Scalable Typestate Analysis for Low-Latency Environments

Alen Arslanagić, Pavle Subotić, and Jorge A. Pérez

1 University of Groningen, The Netherlands
2 Microsoft, Serbia

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

1 Introduction

Industrial-scale software is generally composed of multiple interacting components, which are typically produced separately. As a result, software integration is a major source of bugs [17]. 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 [11], 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 [19]. 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 [8], SAFE [22], and Infer’s ToPL checker [2], to define the analysis property using Deterministic Finite Automata (DFAs). The abstract domain of the analysis is a set of states in the DFA; each operation on the object modifies the set of possible reachable states. If the set of abstract states contains an error state, then the analyzer warns the user that a code contract may be violated. Widely applicable and conceptually simple, DFAs are the de facto model in typestate analyses.
Here we target the analysis of realistic code contracts in low-latency environments such as, e.g., Integrated Development Environments (IDEs) [21][20]. In this context, to avoid noticeable disruptions in the users’ workflow, the analysis should ideally run under a second [1]. However, relying on DFAs jeopardizes this goal, as it can lead to scalability issues. Consider, e.g., a class with \( n \) methods in which each method enables another one and then disables itself: the contract can lead to a DFA with \( 2^n \) states. Even with a small \( n \), such a contract can be impractical to codify manually and will likely result in sub-par performance.

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

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

2. **A sub-class of DFAs suffices to express many useful code contracts.** To give semantics to lightweight annotations, we define Bit-Vector Finite Automata (BFAs): a sub-class of DFAs whose analysis uses bit-vector operations. In many practical scenarios, BFAs suffice to capture information about the enabled and disabled methods at a given point. Because this information can be codified using bit-vectors, associated static analyses can be performed efficiently; in particular, our technique is not sensitive to the number of BFA states, which in turn ensures scalability with contract and program size.

We have implemented our lightweight typestate analysis in the industrial-strength static analyzer INFER [3]. Our analysis exhibits concrete usability and performance advantages and is expressive enough to encode many relevant typestate properties in the literature. On average, compared to state-of-the-art typestate analyses, our approach requires less annotations than DFA-based analyzers and does not exhibit slow-downs due to state increase. We summarise our contributions as follows:

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

\[3\] Our code is available at [https://github.com/aalen9/lfa.git](https://github.com/aalen9/lfa.git)
Extensive evaluations for our lightweight analysis technique, which demonstrate considerable gains in performance and usability (§4).

2 Bit-vector Typestate Analysis

2.1 Annotation Language

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

Definition 1 (Annotation Language). Let \( C = \{m_0, \ldots, m_n\} \) 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 specifications is given in §2.2. We note, if \( E_i \) or \( D_i \) is \( \emptyset \) then we omit the corresponding annotation. Moreover, the BFA language can be used to derive other useful annotations defined as follows:

\[
\begin{align*}
\@EnableOnly(E_i) m_i & \overset{\text{def}}{=} \@Enable(E_i) \@Disable(C \setminus E_i) m_i \\
\@DisableOnly(D_i) m_i & \overset{\text{def}}{=} \@Disable(D_i) \@Enable(C \setminus E_i) m_i \\
\@EnableAll m_i & \overset{\text{def}}{=} \@Enable(C) m_i
\end{align*}
\]

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

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

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

\[4\] https://eigen.tuxfamily.org/dox/classEigen_1_1SparseLU.html
The SparseLU class implements a lower-upper (LU) decomposition of a sparse matrix. Eigen’s implementation uses assertions to dynamically check that: (i) analyzePattern is called prior to factorize and (ii) factorize or compute are called prior to solve. At a high-level, this contract tells us that compute (or analyzePattern().factorize()) prepares resources for invoking solve.

We notice that there are method call sequences that do not cause errors, but have redundant computations. For example, we can disallow consecutive calls to compute as in, e.g., sequences like ‘compute().compute().solve()’ as the result of the first compute is never used. Further, compute is essentially implemented as ‘analyzePattern().factorize()’. Thus, it is also redundant to call factorize after compute. The DFA that substitutes dynamic checks and avoids redundancies is given in Figure 2. Following the literature [8], this DFA can be annotated inside a class definition as in Listing 1.1. Here states are listed in the class header and transitions are specified by @Pre and @Post conditions on methods. However, this specification is too low-level and unreasonable for software engineers to annotate their APIs with, due to high annotation overheads.

In contrast, using BFA annotations the entire SparseLU class contract can be succinctly specified as in Listing 1.2. Here, the starting state is unspecified; it is determined by annotations. In fact, methods that are not guarded by other methods (like solve is guarded by compute) are enabled in the starting state. We remark that this can be overloaded by specifying annotations on the constructor method. We can specify the contract with only 4 annotations; the corresponding DFA requires 8 annotations and 4 states specified in the class header. We remark that a small change in local method dependencies by BFA annotations can result
in a substantial change of the equivalent DFA. Let \( \{m_1, m_2, m_3, \ldots, m_n\} \) be methods of some class with DFA associated (with states \( Q \)) in which \( m_1 \) and \( m_2 \) are enabled in each state of \( Q \). Adding \( \text{Enable}(m2) \) \( m1 \) doubles the number of states of the DFA as we need the set of states \( Q \) where \( m_2 \) is enabled in each state, but also states from \( Q \) with \( m_2 \) disabled in each state. Accordingly, transitions have to be duplicated for the new states and the remaining methods \( (m_3, \ldots, m_n) \).

### 2.2 Bit-vector Finite Automata

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

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

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

A BFA for a class with \( n \) methods considers states \( q_b \), where, following Def. 2, the bit-vector \( b_A \in \mathcal{B}^n \) denotes the set \( A \subseteq \Sigma_c \) enabled at that point. We often write ‘\( b \)’ (and \( q_b \)) rather than ‘\( b_A \)’ (and ‘\( q_{b_A} \)’), for simplicity. As we will see, the intent is that if \( m_i \in b \) (resp. \( m_i \notin b \)), then the \( i \)-th method is enabled (resp. disabled) in \( q_b \). Def. 3 will give a mapping from methods to triples of bit-vectors. Given \( k > 0 \), let us write \( 1^k \) (resp. \( 0^k \)) to denote a sequence of 1s (resp. 0s) of length \( k \). The initial state of the BFA is then \( q_0 = 1^n \), 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 \( \Sigma_c \) as follows

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

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

In an BFA, transitions between states \( q_b, q_{b'}, \ldots \) are determined by \( \mathcal{L}_c \). Given \( m_i \in \Sigma_c \), we have \( j \in E_i \) if and only if the \( m_i \) enables \( m_j \); similarly, \( k \in D_i \) if and only if \( m_i \) disables \( m_k \). A transition from \( q_b \) labeled by method \( m_i \) leads to state \( q_{b'} \), where \( b' \) is determined by \( \mathcal{L}_c \) using \( b \). Such a transition is defined only if a pre-condition for \( m_i \) is met in state \( q_b \), i.e., \( P \subseteq b \). In that case, \( b' = (b \cup E_i) \setminus D_i \).
These intuitions should suffice to illustrate our approach and, in particular, the local nature of enabling and disabling dependencies between methods. The following definition makes them precise.

**Definition 4 (BFA).** Given a context-independency property (BFA) for a finite sequence of method names in Automaton \((Q, \Sigma_c, \delta, q_{10^n-1}, L_c)\) where:

- \(Q\) is a finite set of states \(q_{10^n-1}, q_b, q_{0'}, \ldots\)
- \(q_{10^n-1}\) is the initial state;
- \(\Sigma_c = \{m_1, \ldots, m_n\}\) is the alphabet (method identities);
- \(L_c\) is a BFA mapping (cf. Def. 3);
- \(\delta : Q \times \Sigma_c \rightarrow Q\) is the transition function, where \(\delta(q_b, m_i) = q_{0'}\) (with \(b' = (b \cup E_i) \setminus D_i\) if \(P_i \subseteq b\), and is undefined otherwise.

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

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

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

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

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

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

\[
\begin{align*}
\delta(q_{1000}, \text{constr}) &= q_{1100} \quad \delta(q_{1100}, \text{AP}) = q_{0010} \quad \delta(q_{1100}, \text{compute}) = q_{0010} \\
\delta(q_{0010}, \text{factorize}) &= q_{0001} \quad \delta(q_{0001}, \text{solve}) = q_{1111} \quad \delta(q_{1111}, \text{AP}) = q_{0010} \\
\delta(q_{1111}, \text{compute}) &= q_{0001} \quad \delta(q_{1111}, \text{factorize}) = q_{0001} \quad \delta(q_{1111}, \text{solve}) = q_{1111}
\end{align*}
\]

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

**Definition 5 (Method sequences and concatenation).** We use \(\bar{m}\) to denote a finite sequence of method names in \(\Sigma\). Further, we use ‘·’ to denote sequence concatenation, defined as expected.

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

**Theorem 1 (Context-independency).** Let \(M = (Q, \Sigma_c, \delta, q_{10^n-1}, L_c)\) be a BFA. Also, let \(L = \{\tilde{m} : \hat{\delta}(q_{10^n-1}, \tilde{m}) = q' \land q' \in Q\}\) be the language accepted by \(M\). Then, for \(m_n \in \Sigma_c\) we have...
1. If there is \( \tilde{p} \in L \) and \( m_{n+1} \in \Sigma \) s.t. \( \tilde{p} \cdot m_{n+1} \notin L \) and \( \tilde{p} \cdot m_n \cdot m_{n+1} \in L \) then there is no \( \tilde{m} \in L \) s.t. \( \tilde{m} \cdot m_n \cdot m_{n+1} \notin L \).

2. If there is \( \tilde{p} \in L \) and \( m_{n+1} \in \Sigma \) s.t. \( \tilde{p} \cdot m_{n+1} \in L \) and \( \tilde{p} \cdot m_n \cdot m_{n+1} \notin L \) then there is no \( \tilde{m} \in L \) s.t. \( \tilde{m} \cdot m_n \cdot m_{n+1} \in L \).

Proof. Directly by Def. 4. See App. A for details.

Informally, the above theorem tells that previous calls (\( \tilde{m} \)) (i.e., context) cannot impact the effect of a call to \( m_n \) to subsequent calls (\( m_{n+1} \)). That is, Item 1. (resp. Item 2.) tells that method \( m_n \) enables (resp. disables) the same set of methods in any context. For example, a DFA that disallows modifying a collection while iterating is not a BFA (as in Fig. 3 in [4]). Let it be a Java iterator with its usual methods for collection \( c \). For the illustration, we assume a single DFA relates the iterator and its collection methods. Then, the sequence 'it.hasNext;it.next;c.remove;it.hasNext' should not be allowed, although 'c.remove;it.hasNext' should be allowed. That is, \( c.remove \) disables it.hasNext only if it.hasNext is previously called. Thus, the effect of calling \( c.remove \) depends on the calls that precede it.

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

3 Compositional Analysis Algorithm

Since BFAs can be ultimately encoded as bit-vectors, for the non-compositional case e.g., intra-procedural, standard data-flow analysis frameworks can be employed [14]. However, in the case of member objects methods being called, we present a compositional algorithm that is tailored for the INFER compositional static analysis framework. We motivate our compositional analysis technique with the example below.

Example 2. Let Foo be a class that has member \( lu \) of class SparseLU (cf. Listing 1.3). For each method of Foo that invokes methods on \( lu \) we compute a symbolic summary that denotes the effect of executing that method on typestates of \( lu \). To check against client code, a summary gives us: (i) a pre-condition (i.e., which methods should be allowed before calling a procedure) and (ii) the effect on the typestate of an argument when returning from the procedure. A simple instance of a client is wrongUseFoo in Listing 1.4.

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

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

Now, we illustrate checking client code wrongUseFoo() with computed summaries:

Above, at line 2 the abstract state is initialized with annotations of constructor Foo. At the invocation of setupLU1() (line 4) we apply sum1 in the same way as user-entered annotations are applied to transfer $s_2$ to $s_3$ above. Next, at line 6 we can see that aP is in the pre-condition set in the summary for setupLU2()
(\textit{sum}_2), computed similarly as \textit{sum}_1, but not in the enabling set of the current abstract state \textit{d}_2. Thus, a warning is raised: \texttt{foo.lu} set up by \texttt{foo.setupLU1()} is never used and overridden by \texttt{foo.setupLU2()}.

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

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

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

**Integrating Aliasing** Now, we discuss how \textit{aliasing information} can be integrated with our technique. In Ex. 2 member \texttt{lu} of object \texttt{foo} can be aliased. Thus, we keep track of BFA triplets for all base members instead of constructing an explicit BFA contract for a composed class (e.g., \texttt{Foo}). Further, we would need to generalize an abstract state to a mapping of \textit{alias sets} to BFA triplets. That is, the elements of abstract state would be \{\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_n\} \mapsto \langle \textit{E}, \textit{D}, \textit{P} \rangle where \{\texttt{a}_1, \texttt{a}_2, \ldots, \texttt{a}_n\} is a set of access paths. For example, when invoking method \texttt{setupLU1} we would need to apply its summary (\textit{sum}_1) to triplets of each alias set that contains \texttt{foo.lu} as an element. Let \textit{d}_1 = \{\texttt{S}_1 \mapsto \texttt{t}_1, \texttt{S}_2 \mapsto \texttt{t}_2, \ldots\} be an abstract state where \texttt{S}_1 and \texttt{S}_2 are the only keys such that \texttt{foo.lu} \in \texttt{S}_i for \(i \in \{1, 2\}\) and \texttt{t}_1 and \texttt{t}_2 are some BFA triplets.

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

Above, at line 2 we would need to update bindings of \texttt{S}_1 and \texttt{S}_2 (\texttt{.resp}) by applying an BFA triplet for \texttt{this.foo} from \textit{sum}_1, that is \texttt{t}_3 to \texttt{t}_1 and \texttt{t}_2 (\texttt{.resp}). The resulting abstract state \textit{d}_2 is given at line 4. We remark that if a procedure does not alter aliases, we can soundly compute and apply summaries, as shown above.

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

\[
\textit{D} : \texttt{AP} \rightarrow \bigcup_{c \in \mathcal{C}} \text{Cod}(\mathcal{L}_c)
\]

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

**Definition 6 (Join Operator).** We define \( \sqcup : \text{Cod}(\mathcal{L}_c) \times \text{Cod}(\mathcal{L}_c) \rightarrow \text{Cod}(\mathcal{L}_c) \) 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 \( \text{Cod}(\mathcal{L}_c) \) is lifted to \( \mathbb{D} \) by taking the union of un-matched entries in the mapping.

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

**Definition 7 (well-formed(\( \mathcal{L}_c \))).** Let \( c \) be a class, \( \Sigma \) methods set of class \( c \), and \( \mathcal{L}_c \). Then, \( \text{well-formed}(\mathcal{L}_c) = \text{true} \) iff \( \mathcal{L}_c(\text{constr}) = \langle E, \Sigma \setminus E, P \rangle \).

**Definition 8 (warning(\( \cdot \))).** Let \( G \) be a CFG and \( \mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k} \) be a collection of BFAs. We define \( \text{warning}(G, \mathcal{L}_{c_1}, \ldots, \mathcal{L}_{c_k}) = \text{true} \) if there is a path in \( G \) that violates some of \( \mathcal{L}_i \), for \( i \in \{1, \ldots, k\} \).

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

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

For convenience, we use \( \text{dot notation} \) to access elements of BFA triplets:

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

**Guard Predicate** Predicate \( \text{guard}(v, \sigma) \) checks whether a pre-condition for method call node \( v \) in the abstract state \( \sigma \) is met (cf. Alg. 2). We represent a call node as \( m_j(p_0 : b_0, \ldots, p_n : b_n) \) where \( p_i \) are formal and \( b_i \) are actual arguments (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 $\sigma[b_i]$. For this predicate we need the property $D = \Sigma_{c_i} \setminus E$, where $\Sigma_{c_i}$ is a set of methods for class $c_i$. This is ensured by condition well_formed($\mathcal{L}_{c_i}$) (Def. 7) and by definition of transfer() (see below).

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

**Entry-node:** (lines 3–6) This is a function entry node. For simplicity we represent it as $m_j(p_0, \ldots, p_n)$ 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 class $c_i$ that has user-supplied annotations $\mathcal{L}_{c_i}$. in line 5 we initialize the domain to the singleton map (this mapped to $\mathcal{L}_{c_i}(m_j)$). Otherwise, we return an empty map meaning that a summary has to be computed.

**Call-node:** (lines 7–20) We represent a call node as $m_j(p_0 : b_0, \ldots, p_n : b_n)$ where we assume actual arguments $b_0, \ldots, b_n$ are access paths for objects and $b_0$ represents a receiver object. The analysis is skipped if this is in the domain (line 10): this means the method has user-entered annotations. Otherwise, we transfer an abstract state for each argument $b_i$, but also for each class member whose state is updated by $m_j$. Thus, we consider all access paths in the domain of $\sigma_w$, that is $ap \in \text{dom}(\sigma_w)$ (line 11). We construct access path $ap'$ given $ap$. We distinguish two cases: $ap$ denotes (i) a member and (ii) a formal argument of $m_j$. By line 12 we handle both cases. In the former case we know $ap$ has form $this.c_1, \ldots, c_n$. We construct $ap'$ as $ap$ with this substituted for $b_0$ (actual_arg() is the identity in this case, see Def. 10): e.g., if receiver $b_0$ is $this.a$ and $ap$ is $this.c_1, \ldots, c_n$.
Algorithm 3: Transfer Function

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

1 Procedure transfer \((v, \sigma)\)
2 \hspace{1em} switch \(v\) do
3 \hspace{2em} case Entry-node\(m_j(p_0, \ldots, p_n)\) do
4 \hspace{3em} Let \(c_i\) be the class of method \(m_j(p_0, \ldots, p_n)\);
5 \hspace{3em} if There is \(L_w\) then return \(\{this \mapsto L_w(m_j)\}\);
6 \hspace{3em} else return EmptyMap;
7 \hspace{2em} case Call-node\(m_j(p_0 : b_0, \ldots, p_n : b_n)\) do
8 \hspace{3em} Let \(\sigma_w\) be an abstract state of exit_node\((v)\);
9 \hspace{3em} Initialize \(\sigma' := \sigma\);
10 \hspace{3em} if this not in \(\sigma'\) then
11 \hspace{4em} for \(ap\) in dom\((\sigma_w)\) do
12 \hspace{5em} \(ap' = \text{actual}_\text{arg}(ap\{b_0/this\}, v)\);
13 \hspace{5em} if \(ap'\) in dom\((\sigma)\) then
14 \hspace{6em} \(E' = (\sigma[ap'], E \cup \sigma_w[ap], E) \setminus \sigma_w[ap], D)\);
15 \hspace{6em} \(D' = (\sigma[ap'], D \cup \sigma_w[ap], D) \setminus \sigma_w[ap], E)\);
16 \hspace{6em} \(P' = \sigma[ap'], P \cup (\sigma_w[ap], P) \setminus \sigma[ap'], E)\);
17 \hspace{6em} \(\sigma'[ap] = (E', D', P')\);
18 \hspace{5em} else
19 \hspace{6em} \(\sigma'[ap] := \sigma_w[ap]\);
20 return \(\sigma'\)
21 otherwise do
22 return \(\sigma\)
23

then \(ap' = this.a.c_1, \ldots, c_n\). In the latter case \(ap\) denotes formal argument \(p_i\) and \(\text{actual}_\text{arg}(\cdot)\) returns corresponding actual argument \(b_i\) (as \(p_i\{b_0/this\} = p_i\)).

Now, as \(ap'\) is determined we construct its BFA triplet. If \(ap'\) is not in the domain of \(\sigma\) (line 13) we copy a corresponding BFA triplet from \(\sigma_w\) (line 19). Otherwise, we transfer elements of an BFA triplet at \(\sigma[ap']\) as follows. The resulting enabling set is obtained by (i) adding methods that \(m_j\) enables \((\sigma_w[ap], E)\) to the current enabling set \(\sigma[ap'], E\), and (ii) removing methods that \(m_j\) disables \((\sigma_w[ap], D)\), from it. The disabling set \(D'\) is constructed in a complementary way. Finally, the pre-condition set \(\sigma[ap'], P\) is expanded with elements of \(\sigma_w[ap], P\) that are not in the enabling set \(\sigma[ap'], E\). We remark that the property \(D = \Sigma_w \setminus E\) is preserved by the definition of \(E'\) and \(D'\). 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 per argument. That is, BFA analysis is insensitive to the number of states, as a set of states is abstracted as a single set.

Note, in our implementation we use several features specific to INFER: (1) INFER’s summaries which allow us to use a single domain for intra and inter-procedural analysis; (2) scheduling on CFG top-down traversal which simplify the handling of branch statements. In principle, BFA can be implemented in other frameworks e.g., IFDS \cite{18}.

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

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

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

1. For \( m \in \Sigma_c \), it holds; \( \delta(q_b, m) \) is defined for all \( q_b \in P \) iff \( \delta(q_b, m) \) is defined.

2. Let \( \sigma = \mathcal{L}_c(m) \). If \( P' = \{ \delta(q_b, m) : q_b \in P \} \) then \( \bigcap_{q_i \in P'} b = \llbracket \sigma \rrbracket(b_\ast) \).

**Proof.** By induction on cardinality of \( P \) and Def. 4. See App. A for details.

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

**Definition 13 (\( \text{dtransfer}_c(\cdot) \)).** Let \( c \in \mathcal{C} \) be a class, \( \Sigma_c \) be a set of methods of \( c \), and \( \mathcal{L}_c \) be a BFA. Further, let \( m \in \Sigma_c \) be a method, \( \langle E^m, D^m, P^m \rangle = \mathcal{L}_c(m) \), and \( \langle E, D, P \rangle \in \text{Cod}(\mathcal{L}_c) \). Then,
\[
\text{dtransfer}_c(m, \langle E, D, P \rangle) = \langle E', D', P' \rangle
\]
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 and \( \phi = \langle E, D, P \rangle \), then
\[
\text{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \phi) = \text{dtransfer}_c(m_{n+1}, \text{dtransfer}_c(m_1, \ldots, m_n, \phi))
\]
Relying on Thm. 2 we state the soundness of join:

**Theorem 3 (Soundness of \( \sqcap \)).** Let \( q_i \in Q \) and \( \phi_i = \langle E_i, D_i, P_i \rangle \) for \( i \in \{1, 2\} \). Then, \( \llbracket \phi_1 \rrbracket(b) \sqcap \llbracket \phi_2 \rrbracket(b) = \llbracket \phi_1 \sqcup \phi_2 \rrbracket(b) \).

**Proof.** By definitions Def. 6 and Def. 12 and set laws. See App. A for details.

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

**Theorem 4 (Correctness of \( \text{dtransfer}_c(\cdot) \)).** Let \( M = (Q, \Sigma, \delta, q_{10^{n-1}}, \mathcal{L}_c) \). Let \( q_b \in Q \) and \( m_1 \ldots m_n \in \Sigma^* \). Then
\[
\text{dtransfer}_c(m_1 \ldots m_n, (\emptyset, \emptyset, \emptyset)) = \langle E', D', P' \rangle \iff \hat{\delta}(q_b, m_1 \ldots m_n) = q_b'
\]
where \( b' = \llbracket \langle E', D', P' \rangle \rrbracket(b) \).

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

We evaluate our technique to validate the following two claims:

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

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

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

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

We analyze a benchmark of 18 contracts that specify common patterns of locally dependent contract annotations for a class. Moreover, we auto-generate 122 client programs parametrized by lines of code, number of composed classes, if-branches, and loops. Note, the code is such that it does not invoke the need for aliasing (as we do not support it yet in our BFA implementation). Client programs follow the compositional patterns we described in Ex. 2 which can also be found in [12].

The annotations for BFA are manually specified; from them, we generate minimal DFAs representations in DFA annotation format and Topl annotation format.

Our use case is to integrate static analyses in interactive IDEs e.g., Microsoft Visual Studio Code [20], so that code can be analyzed at coding time. For this reason, our use case requires low-latency execution of the static analysis. Our SLA is based on the RAIL user-centric performance model [1].

**Usability Evaluation** Fig. 4 outlines the key features of the 18 contracts we considered, called CR-1 – CR-18. In App. B we detail CR-4 as an example. For each contract, we specify the number of methods, the number of DFA states the contract corresponds to, and number of atomic annotation terms in BFA, DFA, and Topl. An atomic annotation term is a standalone annotation in the given
Fig. 4: Details of the 18 contracts in our evaluation.
annotation language. We can observe 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 4 or more methods, a manual specification using DFA or TOPL annotations becomes impractical. Overall, we validate Claim-I by the fact that BFA requires less annotation overhead on all of the contracts, making contract specification more practical.

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

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

Comparing BFA-based analysis vs TOPL typestate implementations (Execution time). Here again client programs do not require aliasing. Fig. 5c compares various class compositions for client programs of 500-1K LoC. The TOPL implementation sharply increases in execution-time as the number of states increases, quickly missing the SLA. In contrast, the BFA implementation remains constant always under the SLA. Overall, BFA produces a geometric mean speedup over TOPL of 6.59×. Fig. 5d compares various class compositions for client programs of 15K LoC. Both implementations fail to meet the SLA. The TOPL implementa-
Fig. 5: Runtime comparisons. Each line represents a different number of base classes composed in a client code.

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

5 Related Work

We focus on comparisons with restricted forms of typestate contracts. We refer to the typestate literature \[19, 15, 8, 5, 7\] for a more general treatment. The work \[13\] proposes restricted form of typestates tailored for use-case of the object construction using the builder pattern. This approach is restricted in that it only accumulates called methods in an abstract (monotonic) state, and it does not
require aliasing for supported contracts. Compared to our approach, we share the idea of specifying typestate without explicitly mentioning states. On the other hand, their technique is less expressive than our annotations. They cannot express various properties we can (e.g., the property "cannot call a method"). Similarly, [10] 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 [9] 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 [7] allows DFA-based specifications, but also annotations for describing specific resource protocols contracts. These annotations have a locality flavor — annotations on one method do not refer to other methods. Moreover, we share the idea of specifying typestate without explicitly mentioning states. These additional annotations in Fugue are more expressive than DFA-based typestates (e.g., “must call a release method”). We conjecture that “must call” property can be encoded as bit-vectors in a complementary way to our BFA approach. We leave this extension for future work.

Our annotations could be mimicked by having a local DFA attached to each method. In this case, the DFAs would have the same restrictions as our annotation language. We are not aware of prior work in this direction. We also note that while our technique is implemented in Infer using the algorithm in §2, the fact that we can translate typestates to bit-vectors allows typestate analysis for local contracts to be used in distributive dataflow frameworks, such as IFDS [18], without the need for modifying the framework for non-distributive domains [16].

6 Concluding Remarks

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

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

References

1. RAIL model. [https://web.dev/rail/](https://web.dev/rail/) accessed: 2021-09-30
2. Infer TOPL. [https://fbinfer.com/docs/checker-topl/](https://fbinfer.com/docs/checker-topl/) (2021)
3. Arslanagić, A., Subotić, P., Pérez, J.A.: Lfa checker: Scalable typestate analysis for low-latency environments (Mar 2022). https://doi.org/10.5281/zenodo.6393183

4. Bierhoff, K., Aldrich, J.: Modular Typestate Checking of Aliased Objects. In: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. p. 301–320. OOPSLA ’07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1297027.1297050

5. Bodden, E., Hendren, L.: The Clara framework for Hybrid Typestate Analysis. Int. J. Softw. Tools Technol. Transf. 14(3), 307–326 (Jun 2012)

6. Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 459–465. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)

7. Deline, R., Fähndrich, M.: The Fugue protocol checker: Is your software Baroque? Tech. Rep. MSR-TR-2004-07, Microsoft Research (04 2004)

8. DeLine, R., Fähndrich, M.: Typestates for Objects. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming, pp. 465–490. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

9. Emmi, M., Hadarean, L., Jhala, R., Pike, L., Rosner, N., Schäf, M., Sengupta, A., Visser, W.: RAPID: Checking API Usage for the Cloud in the Cloud. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 146–1426. ESEC/FSE 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3468264.3473934

10. Fähndrich, M., Leino, R.: Heap Monotonic Typestate. In: Proceedings of the first International Workshop on Alias Confinement and Ownership (IWACO) (July 2003), https://www.microsoft.com/en-us/research/publication/heap-monotonic-typestate/

11. Fähndrich, M., Logozzo, F.: Static Contract Checking with Abstract Interpretation. In: Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software. pp. 10–30. FoVeOOS’10, Springer-Verlag, Berlin, Heidelberg (2010)

12. Jakobsen, M., Ravier, A., Dardha, O.: Papaya: Global Typestate Analysis of Aliased Objects. In: 23rd International Symposium on Principles and Practice of Declarative Programming. PPDP 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3479394.3479414

13. Kellogg, M., Ran, M., Sridharan, M., Schäf, M., Ernst, M.D.: Verifying Object Construction. In: ICSE 2020, Proceedings of the 42nd International Conference on Software Engineering, Seoul, Korea (May 2020)

14. Khedker, U., Sanyal, A., Sathe, B.: Data Flow Analysis: Theory and Practice. CRC Press (2017), https://books.google.rs/books?id=9PyrtgNBdgoC

15. Lam, P., Kuncak, V., Rinard, M.: Generalized Typestate Checking Using Set Interfaces and Pluggable Analyses. SIGPLAN Not. 39(3), 46–55 (Mar 2004). https://doi.org/10.1145/981009.981016

16. Naeem, N.A., Lhoták, O., Rodriguez, J.: Practical Extensions to the IFDS algorithm. In: Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction. pp. 124–144. CC’10/ETAPS’10, Springer-Verlag, Berlin, Heidelberg
17. Paul, R., Turzo, A.K., Bosu, A.: Why Security Defects Go Unnoticed during Code Reviews? A Case-Control Study of the Chromium OS project. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. pp. 1373–1385. IEEE (2021). https://doi.org/10.1109/ICSE43902.2021.00124

18. Reps, T., Horwitz, S., Sagiv, M.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61. POPL ’95, Association for Computing Machinery, New York, NY, USA (1995). https://doi.org/10.1145/199448.199462

19. Strom, R.E., Yemini, S.: Typestate: A Programming Language Concept for Enhancing Software Reliability. IEEE Trans. Software Eng. 12(1), 157–171 (1986). https://doi.org/10.1109/TSE.1986.6312929

20. Subotić, P., Milikić, L., Stojić, M.: A Static analysis Framework for Data Science Notebooks. In: ICSE’22: The 44th International Conference on Software Engineering (May 21-May 29 2022)

21. Szabó, T., Erdweg, S., Voelter, M.: IncA: A DSL for the Definition of Incremental Program Analyses. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 320–331. ASE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2970276.2970298

22. Yahav, E., Fink, S.: The SAFE Experience, pp. 17–33. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-19823-6_3
### A Proofs

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

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

**Proof.** We only consider the first item, as the second item is shown similarly. By Theorem 1 (Context-independency).

We have $m_{n+1} \in E_n$ (1)

Further, for any $\tilde{m} \in \Sigma^*$ let $q_0$ be such that $\delta(q_{10^n-1}, \tilde{m}) = q_0$ and $q_{0'}$ s.t. $\delta(q_0, m_n) = q_{0'}$. Now, by the definition of Def. 4 we have that $P_{n+1} = \{ m_{n+1} \} \subseteq \emptyset$. Thus, for all $\tilde{m} \in L$ we have $\tilde{m} \cdot m_n \cdot m_{n+1} \in L$. This concludes the proof.

**Theorem 2 (BFA ⊨-Property).** Let $M = (Q, \Sigma_c, \delta, q_{10^n-1}, L_c)$, $P \subseteq Q$, and $b_0 = \bigcap_{q_0 \in P} b$, then

1. For $m \in \Sigma_c$, it holds: $\delta(q_0, m)$ is defined for all $q_0 \in P$ iff $\delta(q_0, m)$ is defined.
2. Let $\sigma = L_c(m)$. If $P' = \{ \delta(q_0, m) : q_0 \in P \}$ then $\bigcap_{q_0 \in P'} b = \| \sigma \| (b_0)$.

**Proof.** We show two items:

1. By Def. 1 for all $q_0 \in P$ we know $\delta(q_0, m)$ is defined when $P \subseteq b$ with $(E, P, D) = L_c(m)$. So, we have $P \subseteq \bigcap_{q_0 \in P} b = b$ and $\delta(q_0, m)$ is defined.
2. By induction on $|P|$, $|P| = 1$. Follows immediately as $\bigcap_{q_0 \in \{ q_0 \}} q_0 = q_0$.
   - $|P| > 1$. Let $P = P_0 \cup \{ q_0 \}$. Let $|P_0| = n$. By IH we know
     \[
     \bigcap_{q_0 \in P_0} \| \sigma \| (b) = \| \sigma \| \bigcap_{q_0 \in P_0} b
     \]
     (2)

We should show

\[
\bigcap_{q_0 \in (P_0 \cup \{ q_0' \})} \| \sigma \| (b) = \| \sigma \| \bigcap_{q_0 \in (P_0 \cup \{ q_0' \})} b
\]

We have

\[
\bigcap_{q_0 \in (P_0 \cup \{ q_0' \})} \| \sigma \| (b) = \bigcap_{q_0 \in P_0} \| \sigma \| (b) \cap \| \sigma \| (b')
\]

(by [2])

\[
= \| \sigma \| (b) \cap \| \sigma \| (b')
\]

(by [2])

\[
= (b \cup E) \cap ((b' \cup E) \setminus D)
\]

(by set laws)

\[
= \| \sigma \| (b \cap b') = \| \sigma \| \bigcap_{q_0 \in (P_0 \cup \{ q_0' \})} b
\]
where \( b_\ast = \llbracket \sigma \rrbracket (\bigcap_{q_b \in F_b} b) \). This concludes the proof.

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

**Proof.** By set laws we have:

\[
\llbracket \phi_1 \rrbracket (b) \cap \llbracket \phi_2 \rrbracket (b) = ((b \cup E_1) \setminus D_1) \cap ((b \cup E_2) \setminus D_2) \\
= ((b \cup E_1) \cap (b \cup E_2)) \setminus (D_1 \cup D_2) \\
= (b \cup (E_1 \cap E_2)) \setminus (D_1 \cup D_2) \\
= (b \cup (E_1 \cap E_2) \setminus (D_1 \cup D_2)) \setminus (D_1 \cup D_2) = \llbracket \phi_1 \sqcup \phi_2 \rrbracket (b)
\]

This concludes the proof.

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

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

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

**Proof.** — (\( \Rightarrow \)) Soundness: By induction on the length of method sequence \( \overline{m} = m_1, \ldots, m_n \).

- Case \( n = 1 \). In this case we have \( \overline{m} = m_1 \). Let \( \langle E^m, D^m, \{m_1\} \rangle = \mathcal{L}_c(m_1) \).

  By Def. 13 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. 4 we have \( \hat{\delta}(q_b, m_1, \ldots, m_n) = q_{b'} \).

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

\[
dtransfer_c(m_1, \ldots, m_n, (\emptyset, \emptyset, \emptyset)) = \langle E', D', P' \rangle \Rightarrow \hat{\delta}(q_b, m_1, \ldots, m_n) = q_{b'} \tag{3}
\]

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

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

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 Def. 13 we have

\[
dtransfer_c(m_1, \ldots, m_n, m_{n+1}, (\emptyset, \emptyset, \emptyset)) = dtransfer_c(m_{n+1}, \langle E', D', P' \rangle)
\]
Further, we have
\[ E'' = (E' \cup E^n) \setminus D^n \quad D'' = (D' \cup D^n) \setminus E^n \quad P'' = P' \cup (P^m \setminus E') \] (5)

Now, by substitution and De Morgan’s laws we have:
\[
b'' = (b \cup E'') \setminus D'' =
= (b \cup ((E' \cup E^n) \setminus D^n)) \setminus ((D' \cup D^n) \setminus E^n)
= ((b \cup (E' \cup E^n)) \setminus (D' \setminus E^n)) \setminus D^n
= (((b \cup E') \setminus D') \cup E^n) \setminus D^n
= (b' \cup E^n) \setminus D^n
\]

Further, 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)). So, we can see that by definition of Def. 4 we have \( \delta(q_{b'}, m_{n+1}) = q_{b''} \). This concludes this case.

- (\( \Leftarrow \)) Completeness:
  
  - \( n = 1 \). In this case \( \tilde{m} = m_1 \). Let \( \langle E^n, D^n, \{m_1\} \rangle = L_c(m_1) \). By Def. 4 we have \( b' = (b \cup E^n) \setminus D^n \) and \( \{m_1\} \subseteq b \). By Def. 13 we have \( E' = E^n \), \( D' = D^n \), and \( P' = \{m_1\} \). Thus, as \( \{m_1\} \cap \emptyset = \emptyset \) we have \( b' = ([E', D', P']) \langle b \rangle \).
  
  - \( n > 1 \). Let \( \tilde{m} = m_1, \ldots, m_n, m_{n+1} \). By IH we know
    \[
    \hat{\delta}(q_b, m_1, \ldots, m_n) = q' \Rightarrow \text{dtransfer}_c(m_1, \ldots, m_n, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E', D', P' \rangle
    \] (6)
    
    where \( b' = (b \cup E') \setminus D' \) and \( P' \subseteq b \). Now, we assume
    \[
    \hat{\delta}(q_b, m_1, \ldots, m_n, m_{n+1}) = q_{b''}
    \] (7)
    
    We should show that
    \[
    \text{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \langle E'', D'', P'' \rangle
    \]
    such that \( b'' = (b \cup E'') \setminus D'' \) and \( P'' \subseteq b \). We know
    \[
    \text{dtransfer}_c(m_1, \ldots, m_n, m_{n+1}, \langle \emptyset, \emptyset, \emptyset \rangle) = \text{dtransfer}_c(m_{n+1}, \langle E', D', P' \rangle)
    \]
    
    By Def. 4 we have:
    \[
    \delta(q_b, m_1, \ldots, m_n, m_{n+1}) = \delta(\delta(q_b, m_1, \ldots, m_n), m_{n+1}) = q_{b''}
    \]
    
    So by (4) and (7) we have \( \{m_{n+1}\} \subseteq b' \) and \( b' = (b \cup E') \setminus D' \). It follows \( \{m_{n+1}\} \cap D' = \emptyset \). That is, \( \text{dtransfer}_c(m_{n+1}, \langle E', D', P' \rangle) \) 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.
Now, we discuss specialization of Thm. 4 for the code checking. In this case, we know that a method sequence starts with the constructor method (i.e., the sequence is of the form $\text{constr}, m_1, \ldots, m_n$) and $q_{10^{n-1}}$ is the input state. By $\text{well\_formed}(L_c)$ (Def. 7) we know that if $\delta(q_{10^{n-1}}, \text{constr}) = q_b$ and 

$$dtransfer_c(\text{constr}, m_1, \ldots, m_n, (\emptyset, \emptyset, \emptyset)) = \sigma$$

then methods not enabled in $q_b$ are in the disabling set of $\sigma$. 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 $\text{well\_formed}(L_c)$ did not hold, the algorithm would fail to detect an error as it would put $m_k$ in $P$ since $m_k \notin E$. 
B Sample Contract used in Evaluations (§4)

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

**Listing 1.5: SparseLU LFA CR4 contract**

```java
class SparseLU {
    states q0, q1, q2, q3, q4;
    void analyzePattern(Mat a);
    void factorize(Mat a);
    void compute(Mat a);
    void solve(Mat b);
    void transpose();
}
```

**Listing 1.6: SparseLU DFA CR4 contract**

![DFA diagram of SparseLU CR-4 contract](image)

**Fig. 6: DFA diagram of SparseLU CR-4 contract**

```
property SparseLU
    prefix = "SparseLU"
```
Listing 1.7: SparseLU TOPL CR4 contract