_2 \|(b) \).

**Proof.** By set laws we have:
\[
\| \phi_1 \|(b) \cap [\| \phi_2 \|(b) = ((b \cup E_1) \setminus D_1) \cap ((b \cup E_2) \setminus D_2)
\]
\[
= ((b \cup E_1) \cap (b \cup E_2)) \setminus (D_1 \cup D_2)
\]
\[
= (b \cup (E_1 \cup E_2)) \setminus (D_1 \cup D_2)
\]
\[
= \| \phi_1 \cup \phi_2 \|(b)
\]

This concludes the proof.

► **Theorem 3** (Correctness of Declarative Transfer). Let \( M = (Q, \Sigma, \delta, q_{010^{-}}, L_c) \). Let \( q_b \in Q \) and \( \tilde{m} = m_1, \ldots, m_n \) be a method call sequence where \( m_i \in \Sigma \) for \( i \in \{1, \ldots, n\} \).

\( \text{dtransfer}_{\Sigma}(m_1, \ldots, m_n, \{0, 0, 0, \}) = \langle E', D', P' \rangle \iff \delta(q_b, m_1, \ldots, m_n) = q_b' \)

where \( b' = \| \langle E', D', P' \rangle \|(b) \).
Proof. \( \implies \) Soundness: By induction on the length of method sequence \( \bar{m} = m_1, \ldots, m_n \).

Case \( n = 1 \). In this case we have \( \bar{m} = m_1 \). Let \( \langle E^m, D^m, \{m_1\} \rangle = \mathcal{L}_c(m_1) \). By Def. \ref{def:transf} 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 Def. \ref{def:delta} we have \( \delta(q_b, m_1, \ldots, m_n) = q_{b'} \).

Case \( n > 1 \). Let \( \bar{m} = m_1, \ldots, m_n, m_{n+1} \). By IH we know

\[
d transfer_c(m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E', D', P' \rangle \Rightarrow \delta(q_b, m_1, \ldots, m_n) = q_{b'}
\] (2)

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

\[
\text{transfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E'', D'', P'' \rangle
\]

We should show

\[
\delta(q_b, m_1, \ldots, m_n, m_{n+1}) = q_{b''}
\] (3)

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 Def. \ref{def:transf} we have

\[
d transfer_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = d transfer_c(m_{n+1}, \langle E', D', P' \rangle)
\]

Further, we have

\[
E'' = (E' \cup E^m) \setminus D^m \quad D'' = (D' \cup D^m) \setminus E^m \quad P'' = P' \cup (P^m \setminus E')
\] (4)

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

\[
b'' = (b \cup E'') \setminus D'' = \begin{align*}
&= (b \cup ((E' \cup E^m) \setminus D^m)) \setminus ((D' \cup D^m) \setminus E^m) \\
&= ((b \cup E') \setminus (D' \setminus E^m)) \setminus D^m \\
&= ((b \cup E') \setminus D') \cup (E^m) \setminus D^m \\
&= (b' \cup E^m) \setminus D^m
\end{align*}
\]

Further, by \( P'' \subseteq b, P'' = P' \cup (P^m \setminus E'), \) and \( P^m \cap D' = \emptyset, \) we have \( P'' \subseteq (b \cup E') \setminus D' = b' \) (by \( \Delta \)). So, we can see that by definition of Def. \ref{def:delta} we have \( \delta(q_{b'}, m_{n+1}) = q_{b''} \). This concludes this case.

\( \Leftarrow \) Completeness:

\( n = 1 \). In this case \( \bar{m} = m_1 \). Let \( \langle E^m, D^m, \{m_1\} \rangle = \mathcal{L}_c(m_1) \). By Def. \ref{def:transf} we have \( b' = (b \cup E^m) \setminus D^m \) and \( \{m_1\} \subseteq b \). By Def. \ref{def:transf} we have \( E' = E^m, D' = D^m, P' = \{m_1\} \). Thus, as \( \{m_1\} \cap \emptyset = \emptyset \) we have \( b'' = \langle E', D', P' \rangle \). In this case we have \( \delta(q_b, m_1, \ldots, m_n) = \delta(q_{b''}, m_{n+1}) \).

\( n > 1 \). Let \( \bar{m} = m_1, \ldots, m_n, m_{n+1} \). By IH we know

\[
\delta(q_b, m_1, \ldots, m_n) = q_{b'} \Rightarrow d transfer_c(m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E', D', P' \rangle
\] (5)

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

\[
\delta(q_b, m_1, \ldots, m_n, m_{n+1}) = q_{b''}
\] (6)

We should show that

\[
d transfer_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E'', D'', P'' \rangle
\]
such that $b'' = (b \cup E'') \setminus D''$ and $P'' \subseteq b$. We know

$$d_{\text{transfer}}(m_1, \ldots, m_n, m_{n+1}, (\emptyset, \emptyset, \emptyset)) = d_{\text{transfer}}(m_{n+1}, (E', D', P'))$$

By Def. 4 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 (5) and (6) we have $\{m_{n+1}\} \subseteq b'$ and $b' = (b \cup E') \setminus D'$. It follows $\{m_{n+1}\} \cap D' = \emptyset$. That is, $d_{\text{transfer}}(m_{n+1}, (E', D', P'))$ is defined. Finally, showing that $b'' = (b \cup E'') \setminus D''$ is by the substitution and De Morgan’s laws as in the previous case. This concludes the proof.

$\Square$
## Sample Contract used in Evaluations (§5)

```java
class SparseLU {
    SparseLU();
    @EnableOnly(factorize)
    void analyzePattern(Mat a);
    void factorize(Mat a);
    @EnableOnly(solve, transpose)
    void compute(Mat a);
    @EnableAll
    void solve(Mat b);
    @Disable(transpose)
    void transpose();
}
```

**Listing 6** SparseLU LFA CR4 contract

```java
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)
    void solve(Mat b);
    @Pre(q2) @Post(q4)
    @Pre(q4) @Post(q3)
    void transpose();
}
```

**Listing 7** SparseLU DFA CR4 contract

**Figure 8** DFA diagram of SparseLU CR-4 contract
property SparseLU
prefix "SparseLU"

Listing 8 SparseLU TOPL CR4 contract