PAPER

Optimization Methods for Nop-Shadows Typestate Analysis

Chengsong WANG[†a)], **Xiaoguang MAO**[†,††b)], ***Nonmembers***, **Yan LEI**[†c)], ***Student Member***, ***and*** **Peng ZHANG**[†d)], ***Nonmember***

SUMMARY In recent years, hybrid typestate analysis has been proposed to eliminate unnecessary monitoring instrumentations for runtime monitors at compile-time. Nop-shadows Analysis (NSA) is one of these hybrid typestate analyses. Before generating residual monitors, NSA performs the data-flow analysis which is intra-procedural flow-sensitive and partially context-sensitive to improve runtime performance. Although NSA is precise, there are some cases on which it has little effects. In this paper, we propose three optimizations to further improve the precision of NSA. The first two optimizations try to filter interferential states of objects when determining whether a monitoring instrumentation is necessary. The third optimization refines the inter-procedural data-flow analysis induced by method invocations. We have integrated our optimizations into Clara and conducted extensive experiments on the DaCapo benchmark. The experimental results demonstrate that our first two optimizations can further remove unnecessary instrumentations after the original NSA in more than half of the cases, without a significant overhead. In addition, all the instrumentations can be removed for two cases, which implies the program satisfy the typestate property and is free of runtime monitoring. It comes as a surprise to us that the third optimization can only be effective on 8.7% cases. Finally, we analyze the experimental results and discuss the reasons why our optimizations fail to further eliminate unnecessary instrumentations in some special situations.

key words: *typestate analysis, runtime monitoring, static analysis, Nop-shadows Analysis, data-flow analysis*

1. Introduction

Programmers usually have to conform to some constraints when using the third-party or system libraries. For example, programmers cannot call the method "*write*" until the method "*open*" is called on the same File object. These constraints are also known as the typestate properties, and can be expressed by temporal specifications with free variables. A typestate property [1] describes the acceptable operations on a single object or a group of inter-related objects, according to the current state (i.e., the typestate) of the object or the group [2], [3]. Lots of large-scale software system errors are caused by the violations of typestate properties. What is worse, it is very difficult and time-consuming to find and fix these errors [4], [5]. The static analysis of a program with respect to a typestate property is generally undecidable. Existing static typestate property checking tools [6], [7] suffer from scalability and false-alarm problems. Dynamic typestate property checking approaches complement static approaches with runtime monitoring to improve the scalability and the precision of analyses, but sacrifice the completeness.

Usually, dynamic typestate analysis approaches, such as runtime verification [8]–[11], automatically convert typestate properties into runtime monitors that can detect the property violations at runtime. Implementing runtime monitoring needs to instrument the monitored programs. The instrumentation can be done manually or automatically based on existing techniques, such as AOP [12]. However, the programs monitored by runtime monitors usually contain many redundant instrumentations, which result in a significant monitoring overhead. Therefore, some approaches [4], [13]–[17] exploit static analysis information to remove provable unnecessary instrumentations at compile time to reduce the overhead of runtime monitoring. These approaches are often called hybrid typestate analyses.

Theoretically, hybrid typestate analysis is equivalent to the static analysis of typestate properties. If all the instrumentations of runtime monitors can be removed, the program is proved to satisfy the typestate property and requires no monitoring at runtime. Nop-shadows analysis (NSA) [2], [4], [18] is one of the existing hybrid analysis approaches. It is implemented in Clara [19] to mitigate the overhead of runtime monitoring of large-scale Java programs. NSA uses a novel combination of forward and additional backward data-flow analysis to remove redundant instrumentations generated for monitors. In order to be efficient, it only performs flow-sensitive data-flow analysis on an intra-procedural level only. Although NSA is precise, there are more than 20% cases in which some unnecessary instrumentations still remain after NSA. What is even worse, NSA has little effects on some of these cases. In order to find out the reasons, we dissected the algorithm of data-flow analysis used in the NSA, and made an investigation on the failed cases. One of the main reasons is that NSA does not make full use of the program structure information to refine the intra-procedural data-flow analysis. Secondly, NSA is not inter-procedural flow-sensitive, and the overly conservative approximations of inter-procedural cases in NSA reduce its accuracy.

Manuscript received October 8, 2014.
Manuscript revised January 19, 2015.
Manuscript publicized February 23, 2015.

[†]The authors are with School of Computer, National University of Defense Technology, 410073, Changsha, China.
[††]The author is with Laboratory of Science and Technology on Integrated Logistics Support, National University of Defense Technology, Changsha 410073, China.
a) E-mail: jameschen186@gmail.com
b) E-mail: xgmao@nudt.edu.cn
c) E-mail: yanlei@nudt.edu.cn
d) E-mail: pengZhang@nudt.edu.cn

DOI: 10.1587/transinf.2014EDP7329

In this paper, we propose three optimizations to improve the precision of NSA. The first two optimizations can both filter interferential states of objects produced during the data-flow analysis. Interferential states of objects refer to the states that cannot be actually reached by the corresponding objects at runtime, but can prevent NSA from identifying some unnecessary instrumentations. Specifically, the first optimization identifies changeless configurations produced by the backward data-flow analysis of NSA. the second one utilizes local object information to refine the iterations of data-flow analysis on the currently analysed method. The third optimization refines the inter-procedural data-flow analysis induced by method invocations with partial flow-sensitive information contained in the called methods. Compared to the full inter-procedural flow-sensitive static analysis, the extra overhead incurred by our optimization is negligible. These three optimizations can remove more unnecessary monitoring instrumentations separately in different situations.

To evaluate our optimizations, we have integrated our optimizations into Clara, and applied them to the DaCapo benchmark suite [20]. Our optimizations can further remove unnecessary instrumentations after the original NSA in more than half of the cases. In two cases, we get a perfect result, *i.e.*, all the monitoring instrumentations are removed, entirely obviating the need for monitoring at runtime.

To summarize, our paper has the following contributions:

(i) Propose three optimizations for NSA to improve the precision of the analysis. The first two optimizations filter interferential states of objects by identifying changeless configurations and exploiting local object information respectively. The third optimization refines inter-procedural data-flow analysis with partial flow-sensitive information contained in the called methods.
(ii) Propose and implement an approximate, but sound, intra-procedural flow-sensitive algorithm to determine whether a variable points to a local object.
(iii) Design an algorithm to determine whether two static objects coming from a caller and the responding callee respectively are must-alias during the execution of the caller.
(iv) Implement the three optimizations and integrate them into Clara.
(v) Conduct extensive experiments on the DaCapo benchmark suite to show the effectiveness of our optimizations.

This paper is extended from our conference paper [21] published in RV 2013. Specifically, compared to the works in that paper, we propose a new idea to refine the inter-procedural data-flow analysis (i.e., the third optimization). Additionally, we implemented the new optimization and conducted more experiments on the DaCapo benchmark suite [20]. This paper also analyses the experimental results with respect to the new optimization.

The remainder of this paper is organized as follows. We begin with an overview of NSA in Sect. 2. In Sect. 3, we show three different motivating examples separately to illustrate the responding optimization methods. Section 4 formulates the details of our proposed optimizations. We describe our experiments and analyze the experimental results in Sect. 5. Section 6 describes the related work and the paper is concluded in Sect. 7.

2. Nop-shadows Analysis

As in the literature [2], [4], [22], we also use the term "shadow" to represent an instrumentation point created for runtime monitoring. Usually, a shadow is a method invocation on an object or a group of inter-related objects. NSA [2], [4] is a hybrid typestate analysis method and implemented in the Clara framework [4], which extends tracematch [23] with static analysis to remove "nop shadows". Here a "nop shadow" means that the shadow does not influence the results of runtime monitoring, *i.e.*, it can neither trigger nor suppress a property violation.

Clara consists of three static analysis stages, in which NSA is the most expensive and precise one. Given a typestate property (usually a finite-state machine (FSM)) and an instrumented Java program, NSA checks whether a shadow in a method of the program can be removed via data-flow analysis. The basic idea of NSA is to compute the reachable states of each statement in a program according to the semantics of the program and the monitored typestate property. Given an FSM typestate property M and its state set S, for each statement st, there are two types of reachable states: $source(st)$ and $futures(st)$, which are calculated respectively by a *forward* data-flow analysis (forward analysis) and a *backward* data-flow analysis (backward analysis). The source set $source(st) \subseteq S$ contains all the states that can be reached before executing st from the beginning of the program; $futures(st) \subseteq \mathbb{P}(S)$ is the future set, and each element of $futures(st)$ contains the states from which the remainder of program execution after st can reach a final state (usually the error state) of M. Therefore, for a given shadow s, NSA identifies s as a "nop shadow" if the execution of the shadow has no impact on the monitoring result, which can be formalized by the following two conditions:

- $target(s) \cap F = \emptyset$, where $target(s) = \{q_2 \mid \exists q_1 \in sources(s) \bullet q_2 = \delta(q_1, s)\}$ is the resulting state set after executing s, δ is the transition function of M, and F is the final state set of M. This condition means the execution of s does not directly lead to an error state.
- $\forall q_1 \in source(s), \forall Q \in futures(s) \bullet q_1 \in Q \Leftrightarrow \delta(q_1, s) \in Q$. It means the execution of s does not influence whether or not a final state will be reached.

The shadow s can be removed if both conditions are satisfied.

Figure 1 gives an example for NSA. The left part is an FSM for "ConnectionClosed" [2] typestate property, which requires the "*write*" operation should not be called after a

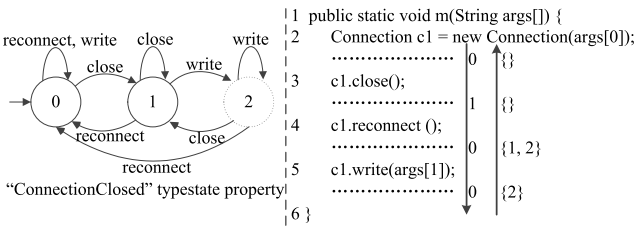

Fig. 1 An example for Nop-shadows Analysis.

Connection object is closed. The right part displays a program segment annotated with the state information of each statement. The elements in the *source* set and the *futures* set of each statement are next to the downward and upward arrows, respectively. For instance, for the shadow s_3 at line 3, we have:

$$source(s_3) = \{0\}$$
$$target(s_3) = \{1\}$$
$$futures(s_3) = \emptyset$$

$futures(s_3) = \emptyset$ means that there is no state from which the property state machine can reach the final state via the execution after line 3. According to the preceding two conditions, s_3 is a "nop shadow" that can be safely removed. That is, the runtime monitor does not need to monitor the method call statement at line 3 in this example at runtime.

After removing a "nop shadow", the $source(st)$ and $futures(st)$ of each statement will be calculated again, until no "nop shadow" exists. If there is no shadow left after NSA, the program is proved to satisfy the typestate property. For example, all the shadows of the program in Fig. 1 will be removed eventually.

2.1 Worklist Algorithm

Eric Bodden presented the Algorithm 1 to compute the reachable states for every statement in a given method [2], [4]. In order to distinguish the states of multiple different objects or groups of inter-related objects, the algorithm propagates "configurations" instead of state sets. A configuration specifies the state information of some specific objects. A configuration $c = (Q, b)$ is composed by a state set Q and a variable binding b. Informally, the variable binding specifies the static objects [24] which represent the concrete runtime objects at compile-time. Actually, a shadow s also has a variable binding [25] specifying the objects whose states can be changed by s. Two variable bindings are *compatible* if they can be bound to the same static object or the same group of inter-related static objects. A configuration and a shadow are *compatible* if their variable bindings are compatible. For a statement st associated with a configuration (Q, b), in forward analysis, the elements in set Q represent all the possible states which the static objects specified by the variable binding b can reach just before executing st; in backward analysis, they are the states from which the static objects specified by b can reach a final state via the execution after st. For example, the configuration S_8 in Fig. 3 represents that the static object O_2 can reach state 2 before executing the statement at line 8. The configuration F_{12} in Fig. 4 represents that the static object O can reach the final state from state 0 or 1 via the execution after line 3. Moreover, Algorithm 1 introduces a notion, $job(stmt, cs)$, to represent a configuration set cs associated to the statement $stmt$.

Algorithm 1 worklist($initial$, $succ_{cfg}$, $succ_{ext}$, δ)

```
1:  wl := initial;
2:  before := after := λstmt.∅
3:  while wl non-empty do
4:      pop job(stmt, cs) from wl
5:      before := before[stmt ↦ before(stmt) ∪ cs]
6:      cs' := { cs   if shadows(stmt) = ∅
               { ∅    otherwise
7:      for c ∈ cs, s ∈ shadows(stmt) do
8:          cs' := cs' ∪ transition(c, s, δ)
9:      end for
10:     cs_new := cs' − after(stmt)
11:     if cs_new non-empty then
12:         after := after[stmt ↦ after(stmt) ∪ cs_new]
13:         for stmt' ∈ succ_cfg(stmt) do
14:             wl := wl ∪ {(stmt', cs_new)}
15:         end for
16:         for stmt' ∈ succ_ext(stmt) do
17:             wl := wl[stmt' ↦ wl(stmt')
18:                   ∪reaching(cs_new, relatedShadows(stmt))]
19:         end for
20:     end if
21: end while
```

In fact, for a given shadow-bearing method m, both the forward analysis and the backward analysis are performed by running the general worklist algorithm, *i.e.*, Algorithm 1. The differences between them are the initial values of arguments passed to the algorithm. For the forward analysis, the first argument *initial* is a *job* set, which contains all the configurations that may be reached at the entry statement of m from the beginning of the program. The second and third arguments, *i.e.*, $succ_{cfg}$ and $succ_{ext}$, are successor functions modelling all the possible intra and inter procedural control-flow of m respectively. As shown in Fig. 2, the dashed arrows represent the intra procedural control-flow of m, and solid arrows (1, 2, 3, 4) represent four types of possible inter procedural control-flow of m. Solid arrows (1) and (2) are used to model the transitively recursive method calls to m. We cannot determine whether a method call must be transitively recursive at compile-time. Hence, both of the arrows (3a) and (3b) are used to model the non-recursive method calls within m. Additionally, method m can re-executes again after its returning. Arrows (4) is used to model this case. The last argument δ is the translation function of $\mathcal{M}$. As to the backward analysis, we first 1) flip all edges and exchange the initial states and final states in the $\mathcal{M}$ to get a reversed FSM $\mathcal{M}'$; 2) get reversed successor functions, $succ'_{cfg}$ and $succ'_{ext}$, by flipping all the edges in intra and inter procedural control-flow graph of m. Hence, $succ'_{cfg}$, $succ'_{ext}$ and δ' (the translation function of $\mathcal{M}'$) are passed separately as the second, third and forth argument to Algorithm 1. Moreover, the first argument is the set of *jobs* which

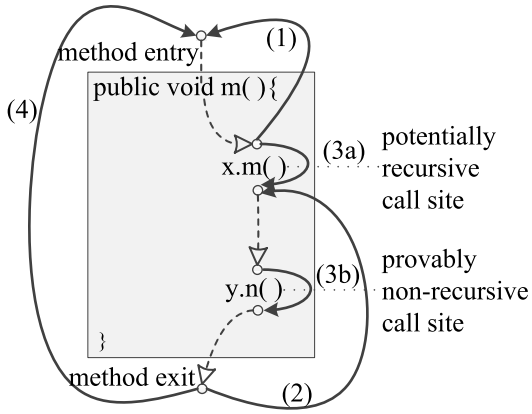

Fig. 2 Intra and inter procedural control-flow of the method *m* [2].

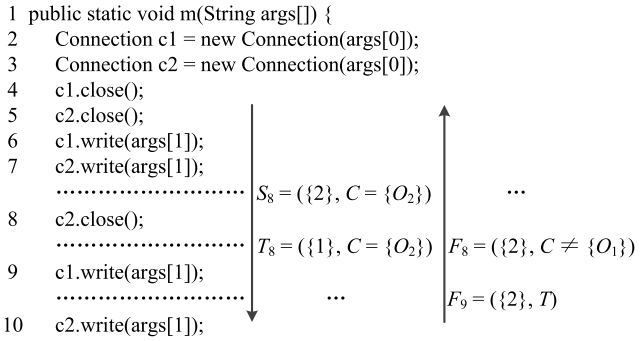

Fig. 3 The example for motivating the first optimization.

associate any "final" shadow[†], exit or method call statement *stmt* in *m* with any configuration that can reach a final state via the execution after *stmt*.

Algorithm 1 first initializes *job* set *wl* with the first argument, and then initializes two mappings *before* and *after* which map each statement to the corresponding configurations that have been calculated before and after that statement so far (lines 1-2). Next, Algorithm 1 loops on the *job* set *wl* until the mappings *before* and *after* reach a fixed point (lines 3-20). After popping a $job(stmt, cs)$ from *wl*, the algorithm updates the *before(stmt)* with configuration set *cs* (lines 4-5). Then, Algorithm 1 computes successor configuration set cs' by performing transitions on configurations in *cs* with shadows at *stmt* (lines 6-9). Next, the algorithm filters out the configurations that have been calculated in the former iterations (line 10). If the resulting *job* set cs_{new} is non-empty, Algorithm 1 updates the *after(stmt)* and propagates cs_{new} to the successor statements of *stmt* in *m*'s intra and inter procedural control-flow graph (lines 11-20). In line 17, if *stmt* is a method call statement, $relatedShadows(stmt)$ denotes all the shadows that can be transitively reached via *stmt*. Otherwise, *stmt* is an exit statement of *m*, and $relatedShadows(stmt)$ denotes all the shadows in other methods. $RelatedShadows(stmt)$ does not contain the shadows in *m* in both cases. $Reaching(cs_{new}, relatedShadows(stmt))$ calculates all configurations that can be obtained from the configurations in cs_{new} by executing shadows in $relatedShadows(stmt)$) in any order.

3. Motivating Examples

We motivate our three optimizations for NSA separately through three separate examples. We also use the "ConnectionClosed" property in Fig. 1 as the typestate property. Figure 3 shows an example that invokes the "*close*" and "*write*" methods of the *Connection* class. The shadows at line 7 and 10 violate the typestate property, because they can both drive the state machine into the final state. The "*close*" operation at line 8 is between these two violating shadows. Hence, from the semantics of the program and the property FSM (*c.f.* Fig. 1), the runtime monitor does not need to monitor the shadow at line 8. Whereas, the original NSA cannot identify the shadow at line 8 as a "nop shadow" at compile-time. The reason is explained as follows.

For the sake of brevity, Fig. 3 only shows partial critical state information calculated by the forward and backward analysis. We denote C as the variable defined in FSM and assume that the program creates static objects O_1 and O_2 at line 2 and line 3 at compile-time respectively. The variable binding of the shadow at line 8 is $C = \{O_2\}$, and the shadow at line 8 changes the configuration from $S_8 = (\{2\}, C = \{O_2\})$ to $T_8 = (\{1\}, C = \{O_2\})$ in the forward analysis. $F_8 = (\{2\}, C \neq \{O_1\})$ associated to the shadow at line 8 is one of the resulting configurations produced by the backward analysis starting at line 9, which means the typestate of the object does not change if the object is not O_1. The variable bindings of S_8 and F_8 are both compatible to that of the shadow at line 8. According to the "nop shadow" conditions, because the states of the state transition caused by the shadow at line 8, *i.e.*, state 2 in S_8 and state 1 in T_8, are not both contained in the state set {2} of F_8, NSA fails to identify this shadow as a "nop shadow".

Actually, the configuration F_8 is induced by the "final" shadow at line 9, in which the variable c_1 is totally unrelated to the variable c_2 at line 8, *i.e.*, they must not alias. Hence, in principle, we should not consider the configuration F_8 when checking whether the shadow at line 8 can be removed. Based on this insight, our optimized NSA can successfully filter this type of interferential configurations generated from backward analysis and identify more shadows, similar to the shadow at line 8, as "nop shadows" in some situations.

Figure 4 shows another example to motivate the second optimization approach. Different from the former one, the typestate property "ConnectionClosed" is not violated by the method *m*. Therefore, all the shadows in method *m* should be safely removed. However, all the shadows remain after using the original NSA. Figure 4 shows partial forward and backward analysis results that are next to the

[†]"Final" shadow, which can drive the property FSM into a final state [4].

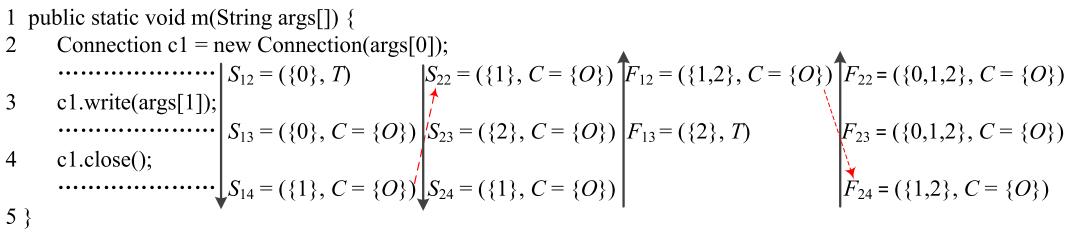

Fig. 4 The example for optimization based on local object information.

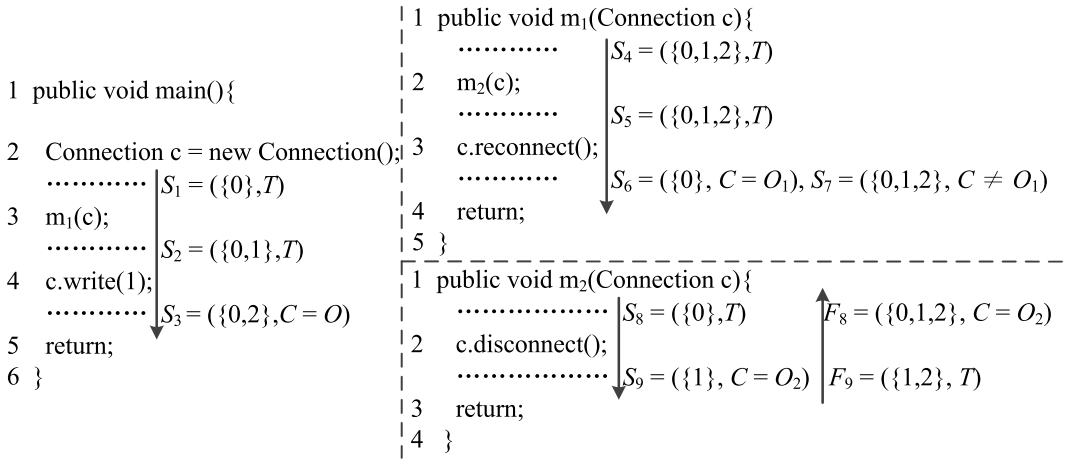

Fig. 5 The example for demonstrating partial inter-procedural flow-sensitive analysis.

two downward arrows and two upward arrows, respectively. Because there may be several consecutive method calls to a method in a program, for ensuring the soundness, the forward analysis needs to propagate the configuration at the end of a method to the entry of the method until a fixed-point is reached. For example, S_{14} is propagated to the entry configuration S_{22} of the next iteration (indicated by the red dotted line). The propagation also happens in backward analysis. After reaching the fixed-point, the shadow at line 3 can produce the configuration S_{23}, which contains an error state. Thus, this shadow cannot be removed. In addition, the shadow at line 4 changes the configuration S_{13} to S_{14}, but state 0 in S_{13} and state 1 in S_{14} are not both contained in the state set {1, 2} of the configuration F_{24}. Therefore, the shadow at line 4 cannot be removed either.

After carefully analyzing the example program, we find the reason is that the configuration propagation disregards the *local object* information. In this paper, we call a static object, which is created by a "*new*" statement within the method currently being analyzed, a *local object*. For example, the static object O created by the statement at line 2 is a local object. Obviously, each local object will be assigned with a different runtime object each time when the method is invoked at runtime. Therefore, for the example in Fig. 4, the configuration S_{22} should not have the same variable binding as S_{14}. If we have the local object information of the example program, *i.e.*, no need to do the second forward and backward iteration, then both of the "nop shadows" at line 3 and line 4 can be removed. Based on the observations motivated by this example, we optimize NSA by exploiting local object information.

The example in Fig. 5 demonstrates the third optimization for NSA. Different from the above two examples, this example program involves three relevant methods. Before invoking the "*write*" operation on the *Connection* Object pointed to by variable c in the *main* method, the program always call the method m_1 in which the operation "reconnect" on the same object is invoked. Clearly, the programm doesn't violate the "ConnectionClosed" property and all the shadows in the three methods should be removed. But the original NSA can not remove any shadow in this example.

The imprecision of the original NSA is mainly caused by the overly conservative inter-procedure data-flow analysis. For example, when a configuration is propagated through a method call statement during analysis, the original NSA can only obtain all the shadows in other methods that can be transitively reached via this method call statement. But, it cannot determine the orderings of these reachable shadows in control flows. In order to avoid jeopardizing soundness, the original NSA assumes that all these reachable shadows can appear in any order at runtime [4]. As shown in Fig. 5, the method call statement at line 3 in *main* method transforms the configuration $S_1 = (\{0\}, T)$ (T means that this configuration is compatible to any shadow) into $S_2 = (\{0, 1\}, T)$ in the forward analysis. We assume that the *main* method creates static object O at line 2. Because the shadow at line 4 in *main* method changes the configuration S_2 into $S_3 = (\{0, 2\}, C = O)$ which contains an error

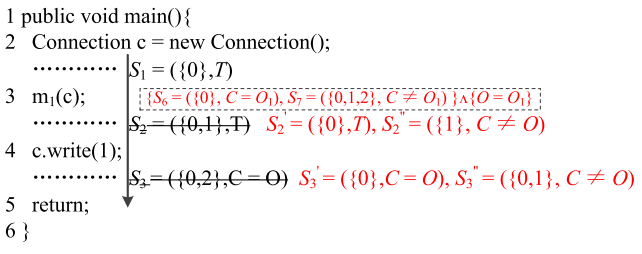

Fig. 6 The example for refining method call statement.

state, this shadow cannot be removed. Based on the same argument, the shadows in methods m_1 and m_2 cannot be removed by the original NSA either.

Let us assume that the formal parameter of method m_1 points to static object O_1 at compile-time. Figure 5 shows all the configurations (*i.e.*, $S_6 = (\{0\}, C = O_1)$ and $S_7 = (\{0,1,2\}, C \neq O_1)$) associated to the statement at line 3 in m_1 in the forward analysis, which means that if the current state of *Connection* object is state 1 or 2 when returning from the method m_1, this object must not be O_1. Additionally, the static object O is passed as the actual parameter to m_1 at line 3 in the *main*, so static objects O and O_1 must be the same object. Based on the above information, we can get the conclusion that the method invocation statement at line 3 in *main* cannot transition the state from 0 into 1 on object O in the forward analysis. Therefore, we refine the configuration S_2 into $S_2' = (\{0\}, T)$ and $S_2'' = (\{1\}, C \neq O)$ (see Fig. 6). Therefore, the shadow at line 4 in *main* method is not compatible to the new configuration S_2'', and it cannot drive the object O into error state and can be safely removed. After removing this shadow, the shadows in method m_1 and m_2 can be easily removed too. Inspired by this example, we can draw the following optimization: for a method call statement *st* which invokes method *m*, although we cannot determine the full orderings of shadows that can be reached via *st*, we can use the original NSA analysis results of *m* to refine the inter-procedural data-flow analysis induced by *st*.

For simplicity, the above motivating examples do not contain complex programming language features, such as recursion, exception handling and aliasing. In Sect. 4, we will give the details of our optimization approaches that can be applied in general.

4. Optimization Approaches

This section presents the details of our optimization approaches which also apply the conditions described in Sect. 2 to identify "nop shadow" as the original NSA. However, our optimizations make use of program structure information to improve the precision of the original NSA. Specifically, our first two optimizations prune irrelevant configurations that can otherwise prevent "nop shadow" from being identified. The third optimization refines the configurations produced by inter-procedural analysis to identify more "nop shadow". These three optimizations are complementary to each other. They address different aspects of the issues separately that can potentially lead NSA to lose precision. Therefore, they can be combined together to further improve the precision of the original NSA.

4.1 Identifying Changeless Configurations

How can we identify changeless configurations, like F_8 in Fig. 3, from the results produced by backward analysis? Basically, if the states of a configuration have never been through a state transition during backward analysis, then we consider the configuration as a *changeless* configuration. For a changeless configuration $C_i = (Q_i, b_i)$ that is induced by a "final" shadow s_f and associated to a shadow s_i, even if C_i is compatible with s_i, there is no need to consider C_i when checking whether the shadow s_i is a "nop shadow". The reason is: the states in set Q_i of the objects specified by the variable bindings b_i will definitely not change anymore before program execution passes the "final" shadow s_f, and the execution of the "final" shadow s_f would not trigger an error either because of the incompatibility of s_f and C_i.

Hence, we extend the original configuration tuple from (Q, b) to (Q, b, T), where Q is the state set, b is the variable bindings and T indicates whether the states of this configuration have ever been through a state transition before. Therefore, we need to record the information of T during the configuration transitions in backward analysis. The new configuration transition algorithm is displayed in Algorithm 2.

Algorithm 2 *transition*$((Q_c, b_c, T_c), s, \delta)$

1: $cs := \emptyset$; // initialize result set
2: $l := label(s), \beta_s := shadowBinding(s)$; //extract label and bindings from s
3: //compute target states
4: $Q_t := \delta(Q_c, l)$;
5: //compute configurations for objects moving to Q_t;
6: $\beta^+ := \underline{and}(b_c, \beta_s)$;
7: **if** $\beta^+ \neq \perp$ **then**
8: $\boxed{cs := cs \cup (Q_t, \beta^+, true);}$
9: **end if**
10: //compute configurations for objects staying in Q_c;
11: $B^- := \bigcup_{v \in dom(\beta_s)} andNot(b_c, \beta_s, v) \setminus \{\perp\}$;
12: $\boxed{cs := cs \cup \{(Q_c, \beta^-, T_c) \mid \beta^- \in B^-\};}$
13: **return** cs;

The algorithm is basically the same as the original one [4]. The different parts are enclosed in boxes. Line 4 computes the state set of the successor configuration. If the shadow s can drive the state set Q_c into Q_t (*c.f.* line 4), and the shadow is compatible to the configuration (determined by $\beta^+ \neq \perp$ at line 7), the value of T in successor configurations is assigned with *true*, indicated by lines 7-9; otherwise, the value of T remains the same during the configuration transition (lines 11-12). Moreover, the value of T in the initial configurations is set to *false*.

In fact, as shown in Algorithm 2, both algorithms calculate the state set Q and variable binding b in the resulting

configurations in the same way. The new algorithm only reserves some extra state transition information at line 4 and 12, compared to the original one. Therefore, our new algorithm is just a simple extension of the original one, and doesn't change the inner workings of the original algorithm. Based on the extra information provided by the new algorithm, our first optimization then prunes changeless configurations that can otherwise prevent "nop shadow" from being identified. Specially, for a given shadow s and a configuration (Q, b, T) in $futures(s)$, if T is *false*, the configuration will be considered to be interferential, and should be filtered out when checking whether the shadow s is a "nop shadow".

4.2 Exploiting Local Object Information

First, we present how to determine whether a static object is a *local object* to a given method. We have the following two observations: first, for a given static object inside a method, if it is created by a "*new*" statement within the method, the object must be a *local object* to this method; second, for any two *strong must-alias* [24] static objects O_1 and O_2 inside a method, these two objects always refer to the same heap object, which implies that they always point to the same local object or the same non-local object. Based on these two insights, we design Algorithm 3 to identify local objects.

Algorithm 3 isLocalObject(m, O)

```
1:  Set⟨staticObject⟩  newObjects;
2:  for all stmt ∈ m do
3:      if stmt is a new statement then
4:          create a new static object O_i;
5:          newObjects := newObjects ∪ {O_i};
6:      end if
7:  end for
8:  for all O_j ∈ newObjects do
9:      if O_j must-alias O then
10:         return true;
11:     end if
12: end for
13: return false;
```

For a given method m and a static object O, the algorithm returns *true* if O is a local object to method m; otherwise returns false. Algorithm 3 first declares a set *newObjects*, and then adds all the local objects created by the "*new*" statements in m to *newObjects* (lines 1-7). Then, the algorithm checks whether there exists an element in *newObjects* that is *strong must-alias* to O. If that is the case, the algorithm returns *true* and terminates running (lines 8-12). Currently, the must-alias analysis is only intra-procedural flow-sensitive, and we make a sound assumption that any two static objects coming from different methods may alias. In order to gain more efficiency, we can extract lines 2-7 from Algorithm 3 and compute the *newObjects* set before performing the optimizations.

Besides identifying local objects, for a configuration, we also need to know whether it is gotten by statically modeling the multiple consecutive invocations of the analyzed method in forward or backward analysis. Similar to the first optimization, we also extend the original configuration tuple to (Q, b, R), where R is a boolean variable. If the current configuration is indeed gotten by statically modeling the multiple consecutive invocations of the method being analysed, R gets assigned *true*.

As shown in Fig. 2, for a given method m, the intra-procedural control-flows cannot lead to the multiple consecutive invocations of m. Hence, the shadows in m cannot change the value of R. Obviously, there are only three types of inter-procedural control flows (*solid* arrows (1), (2) and (4)) in Fig. 2, which can lead to multiple consecutive invocations of the method m. Therefore, for all the configurations that reach the entry statements of m or the recursive call sites within m along these inter-procedural control flows, we should assign *true* to R in these configurations in forward analysis. Based on the same argument, the value of R in configurations, which reach the exit statements of m or the recursive call sites within m along these reverse inter-procedural control flows, should be assigned *true* in backward analysis. Furthermore, for each initial configuration, the value of R is set to *false* in both forward and backward analysis.

Based on Algorithm 3 and the extended configuration, we can filter interferential configurations as follows: for a given shadow s in method m, if there exists a variable v in the variable bindings of s pointing to a local object, any configuration (Q, b, R) in $source(s)$ or $futures(s)$ can be safely eliminated if R is *true*. The reason is: even if the shadow s and the configuration have the same static variable binding with respect to the variable v, v will definitely point to a different new object during each invocation of m at runtime, which means that the shadow s and the configuration are not actually compatible at runtime. After eliminating all the interferential configurations from $sources(s)$ and $futures(s)$, we use the remaining configurations to determine whether the shadow s is a "nop shadow" according to the conditions in Sect. 2.

4.3 Refining Inter-procedural Data-flow Analysis

4.3.1 Combining Analysis Results

The original NSA is only intra-procedural, not inter-procedural, flow-sensitive. When a configuration set cs reaches a given method call statement st which invokes method m, the original NSA will compute the configuration set $TranConfigs(cs, st)$ which contain all the configurations that can be reached from cs by executing 0 or more shadows, transitively reachable via st, in any order [4]. Obviously, the set $TranConfigs(cs, st)$ is a very conservative approximation and includes some unreachable configurations during runtime. Additionally, the original NSA performs intra-procedural flow-sensitive analysis on the method m, and we define $ExitConfigs(m)$ as the set of configurations that are associated to exit (in forward analysis)/entry (in backward analysis) statements or recursive call statements

within m. In other words, the set $ExitConfigs(m)$ contains all the configurations that may be reached when the program just leaves from method m. In Fig. 5, for example, $ExitConfigs(m_1) = \{S_6, S_7\}$ in forward analysis. Therefore, we can refine the inter-procedural data-flow analysis with $ExitConfigs(m)$. We denote $Insect(cs, st)$ as the resulting configuration set after refinement. Therefore, each of the configuration $G_1(Q_1, b_1)$ in $Insect(cs, st)$ should satisfy the following two conditions:

- $\exists G_2(Q_2, b_2) \in TranConfigs(cs, st)$ such that $Q_1 \subseteq Q_2 \wedge b_1 \subseteq_{\overline{\mathbb{B}}} b_2$[†], and
- $\exists G_3(Q_3, b_3) \in ExitConfigs(m)$ such that $Q_1 \subseteq Q_3 \wedge b_1 \subseteq_{\overline{\mathbb{B}}} b_3$.

A variable binding b is defined as $b = (b^+, b^-)$ where both b^+ and b^- are binding functions which map free variables defined in FSM to set of static objects [2], [4]. For a variable v, $b^+(v)$ and $b^-(v)$ represent the static objects that v may and must not point to, respectively. In this paper, we introduce a new operator $*$ on variable bindings, and define $b = b_1 * b_2$ as follows: for any variable v defined in FSM, $b^+(v) = b_1^+(v) \cup b_2^+(v)$ and $b^-(v) = b_1^-(v) \cup b_2^-(v)$. If variable bindings b_1 and b_2 are not compatible, we denote this case by $b = \perp$, which means that b can not be bound to any object. Obviously, the $*$ operation complies to the following properties:

- $b \subseteq_{\overline{\mathbb{B}}} b_1 \wedge b \subseteq_{\overline{\mathbb{B}}} b_2$
- $\forall b'$ such that $b' \subseteq_{\overline{\mathbb{B}}} b_1 \wedge b' \subseteq_{\overline{\mathbb{B}}} b_2 \Rightarrow b' \subseteq_{\overline{\mathbb{B}}} b$

In other words, variable binding b is the "biggest" variable binding that is compatible to both b_1 and b_2. Based on the notion of the above operation, we further define a new operator $\bigstar$ on configurations as follows.

$$(Q, b)\bigstar(Q', b') = (Q \cap Q', b * b')$$

Now, we can safely define the configuration set $Insect(cs, st)$ as:

$$Insect(cs, st) = \{C_1 \bigstar C_2 \mid C_1 \in TranConfigs(cs, st), C_2 \in ExitConfigs(m)\}$$

Informally, $Insect(cs, st)$ contains all the common states of common objects which are implied by both configuration set $TranConfigs(cs, st)$ and $ExitConfigs(m)$.

4.3.2 Inter-procedural Points-to Must-alias analysis

As shown in Fig. 6, if we do not know that the static objects O in method *main* and O_1 in method m_1 must alias, we will get the configurations $S_2^\sharp = (\{1\}, C \neq O_1)$ and $S_3^\sharp = (\{2\}, C = O \wedge C \neq O_1)$ instead of $S_2'' = (\{1\}, C \neq O)$ and $S_3'' = (\{0, 1\}, C \neq O)$ at line 3 and line 4 respectively. Because the $S_3^\sharp$ contains an error state, the shadow at line 4 in *main* method can not be removed. However, the original NSA computes must-alias information on intra-procedural level only. In fact, the actual and formal parameters of a method invocation may serve as springboards to exploit aliasing relationships between the objects coming from the caller and the corresponding callee. Inspired by this insight, we designed the Algorithm 4 to determine whether two static objects coming from a caller and the corresponding callee are must-alias during the execution of the caller.

Algorithm 4 isInterProceduralMustAlias(O, O_1, R)

```
1: for all R_i = (A_i, F_i) ∈ R do
2:     if A_i must-alias O && F_i must-alias O_1 then
3:         return true;
4:     end if
5: end for
6: return false;
```

Without loss of generality, we make the following assumptions:

- Method *mCaller* invokes method *mCallee* at statement *st* in its method body;
- Static objects O and O_1 come from method *mCaller* and *mCalee* respectively;
- A_i and F_i denote the objects pointed to separately by the ith actual and formal parameter of the method invocation at the statement *st*, and N represents the total number of parameters.

Moreover, we define the binary relation set $R = \{(A_i, F_i) | 0 \leqslant i < N\}$. Algorithm 4 returns *true* if O and O_1 are must-alias on inter-procedural level; otherwise returns *false*. For method invocations in java programs, object references are passed in as method parameters, so the actual parameter and its corresponding formal parameter actually point to the same object. Therefore, based on the intra-procedural flow-sensitive information provided by original NSA, if there exist an element $R_i = (A_i, F_i) \in R$ where A_i and F_i are must-alias to O and O_1 respectively, static objects O and O_1 must alias.

4.3.3 Workflows

For a given program, the workflow of the third optimization includes the following steps:

- **Step 1**: We perform the original forward/backward analysis on every shadow-bearing method, and cache the configurations associated to exit/entry statements or recursive call statements of these methods.
- **Step 2**: Based on the information cached in Step 1, we refine the dataflow information of the currently analyzed method according to the approach presented in Sect. 4.3.1. If we can not identify any "nop shadow" in this method, we perform refinements on the next method; otherwise, we 1) remove a "nop shadow", 2) perform the original forward and backward analyses on

[†]The "inclusion relation" $\subseteq_{\overline{\mathbb{B}}}$ on variable bindings is defined in that literature [4]. $b_1 \subseteq_{\overline{\mathbb{B}}} b_2$ means that any object or group of related objects that can be bound to b_1 can also be bound to b_2.

Table 1 Typestates properties used in our experiments [2].

Property Name	Description
FailSafeEnum	do not update a *vector* while iterating over its elements
FailSafeEnumHT	do not update a *hash* table while iterating over its elements or keys
FailSafeIter	do not update a *collection* while iterating over its elements
FailSafeIterMap	do not update a *map* while iterating over its keys or values
HasNextElem	always call *hasMoreElements* before calling *nextElement* on the same *Enumeration* object
HasNext	always call *hasNext* before calling *next* on the same *Iterator* object
Reader	do not use a *Reader* after its *InputStream* is closed
Writer	do not use a *Writer* after its *OutputStream* is closed

the current method again and modify the cached configurations (because the current method may be recursive), 3) re-iterate the refined analysis on the current method, until we reach a fixed point.

- **Step 3**: If some "nop shadows" have been identified in step 2, we return back to step 1. The reason is that removing a "nop shadow" in one method can alter the context and inter-procedural data-flow information of other methods.

Obviously, the optimized NSA analysis is more complex and time-consuming, compared with the original one. In fact, Clara include other two simple, but relatively efficient, flow-insensitive data-flow analyses, *i.e.*, Quick Check and OSA [4]. Therefore, we can apply these less expensive analyses to the program before skipping back to step 1 from Step 3.

5. Experiments and Discussion

We have implemented our optimizations on the Clara framework and conducted experiments on the DaCapo benchmark suite [20]. NSA cannot support multi-threaded programs. Hence, we ignore the multi-threaded programs *hsqldb*, *lusearch* and *xalan* in the benchmark. Our experiments are based on the experiments of the original NSA [2], [4]. We are only interested in 23 property/program combinations for each of which the original NSA cannot remove all the shadows. These 23 combinations involve 8 typestate properties and 7 programs. Table 1 lists the typestate properties used in the experiments.

In order to make our experimental results more convincible, we also limit the maximum number of configurations to be 15000, which is the same as that of the original NSA. Once the number of configurations computed by our optimized analysis is above the threshold, it will abort the analysis of the current method and process the next one.

We evaluate our optimizations as follows: for each optimization, we carry out NSA first, then use the optimized NSA to further identify "nop shadows". This way of evaluation is *different* from using an optimized NSA directly. Actually, according to our experimental results, under the same configuration limit, using an optimized NSA after original NSA will obtain better results than that of using the optimized NSA directly. The reason is that using each optimized analysis directly generates more configurations than the original NSA and makes it easier to abort analysis of programs.

We conducted all the experiments on a Server with 256GB memory and four 2.13GHz XEON CPUs.

5.1 Experiment Results

To justify the effectiveness of our optimizations, we use the original NSA as the baseline for our experiments. Table 2 shows the results of our optimizations, and the cases on which optimizations have no effect are not listed. The forth column ($\mathbf{O_1}$) shows the number of remained shadows after using the first optimization, *i.e.*, filtering changeless configurations generated from backward analysis. For 4 out of 23 combinations (17.4%), our first optimization can further identify removable shadows after the original NSA. In one case (**FailSafeIterMap/bloat**), the shadows removed by our first optimization are more than the shadows removed by the original NSA.

The fifth column ($\mathbf{O_2}$) of Table 2 shows the number of the shadows that remain after using the second optimization, e.g., exploiting the local object information. For 10 out of these 23 combinations (43.5%), the optimized NSA can further remove shadows after the original NSA. In two cases (**FailSafeEnum/fop** and **FailSafeIter/luindex**), the optimization can remove all the shadows that remain after the original NSA. Hence, the optimized NSA based on local object information can give the static guarantee that the program satisfies the typestate property in each of these two cases. Furthermore, for 5 out of these 10 cases (50%), the shadows removed by our second optimization are more than the shadows removed by the original NSA. Especially, in two cases (**FailSafeEnum/fop** and **FailSafeEnumHT/jython**), the original NSA cannot remove any shadow at all.

Surprisingly, the benefits of our third optimization, *i.e.*, refining inter-procedural data-flow analysis, are not significant for these benchmarks. As shown in the sixth column ($\mathbf{O_3}$) of Table 2, for only 2 out of these 23 combinations (8.7%), the optimized NSA can further remove shadows after the original one. Actually, there are large amounts of configurations generated during the whole process of this optimization. For several cases, such as **FailSafeIter/bloat**, the optimized NSA aborts its analysis before identifying any "nop shadow", because the number of generated configurations exceeds the given fixed quota. In Sect. 5.3, we will discuss the reasons why this optimization is not very effec-

Table 2 Results of the optimized NSA.

Property/Program	N_b	N_a	O_1	O_2	O_3	B_1	B_2	B
FailSafeEnum/fop	5	5	5	0	5	0	0	0
FailSafeEnum/jython	47	44	44	36	44	36	36	36
FailSafeEnumHT/jython	76	76	76	72	76	72	72	72
FailSafeIter/bloat	1010	916	911	905	916	899	905	899
FailSafeIter/chart	158	150	150	120	150	120	120	120
FailSafeIter/jython	119	115	115	105	115	105	105	105
FailSafeIter/luindex	30	15	15	0	15	0	0	0
FailSafeIter/pmd	305	290	290	262	287	262	259	259
FailSafeIterMap/bloat	481	479	476	479	479	476	479	476
FailSafeIterMap/jython	153	133	133	119	133	119	119	119
FailSafeIterMap/pmd	372	262	262	260	262	260	260	260
Writer/antlr	44	35	34	35	34	34	34	33
Writer/bloat	19	11	9	11	11	9	11	9

N_b: The number of shadows remained before the original NSA. **N_a**: The number of shadows remained after the original NSA. **O_1**: The number of shadows remained after the first optimization. **O_2**: The number of shadows remained after the second optimization. **O_3**: The number of shadows remained after the third optimization. **B_1**: The number of shadows remained after the combination of the first two optimizations. **B_2**: The number of shadows remained after the combination of the last two optimizations. **B**: The number of shadows remained after the combination of the three optimizations.

Table 3 The results of analysis time (*in seconds*).

Property/Program	The 1st Opt			The 2nd Opt			The 3rd Opt		
	NSA	Opt	Total	NSA	Opt	Total	NSA	Opt	Total
FailSafeEnum/fop	0.98	0.1	251.14	1.21	0.39	276.87	0.98	0.08	252.19
FailSafeEnum/jython	8.06	0.2	243.55	8.23	1.87	269.35	7.5	0.22	242.02
FailSafeEnumHT/jython	10.30	0.69	240.91	10.41	8.19	271.05	10.35	0.71	244.09
FailSafeIter/bloat	298.05	133.94	806.02	288.5	516.82	1214.1	222.12	EX	EX
FailSafeIter/chart	25.35	1.64	305.92	24.15	66.08	393.04	24.55	1.31	313.25
FailSafeIter/jython	16.9	0.74	270.23	17.69	14.87	303.80	17.73	1.66	281.21
FailSafeIter/luindex	2.21	0.07	99.1	2.53	0.47	110.64	2.16	0.07	100.24
FailSafeIter/pmd	46.01	2.51	352.67	46.97	112.01	490.54	44.88	4.41	357.87
FailSafeIterMap/bloat	58.78	30.17	433.4	65.92	85.19	521.37	62.24	EX	EX
FailSafeIterMap/jython	49.67	17.61	276.74	58.38	56	337.73	44.15	EX	EX
FailSafeIterMap/pmd	77.67	4.25	420.05	77.72	96.84	541.66	71.89	3.58	419.04
Writer/antlr	12.95	0.83	223.76	13.7	9.06	253.48	12.27	6.22	229.37
Writer/bloat	1.56	0.24	128.66	1.68	0.34	140.08	1.54	0.1	129.89

NSA: The analysis time that original NSA consumes. **Opt**: The extra analysis time consumed by the optimization. **Total**: The total compilation time of the case. **EX**: Clara throws an exception and aborts analysis, because the number of generated configurations exceeds the given fixed quota.

tive in detail.

The third rightmost column (**B_1**) of Table 2 shows the results of combining the first two optimizations, *i.e.*, optimizing by exploiting local object information first and then by removing changeless configurations. For 13 out of these 23 combinations (56.5%), the combined optimized analysis can further identify "nop shadows" after the original NSA. In one case (**FailSafeIter/bloat**), both the optimizations have positive effects and identify different "nop shadows" respectively. Interestingly, compared to perform these two optimizations after original NSA individually, the combination can identify one more "nop shadow" in this case. The reason is: after the original NSA, the second optimization firstly removes some shadows from the instrumented program, so the first optimization generates less configurations and can identify one more "nop shadow" under the same limit to the number of configurations. In addition, there are three cases where the second optimization cannot remove any "nop shadow" but the other one can, which also justifies that the two optimizations complement each other.

The second rightmost column (**B_2**) of Table 2 shows the results of another combination of the last two optimizations, *i.e.*, optimizing by exploiting local object information first and then by refining inter-procedural data-flow analysis, which can enjoy the benefits of both optimizations. Of course, we can combine the three optimizations together to obtain the benefits of all optimizations.

5.2 Analysis Time

Table 3 displays the analysis time of the cases on which our optimizations have effects. The analysis time of NSA is mainly dominated by the prior supporting analyses, such as constructing call graphs and computing points-to information, and we evaluate each optimization by running original NSA first and then the optimized one. Therefore, the analy-

sis time for evaluating each optimization is definitely longer than that of the original NSA. The original NSA has been run three times which correspond to the three NSA columns in Table 3 respectively. Due to the unstability of the testing environment, the values of the three NSA columns vary in a narrow range is reasonable.

From the experimental results, it can be justified that the analysis time of NSA is just a small part of the total compilation time. The average analysis time of the original NSA is under 1 minute, though it needs several minutes in one case. The total compilation time including our optimization is under 10 minutes, except for the case **FailSafeIter/bloat**. Next, we elaborate upon the extra analysis time consumed by each optimization separately.

In all cases but two, **FailSafeIter/bloat** and **FailSafeIterMap/bloat**, the extra analysis time incurred by the first optimization is under 30 seconds. For the second optimization, the extra analysis time is also relatively few - an average of 75 seconds. The worst case is **FailSafeIter/bloat**, and the corresponding extra analysis time does not exceed 10 minutes. To our surprise, the extra time induced by the third optimization is under 10 seconds for each case. The extra analysis time incurred by our optimizations is determined by the following two factors: 1) the number of shadows remained after the original NSA. Obviously, if there are more shadows in the monitored program, our optimized NSA will produce more configurations during the analyses and consume more time to identify a "nop shadows". 2) the number of "nop shadows" removed by our optimizations. After removing every single "nop shadows", the optimized NSA has to re-iterate the forward and backward analysis over the monitored program. Therefore, removing more "nop shadows" means performing more iterations and consuming more time. For those worst cases, such as **FailSafeIter/bloat** and **FailSafeIterMap/bloat**, the monitored programs contain more than 400 shadows after the original NSA, and some "nop shadows" can be removed respectively by our optimizations. Hence, the extra analysis time of those cases is much longer than that of other cases.

Additionally, for the optimized NSA with the combination of optimizations, the extra analysis time is under 2 minutes in the majority of cases. Overall, our optimization methods do not cause a significant overhead in the weaving process in our experiments. Considering the total compilation time, the overhead incurred by our optimizations is acceptable.

5.3 Discussions

According to the experimental results, the first optimization only has effects on 17.4% cases, which is not very impressive. The reason is that this optimization works well on the methods containing several *interleaved* relevant operations on different objects. For example, in Fig. 7 (a), the program is slightly different with that in Fig. 3 (the method calls on c_1 and c_2 are not interleaved), the original NSA can identify the shadow at line 9 as a "nop shadow". Hence, the capability of the optimized NSA is the same as that of the original one in this situation.

```
1  public static void main(String args[]) {
2      Connection c1 = new Connection(args[0]);
3      Connection c2 = new Connection(args[0]);
4      c1.close();
5      c1.write(args[1]);
6      c1.write(args[1]);
7      c2.close();
8      c2.write(args[1]);
9      c2.close();
10     c2.write(args[1]);
11 }
```

(a) A program similar to the program in figure 3

```
1  public void m(String args[]) {
2      for(int i = 0; i < 10; i++)
3      {
4          Connection c1 = new Connection(args[0]);
5          c1.write(args[1]);
6          c1.close();
7      }
8  }
```

(b) An example similar to the example in figure 4

Fig. 7 Examples on which optimizations have no effect.

The optimization based on local object information has limitations too. For example, it has no effects on the local objects created within loop statements. In Fig. 7 (b), we show a method *m* that extends the method in Fig. 4 by adding a *for* loop. Obviously, the method satisfies the typestate property in Fig. 1, but the optimized NSA based on the local object information cannot remove the shadows at lines 5 and 6. The key reason is that the forward\backward analysis 1) assumes that the static objects created at line 4 during each iteration may be the same object, and 2) propagates the configurations at the end\entry of the "*for*" statement to the entry\end of the "*for*" statement. Therefore, we can further optimize NSA based on the local objects created in loop statements, which will be our future work.

For the third optimization, it is only effective on two cases. We dissected the whole process of this optimization and experimental results to learn why it fails in the majority of cases.

- First, as mentioned in Sect. 5.1, our optimization is prone to abort the analysis because of explosion of the generated configurations, especially for the cases where there are lots of shadow-bearing methods and method invocations. Actually, in our implementation, we only refine the forward analysis to significantly reduce the number of the generated configurations.
- Second, the third optimization is not fully inter-procedural flow-sensitive, and it lacks the context information of the currently analysed method, which is essential for the fine-grained inter-procedural flow-sensitive analysis. For example, the program in Fig. 8 (a) is the same as that in Fig. 5, except for a method invocation of "*write*" on a *Connection* object in m_2. Although our optimization can also identify the shadow at line 4 in *main* as a "nop shadow", it fails to remove other unnecessary shadows in this example. The reason is explained as follows: when our optimized NSA performs data-flow analyses on the method m_1, it does not have the accurate context information whether the method m_2 has been executed or not when the program reaches the entry of method m_1. Hence, the shadow at line 11 in m_1 can not be removed. Based on the same argument, the shadows at lines 16 and 17 in m_2 cannot be removed either.
- Finally, our inter-procedural must-alias points-to anal-

```
1  public void main(){
2    Connection c = new Connection();
3    m1(c);
4    c.write(1);
5    return;
6  }
7
9  public void m1(Connection c){
10   m2(c);
11   c.reconnect();
12   return;
13 }
14
15 public void m2(Connection c){
16   c.write(1);
17   c.disconnect();
18   return;
19 }
```

(a)

```
1  Connection c = new Connection();
2  public void main(){
3    m1();
4    c.write(1);
5    return;
6  }
7
9  public void m1(){
10   m2();
11   c.reconnect();
12   return;
13 }
14
15 public void m2(){
16   c.disconnect();
17   return;
18 }
```

(b)

```
1  public void main(){
2    Connection c = new Connection();
3    m1(c);
4    for(int i = 0; i < 100; i++){
5      c.write(1);
6    }
7    return;
8  }
9
10 public void m1(Connection c){
11   m2(c);
12   c.reconnect();
13   return;
14 }
15
16 public void m2(Connection c){
17   for(int j = 0; j < 100; j++){
18     c.disconnect();
19   }
20   return;
21 }
```

(c)

Fig. 8 Three examples similar to the program in Fig. 5.

ysis can only process the static objects coming from callers and the corresponding callees respectively. For example, the program in Fig. 8 (b) is slightly different from that in Fig. 5. It uses a shared global variable, instead of method parameters, to pass object between callers and the corresponding callees. Our points-to analysis can not determine that the shared global variable *c* in these methods must point to the same static object. Therefore, our optimization cannot remove any shadow contained in this case.

Indeed, our third optimization is not very impressive, but removing a "nop shadow" may yield significant speed-ups during runtime monitoring in some special situations. For example, in Fig. 8 (c), the third optimization can further remove "nop shadow" in loop statements in method *main* and m_2 after the original NSA. In fact, existing inter-procedural flow-sensitive analyses are generally ineffective for identifying "nop shadow" [18]. Although our work only makes a little progress, it inspires us to continue to research relevant issues in this field. Moreover, we should point out that even if the original NSA were fully inter-procedural flow-sensitive, it could not remove any shadow in Fig. 3 and Fig. 4 either. This is because the entire lifetime of *Connection* objects in these two figures are only in the scope of the current methods, and shadows in other methods can not change the states of these objects. In other words, the programs in these two figures can not benefit from fully inter-procedural flow-sensitive data-flow analysis.

6. Related Work

In the past decade, researchers paid special attention to typestate analysis of large-scale programs. Lots of static [6], [7], [26], [27], dynamic [9], [23], [28], [29], and hybrid typestate analysis [4], [13]–[17] approaches have been proposed and implemented. In this section, we give a brief description of these approaches related to our work.

6.1 Static Analysis Approaches

Fink *et al.* propose a context-sensitive, flow-sensitive and integrated static typestate verifier [27]. The verifier utilizes a combined abstract domain of typestate and pointer abstractions to improve the precision of alias analysis. Like Clara, their static analysis framework is also designed to be a staged system to improve the scalability and efficiency. However, their approach can only identify potential program points of failure, and cannot produce residual runtime monitors for dynamic verification at runtime. Additionally, their approach cannot be applied to typestate specifications involving multiple interacting objects.

Jaspan *et al.* propose a static checker Fusion [7]. Differing from Clara [19] and Tracematches [23], Fusion makes use of *relationships* to specify constraints across multi-objects. Fusion defines three variants. Specifically, it allows the users to customize their static analysis to be sound, complete, or compromise between trade-off between these two extremes. Fusion only performs intra-procedural analysis on programs to check constraint violations. Hence, Fusion does not have effects on cases where the related operations are scatter over more than one method. Moreover, Fusion can not be used to enforce API protocols dynamically. Michael *et al.* proposes a fully automatic approach to infer API protocols for Fusion, without users specifying temporal specifications [30]. Their work is complementary to Clara.

6.2 Dynamic Analysis Approaches

Allan *et al.* present Tracemathes to dynamically check typestate properties, which are specified by pattern-based language with free variables [23]. Tracemathes extends AspectJ [31] and has been implemented as an extension of the AspectJ compiler *abc* [32]. However, Tracemathes does not optimize the monitored programs at compile-time, and

the monitored programs usually contain lots of unnecessary monitoring instrumentations. Therefore, Tracemathes usually incurs significant runtime overheads, which impedes its applications in practice [4], [29].

Avgustinov *et al.* propose two optimizations to improve the runtime performance of tracematches [33]. One optimization removes unnecessary runtime monitors to prevent memory leaks [23]. The other optimization presents an indexing data structure for partial matches to improve the tracking efficiency. However, the effectiveness of their optimizations is not very impressive in some situations. Additionally, their optimizations only focus on trace specifications, and does not analyse the monitored base programs at compile-time, which differs from Clara. Hence, their optimizations are orthogonal and complementary to our work.

Purandare *et al.* propose two novel approaches to mitigating monitoring overheads by monitor compaction [34]. The main idea of their techniques is to exploit over-lap, *i.e.*, symbols shared among checked properties or common objects referenced by several property monitors, to synthesize *supermonitors*. *Supermonitors* usually perform fewer transitions and updates at runtime. Clearly, their techniques can only be applied to the cases where several typestate properties are checked at the same time or the property involves multiple objects.

6.3 Hybrid analysis approaches

Naeem et al. propose and implement a a hybrid typestate analysis [16] built on Eric Bodden's early work [25], [35]. In order to statically facilitate reasoning and abstraction, they define a lattice-based operational semantics of tracematches. Hence, their static analysis can track individual objects along control-flow paths, and compute the typestate and points-to information simultaneously. They build a control-flow graph (CFG) for the whole program, and propagate configurations along the edges of this graph. In other words, their analysis is context-sensitive and inter-procedural flow-sensitive. However, there are some cases where NSA is more precise than their inter-procedural analysis, due to the more sophisticated context-sensitive points-to analyses applied by NSA [2], [4]. Additionally, like Eric Bodden's early work [25], [35], they also try to identify "nop shadows" only through "shadow histories" computed by forward analysis. Therefore, their static analysis suffers from unsoundness problem too [2]. As literature [4] pointed out, the original NSA can be extended to inter-procedural flow-sensitive easily. But, it has to perform the forward and backward analysis on the whole program again after removing each "nop shadow". Obviously, this kind of full inter-procedural flow-sensitive analysis is very time-consuming and impractical to the large scale programs. Moreover, it can not remove any shadow in Fig. 3 and Fig. 4 either which can removed by the first two optimizations respectively. The third optimization is only partial inter-procedural flow-sensitive, and improves the precision of original NSA in some cases without a significant overhead.

Besides those work, Purandare *et al.* present a cost model for runtime monitoring which explains determining factors of the monitoring overheads and the close relationships among them [17]. Their optimizations for runtime monitoring are guided by the cost model. Dwyer et al. present an algorithm to identify the *safe* regions in the program for reformulating the typestate property analysis [15]. All the instrumentations in a *safe* region are then replaced with an equivalent summary instrumentation, or removed directly. Clearly, after this transformation, the base programs have fewer instrumentations, and runtime monitors perform fewer translations. However, their static analyses have to be conducted on the whole program, which is very expensive and can not be applied to large-scale programs in practice. For the monitoring instrumentations in the loop statements, such as *while* and *for*, their approach not only identifies unnecessary instrumentations that can be removed permanently, but also tries to identify monitoring instrumentations that can be ignored during the partial of loop iterations at runtime [36]. Furthermore, their propose optimizations to improve the runtime performance by dynamically reclaiming unnecessary monitors [17]. Whereas, their hybrid approach may easily lead to significant overheads at runtime for the typestate properties involving multiple interacting objects. When unchecked exceptions happen, their method may produce unsound results.

7. Conclusion

In this paper, we present three optimization methods for NSA to improve its precision. The first two optimizations filter interferential states of objects produced during the data-flow analysis. The third optimization refine the inter-procedural data-flow analysis induced by method invocations. Experiments on the DaCapo benchmark suite justify that our optimizations, in total, are effective in more than half of the studied cases, without a significant overhead. Additionally, we dissect the experimental results and the situations on which our optimizations have no effects.

In future, we plan to evaluate our optimizations on more benchmarks and real-world large-scale programs. Moreover, we will try to perfect our optimizations, so that they can be effective in more situations. For example, we can further optimize NSA based on the local objects created in loop statements for the second optimization, so that the "nop shadows" in Fig. 7 (b) can be identified. In fact, monitoring instrumentations in a loop statement usually execute as the loop iterates, incurring a significant runtime overhead. Hence, we should design a solution to mitigate this issue induced by the instrumentations related to loop statements. For the third optimization, we should design a more sophisticated inter-procedural flow-sensitive points-to analysis to improve its precision. Additionally, we can integrate the existing dynamic optimization approaches into Clara to further reduce the runtime monitoring overheads.

Acknowledgments

This research is supported in part by grants from the National 973 project 2011CB302603, the National NSFC projects (Nos. 91118007 and 61103013), the National 863 projects (Nos. 2011AA010106 and 2012AA011201), National Natural Science Foundation of China (Nos. 61379054, and 91318301), the Specialized Research Fund for the Doctoral Program of Higher Education 20114307120015, and the Program for New Century Excellent Talents in University.

References

[1] R.E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," IEEE Trans. Softw. Eng., vol.12, no.1, pp.157–171, 1986.
[2] E. Bodden, "Efficient hybrid typestate analysis by determining continuation-equivalent states," Proc. International Conference on Software Engineerin, pp.5–14, New York, USA, 2010.
[3] E. Bodden, P. Lam, and L. Hendren, "Clara: a framework for partially evaluating finite-state runtime monitors ahead of time," Proc. International Conference on Runtime Verification, pp.183–197, St. Julians, Malta, 2010.
[4] E. Bodden, Verifying Finite-state Properties of Large-scale Programs, Ph.D. thesis, McGill University, 2009.
[5] S.A. Slaughter, D.E. Harter, and M.S. Krishnan, "Evaluating the cost of software quality," Commun. ACM, vol.41, no.8, pp.67–73, 1998.
[6] K. Bierhoff and J. Aldrich, "Modular typestate checking of aliased objects," Proc. Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp.301–320, Montreal, Quebec, Canada, 2007.
[7] C. Jaspan and J. Aldrich, "Checking framework interactions with relationships," Proc. European Conference on Object-Oriented Programming, pp.27–51, Genoa, Italy, 2009.
[8] E. Bodden, J-lo-a tool for runtime-checking temporal assertions, Master's thesis, RWTH Aachen university, 2005.
[9] F. Chen and G. Roşu, "Mop: An efficient and generic runtime verification framework," Proc. Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp.569–588, Montreal, Quebec, Canada, 2007.
[10] I.H. Krüger, G. Lee, and M. Meisinger, "Automating software architecture exploration with m2aspects," Proc. International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, pp.51–58, Shanghai, China, 2006.
[11] S. Maoz and D. Harel, "From multi-modal scenarios to code: Compiling lscs into aspectj," Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.219–230, Portland, Oregon, USA, 2006.
[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, Aspect-oriented Programming, Springer, 1997.
[13] S. Goldsmith, R. O'Callahan, and A. Aiken, "Relational queries over program traces," Proc. Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.385–402, San Diego, CA, USA, 2005.
[14] M.C. Martin, V.B. Livshits, and M.S. Lam, "Finding application errors and security flaws using pql: a program query language," Proc. Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.365–383, San Diego, CA, USA, 2005.
[15] M.B. Dwyer and R. Purandare, "Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis," Proc. IEEE/ACM International Conference on Automated Software Engineering, pp.124–133, Atlanta, Georgia, USA, 2007.
[16] N.A. Naeem and O. Lhoták, "Typestate-like analysis of multiple interacting objects," Proc. ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, pp.347–366, Nashville, TN, USA, 2008.
[17] R. Purandare, Exploiting Program and Property Structure for Efficient Runtime Monitoring, Ph.D. thesis, University of Nebraska-Lincoln, 2011.
[18] E. Bodden, P. Lam, and L.J. Hendren, "Partially evaluating finite-state runtime monitors ahead of time," ACM Trans. Program. Lang. Syst., vol.34, no.2, p.7, 2012.
[19] "Clara." http://www.bodden.de/clara/
[20] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," Proc. Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp.169–190, Portland, Oregon, USA, 2006.
[21] C. Wang, Z. Chen, and X. Mao, "Optimizing nop-shadows typestate analysis by filtering interferential configurations," Proc. International Conference on Runtime Verification, pp.269–284, Rennes, France, 2013.
[22] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," Proc. International Conference on Compiler Construction (CC), pp.46–60, Warsaw, Poland, 2003.
[23] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble, "Adding trace matching with free variables to aspectj," ACM SIGPLAN Notices, vol.40, no.10, pp.345–364, 2005.
[24] E. Bodden, P. Lam, and L. Hendren, "Object representatives: A uniform abstraction for pointer information," Proc. International Conference on Visions of Computer Science: BCS International Academic Conference, pp.391–405, London, UK, 2008.
[25] E. Bodden, P. Lam, and L. Hendren, "Finding programming errors earlier by evaluating runtime monitors ahead-of-time," Proc. ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.36–47, Atlanta, Georgia, USA, 2008.
[26] M. Das, S. Lerner, and M. Seigle, "Esp: Path-sensitive program verification in polynomial time," Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.57–68, Berlin, Germany, 2002.
[27] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," Proc. International Symposium on Software Testing and Analysis, pp.133–144, Portland, Maine, USA, 2006.
[28] F. Chen and G. Rosu, "Java-mop: A monitoring oriented programming environment for java," Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp.546–550, Edinburgh, UK, 2005.
[29] P.O. Meredith, D. Jin, F. Chen, and G. Rosu, "Efficient monitoring of parametric context-free patterns," Autom. Softw. Eng., vol.17, no.2, pp.149–180, 2010.
[30] M. Pradel, C. Jaspan, J. Aldrich, and T.R. Gross, "Statically checking api protocol conformance with mined multi-object specifications," Proc. International Conference on Software Engineering, pp.925–935, Zurich, Switzerland, 2012.
[31] "aspectj." http://eclipse.org/aspectj/
[32] P. Avgustinov, A.S. Christensen, L.J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "abc: an extensible aspectj compiler," Proc. International Conference on Aspect-Oriented Software Development, pp.87–98, Chicago, Illinois, USA, 2005.

[33] P. Avgustinov, J. Tibble, and O. de Moor, "Making trace monitors feasible," Proc. Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.589–608, Montreal, Quebec, Canada, 2007.
[34] R. Purandare, M.B. Dwyer, and S.G. Elbaum, "Optimizing monitoring of finite state properties through monitor compaction," Proc. International Symposium on Software Testing and Analysis, pp.280–290, Lugano, Switzerland, 2013.
[35] E. Bodden, L. Hendren, and O. Lhoták, "A staged static program analysis to improve the performance of runtime monitoring," Proc. European Conference on Object-Oriented Programming, pp.525–549, Berlin, Germany, 2007.
[36] R. Purandare, M.B. Dwyer, and S. Elbaum, "Monitor optimization via stutter-equivalent loop transformation," Proc. International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), pp.270–285, Reno/Tahoe, Nevada, USA, 2010.

Chengsong Wang is currently a Ph.D. candidate in computer science at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include runtime verification.

Xiaoguang Mao is currently a full professor at School of Computer, National University of Defense Technology, 410073, Changsha, China. He received his Ph.D. degree in computer science from National University of Defense Technology in 1997. His research interests include high confidence software, software development methodology, software assurance, software service engineering, etc.

Yan Lei is currently a Ph.D. candidate in computer science at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software debugging.

Peng Zhang is currently a master degree candidate in computer science at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software debugging.