Leveraging Specifications of Subcomponents to Mine Precise Specifications of Composite Components

Ziying Dai*, Xiaoguang Mao*†‡, Yan Lei*, Liqian Chen*
*College of Computer, National University of Defense Technology, Changsha 410073, China
†Laboratory of Science and Technology on Integrated Logistics Support,
National University of Defense Technology, Changsha 410073, China
‡Corresponding author, email: xgmao@nudt.edu.cn

***Abstract*—Specifications play an important role in many software engineering activities. Despite their usefulness, formal specifications are often unavailable in practice. Specification mining techniques try to automatically recover specifications from existing programs. Unfortunately, mined specifications are often overly general, which hampers their applications in the downstream analysis and testing. Nowadays, programmers develop software systems by utilizing existing components that usually have some available specifications. However, benefits of these available specifications are not explored by current specification miners. In this paper, we propose an approach to leverage available specifications of subcomponents to improve the precision of specifications of the composite component mined by state-based mining techniques. We monitor subcomponents against their specifications during the mining process and use states that are reached to construct abstract states of the composite component. Our approach makes subcomponents' states encoded within their specifications visible to their composite component, and improves the precision of mined specifications by effectively increasing the number of their states. The empirical evaluation shows that our approach can significantly improve the precision of mined specifications by removing erroneous behavior without noticeable loss of recall.**

I. Introduction

In contemporary software development practice, programmers reuse components by invoking their APIs to construct large systems. These APIs often include constraints on the temporal order of method invocations. Take an example of the file usage. A programmer should first *open* a file, then *read* and/or *write* its content, and at last *close* it. Trying to read or write a closed file will cause exceptions to be thrown. Such constraints are often represented as FSMs (finite-state machines) that encode the valid usage of APIs. API specifications are very useful in many software engineering activities. They can aid the generation of test cases [1]. Program verification tools can use them as input to prove the absence of specification violations [2], and program analysis tools can use them to find certain errors [3][4][7].

Ideally, specifications should have been clearly written by software developers before programmers begin to write the program. However, because writing API specification is cumbersome and requires expert knowledge of corresponding APIs, API specifications are often missing, incomplete or out-of-date in practice despite their usefulness. Mainstream object-oriented programming languages provide only informal documentation to support API specifications. To address this problem, specification mining techniques have been developed to reversely mine API specifications from API client programs [5][6][7][8][9][10][11][12]. Unfortunately, FSM models[1] produced by existing mining techniques are often overly general and include much spurious behavior. Specification miners produce imprecise models especially when these models are large and complex [19]. Even for small, two-state FSMs, the false positive rate of the mined results can be high (e.g., 90-99% [7], and 63% when precision[2] and recall[3] are balanced [18]). Overgeneralized models can hamper the effectiveness of the downstream analysis, verification and validation techniques by producing many false negatives and/or positives. To tackle this problem, Gabel et al. [16] try to validate mined specifications by transforming the training program to test the necessity of a mined specification for this program's correctness.

Nowadays, programmers develop complex software systems by utilizing existing components such as the Java system library. In object-oriented programming, *composition* is one of the most common ways to construct new classes from existing ones. Because of their long-term usage and good understanding, existing components usually have some specifications that are either specified by their developers or mined by specification miners, considering the fact that specification mining techniques have made important progress after their more than a decade's development [15][20]. A composite component commonly invokes methods of its subcomponents to perform its functionality. To this end, it must obey specifications of its subcomponents. However, existing techniques mine specifications of components from scratch, ignoring available specifications of their constituent subcomponents. We observe that this is one important cause of the imprecision of specifications mined by existing miners.

In this paper, we specifically focus on the state-based mining of API specifications of object-oriented components (i.e., classes). The state-based specification mining techniques use values of variables to label the model's states during the mining process. To avoid producing too large and specific models, abstract instead of concrete values are used to label states. Unfortunately, choosing an appropriate state abstraction function at the right abstraction level for specification mining is a challenge [8][14]. State-of-the-art approaches adopt the following state abstraction function abs to mine specifications of classes: values of reference fields (objects and arrays) are abstracted to *null* (=*null*) or *not null* (≠*null*), values of

[1]In the paper, we use the terms "specification" and "model" interchangeably.
[2]Precision is the percentage of mined behavior that is correct.
[3]Recall is the percentage of correct behavior that has been mined.

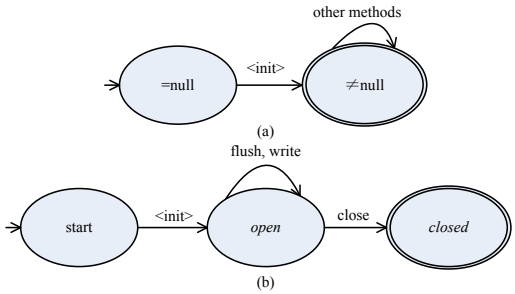

Fig. 1. API specification of an object under the null-abstraction (a), and specification of the `OutputStream` (b).

numerical fields are abstracted to *larger than zero* (>0), *less than zero* (<0), or *equal to zero* (=0), and values of boolean fields remain unchanged. Typical state-based miners include Revolution [29], Crawljax [28], PACHIKA [14], ReAjax [27], and ADABU [8]. Models mined by these tools have already been used to support various testing, debugging and verification techniques [13][14][27][28].

State-of-the-art state-based miners employ the *null-abstraction* for subcomponents that views possibly many different states of a subcomponent as a single state *≠null*. This abstraction is too coarse. It actually assumes that specifications of all subcomponents are the simple two-state FSM of the form presented in Figure 1[4] (a). Such a model specifies that the invocations of instance methods should be made on a created object, and nothing else. Obviously, many important properties of subcomponents are missed from Figure 1 (a). The specification of the composite component mined under this abstraction has a small number of states and much nondeterminism, and can violate the properties of its subcomponents. Figure 2[5] (a) presents the specification for the `BufferedOutputStream` mined by the null-abstraction. This model is almost useless and includes the erroneous behavior that the stream can be written into after it has been closed. Ghezzi et al. claim that specifications mined by such a state abstraction scheme are too imprecise to be used as specifications [17].

In this paper, we argue that exploring existing specifications is beneficial to mining better specifications, and propose to leverage available specifications of subcomponents to mine precise specifications of composite components by using the state-based specification mining approach. Instead of the single *≠null* state of the null-abstraction, we distinguish different states of subcomponent objects encoded in their specifications and use them to construct abstract states of the composite object. When available specifications are finite-state properties, we monitor subcomponent objects against these properties, and use the states that are reached to label states of composite objects during the mining process. In this way, important states of subcomponent objects encoded within their specifications are visible to their composite object, and the number of

[4]Only interesting methods and fields are kept here for brevity. The short arrow denotes that its pointed state is the initial state, and double circled states denote final states. All classes in this paper are from the Java system library and package names are omitted without confusion.

[5]The field *out* is the underlying output stream, the field *buf* is the buffer array, and the field *count* is the number of valid bytes in the buffer. Methods of the `BufferedOutputStream` calls corresponding methods of the *out* to perform its functionality.

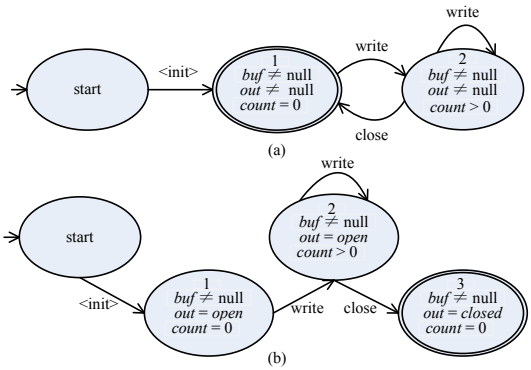

Fig. 2. Mined model for the `BufferedOutputStream` by state-of-the-art state-based specification miners (a), and mined model for the `BufferedOutputStream` by our approach (b).

states of mined specifications of the composite object can be effectively increased. This results in more precise mined specifications. For example, Figure 2 (b) presents the enhanced model of the `BufferedOutputStream` mined by leveraging the specification of the `OutputStream` presented in 1 (b). The `close` of the `BufferedOutputStream` calls the `close` of the `OutputStream` which transits the state of the field *out* from the state *open* to the state *closed* in Figure 1 (b). In this way, the states *open* and *closed* in Figure 1 (b) of the field *out* are visible to the miner. This makes the state *1* and the state *3* in Figure 2 (b) distinguishable and the overgeneralization in Figure 2 (a) is removed.

We empirically validate our approach through comparing specifications mined with and without specifications of subcomponents. We use benchmark programs from the DaCapo benchmark suite [26] as training programs to mine specifications of classes in 17 packages of the Java system library. The evaluation results show that our approach can significantly improve the precision of mined specifications. There are 7 out of 10 overly general FSM models that are enhanced by our approach. In average, 25.05% of the behavior of models mined without considering specifications of subcomponents is erroneous and removed from enhanced models mined by our approach. Meanwhile, no recall is lost for the case of our benchmark programs. Our miner is fast and the overhead introduced by monitoring subcomponents is limited: the time increases are around 10%.

This paper makes the following main contributions:

- We propose an algorithm to explore available specifications of subcomponents to mine more precise specifications of composite components for the state-based specification mining techniques. This algorithm traverses the input trace only once and is scalable.
- We propose a mechanism to coordinate the expressiveness and complexity of mined specifications. Users can provide specifications only for salient subcomponents and our approach utilizes the coarse null-abstraction for other subcomponents.
- We developed a prototype tool and conducted experiments on a large set of traces. The results show that our approach can significantly improve the precision of mined specifications.

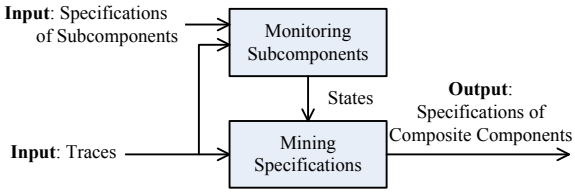

Fig. 3. The specification mining system.

The rest of this paper is organized as follows. Section II discusses our approach in detail. Section III presents the implementation of a prototype tool. Section IV presents the experimental evaluation. Section V discusses related work and Section VI concludes.

II. Approach

In this section, we present our approach in detail. Figure 3 depicts the high-level overview of our approach. The input includes traces of training programs and specifications of subcomponents. Any finite state properties in the form of FSMs can be fed into our approach. These input specifications can be either some available, well-known properties of frequently used libraries such as the resource specification in Figure 1 (b), or mined specifications by existing specification miners. The monitoring of subcomponents and the mining of specifications of composite components proceed concurrently. Subcomponents in the traces are monitored against their specifications and the states that are reached are fed into the specification miner as abstract states for the corresponding subcomponents. The output is mined specifications of composite components.

A. Events and Traces

We distinguish two types of objects in the input traces. The *composite objects* are objects whose specifications we intend to mine, and the *subcomponent objects* are objects that are assigned to fields of composite objects during runtime. A trace $T = \langle e_1, ..., e_n \rangle$ is a sequence of events, where an event e is either a field assignment event fe or a method invocation event me. A field assignment event is a tuple $fe = \langle cn, f, cs, t \rangle$, where a value cn is assigned to the field f of a composite object cs in the thread t. cn is a subcomponent object if f is of reference type. For primitive types (numerical types and boolean types), their concrete values are recorded in field assignment events. A method invocation event is a tuple $me = \langle m, o, me', t \rangle$ with $me' = \langle m', o', me'', t \rangle$, where the method m of the object o (the callee) begins to execute, and m is called by the method m' of the object o' (the caller) in the thread t. We simply say that me is called by me' for brevity. If an event e appears in the trace T, we write $e \in T$. Because we are only interested in objects, calls to static methods are excluded from the trace. So, the receiver object of a method invocation event always exists. We mine specifications of a composite object from the viewpoint of the object's user (caller of the object's public methods), and thus require that $o \neq o'$.

This trace definition is applicable to both single-threaded and multiple-threaded applications. Events are recorded in the order of their occurrence, that is, the order of events is preserved globally. Such global ordering of events can be implemented by a tracing agent based on JVMTI [12]. See Section III for details. In this way, interactions between events coming from different threads can be recognized. Figure 4 presents a trace fragment[6]. Each line corresponds to an event and contains the identifier of the event at the beginning of the line. The event fe_1 assigns the `FileOutputStream` object 647 to the field *out* of the `BufferedOutputStream` object 657. The `close` of the object 657 in me_6 calls the `close` of the object 647 in me_9 to close the wrapped output stream *out*.

me_1. FileOutputSteam.<init>, FileOutputSteam:647, *me*, *main*
me_2. BufferedOutputSteam.<init>, BufferedOutputSteam:657, *me*, *main*
fe_1. FileOutputSteam:647, *out*, BufferedOutputSteam:657, *main*
fe_2. byte[]:658, *buf*, BufferedOutputSteam:657, *main*
me_3. BufferedOutputSteam.write, BufferedOutputSteam:657, *me*, *main*
fe_3. int:1, *count*, BufferedOutputSteam:657, *main*
me_4. BufferedOutputSteam.write, BufferedOutputSteam:657, *me*, *main*
me_5. FileOutputSteam.write, FileOutputSteam:647, me_4, *main*
fe_4. int:0, *count*, BufferedOutputSteam:657, *main*
fe_5. int:1, *count*, BufferedOutputSteam:657, *main*
me_6. BufferedOutputSteam.close, BufferedOutputSteam:657, *me*, *main*
me_7. FileOutputSteam.write, FileOutputSteam:647, me_6, *main*
fe_6. int:0, *count*, BufferedOutputSteam:657, *main*
me_8. FileOutputSteam.flush, FileOutputSteam:647, me_6, *main*
me_9. FileOutputSteam.close, FileOutputSteam:647, me_6, *main*

Fig. 4. Fragment of the trace used to mine the model in Figure 2 (b).

There are no method return events in the trace, but we need to determine when a method invocation exits in some cases. The rule is straightforward: a method invocation exits before its calling method exits; a method invocation exits before a method that appears later in the trace exits if these two methods are called by the same method. Formally, the method invocation event $me_1 = \langle m_1, o_1, me'_1, t \rangle$ exits: (1) just before the first method invocation event me_2 is encountered after me_1 in the trace such that $me_2 = \langle m_2, o_2, me'_1, t \rangle$; or (2) just before me'_1 exits; or (3) when the end of the trace is reached. When there are more than one method invocation events that exit at the same position in the trace, the later the event appears, the earlier it exits. For the example in Figure 4, me_1 exits before me_2 because they are called by the same event me, and me_5 exits before me_4 exits because it is called by me_4. In Section II.C, we give an algorithm that utilizes the data structure of stacks to determine the exit of method invocation events, with the linear complexity to the length of the trace.

B. Monitoring Subcomponents

The specification of a subcomponent can be any FSM with a subset of all public methods of the subcomponent as the input alphabet. Given the specification of a subcomponent, we should monitor its method invocations within a trace to determine its states between method calls in the trace. We create a monitor for each subcomponent in the trace with input specifications. The monitor simulates the trace on the specification FSM, starting from the initial state of the FSM, advancing the FSM by one step when a method invocation of this subcomponent is encountered in the trace. We call the current state in the FSM that is reached as the state of the monitor, and consider it as the abstract state of the subcomponent at the current position in the trace. For example,

[6]We omit signatures of methods for space limit. All `writes` of the `BufferedOutputStream` and all `writes` of the `FileOutputStream` are the same method, respectively.

if we monitor the `FileOutputStream` object 647 in the trace in Figure 4 against the specification in Figure 1 (b), we will get the sequence of state transitions of the object 647 as follows: *start* (me_1) *open* (me_5) *open* (me_7) *open* (me_8) *open* (me_9) *closed*.

As FSM specifications are commonly nondeterministic, more than one states may be simultaneously reached during the monitoring. If the set of all states that have been reached are used as the abstract state of the subcomponent object to label states of the composite object, we need some criterion to identify equivalent states of the composite object. One naive criterion is that if two sets have at least one same state, these two sets can be merged and the resultant abstract state is the union of these two sets. Because it is difficult to choose among various such criteria, we adopt a different solution that is more straightforward. We require that the input specification FSMs of subcomponents are deterministic since deterministic and nondeterministic FSMs are equivalent. If the input specification FSMs are nondeterministic, we first transform them to equivalent deterministic FSMs. In this way, there is a single state that is reached at any time during the monitoring. This single state is used to label states of the composite object. Two states of a subcomponent object are equivalent if and only if they are the same state in the deterministic specification FSM of the subcomponent object.

Although we expect that input specifications of subcomponents are reliable, the requirement of full complete and precise specifications is not practical and will limit the applicability of our approach. In addition, the trace can include erroneous behavior because there may be some events in the trace that violate some subcomponent's specifications but do not cause the program execution to fail. These cases can manifest themselves as violations of input specifications during monitoring. When the monitor encounters a specification violation, we set the current state of the monitored subcomponent object as *≠null* from now on and stop its monitoring while the mining proceeds. In addition, the monitored subcomponent object, the violated specification and the violating trace are logged. Such logs can be used to detect potential bugs in the training program and/or enhance input specifications.

C. Mining Specifications of Composite Components

Mined specifications of our approach are FSM models, each specifying the correct sequences of API invocations of objects of a concrete class. A state of the model represents abstract object states, and a transition represents a method invocation. A state *s* is a vector (v_1, ..., v_n), where each v_i is the abstract state of one of the fields of the class. Two states are equivalent if and only if the corresponding abstract states of fields in their state vectors are equivalent, respectively. We use the same state abstraction function *abs* as that of state-of-the-art state-based specification miners [14][27][28] to compute abstract states of fields. However, instead of the single *≠null* state, the abstract state of a subcomponent object is the state of its monitor that is currently reached. If a field has the value *null* or there are not specifications for the subcomponent, the null-abstraction is used.

Methods of a subcomponent object may be called by a caller that is not its composite object. In such cases, two consecutive method calls of the composite object in the trace may exit (the first method call) and enter (the second method call) in different states of the subcomponent object. This causes problems for labeling states of the composite object. For example, we assume that another object has a reference to the `FileOutputStream` object 647 in the trace in Figure 4 and it calls the `close` of the object 647 between the `<init>` (me_2) and the `write` (me_3) calls of the `BufferedOutputStream` object 657. In such case, there will be an additional event me' appears between fe_2 and me_3 in the trace in Figure 4. me_2 exits with the state of the field *out* as *open*. However, me' transits the state of the *out* from *open* to *closed*, and thus me_3 enters with the state of the *out* as *closed*.

A naive solution is as follows. We define the label of an abstract state of a subcomponent object as a pair $\langle l_1, l_2 \rangle$, where l_1 is the current abstract state of the subcomponent object after a method call of the composite object, and l_2 is the current abstract state of the subcomponent object before the next method call of the composite object in the trace. Two such labels are equivalent if and only if the first states and the second states are equivalent, respectively. If the specification of a subcomponent has *n* different states, there can be n^2 different labels. On one hand, under such a labelling scheme, the miner can produce complex specifications with too many states. On the other hand, we observe that accesses to a subcomponent object from other objects than the composite object are out of control of the composite object, and thus they can vary greatly in different contexts. We adopt a different solution here. When the miner encounters a method call of a subcomponent object from a caller that is not its composite object, the state of this subcomponent object is abstracted to *≠null* and the monitoring of this subcomponent object terminates. This labelling can provide a balance between the complexity and expressiveness of mined specifications.

In a field assignment event $\langle cn, f, cs, t \rangle$ with f of reference type, the subcomponent object cn may be in any of its states. To determine states of cn before and after method calls of the composite object cs, we must begin to monitor cn when it is created. A naive approach is to monitor all objects that are bound to get assigned to a field of composite objects. To decide the future field assignment of subcomponent objects when they are created, we must traverse the trace once before the mining begins and tag these subcomponent objects. Because traces are usually very long and contain numerous events, traversing traces is very time-consuming.

Our first solution to this problem is to monitor all subcomponent objects that have input specifications when they are created. However, because only some of them will be assigned to fields of composite objects in common cases, this approach will cause unnecessary monitoring overhead. Our second fast solution is as follows. We tag every object as *newly created* on its creation. If a method invocation event of a newly created object is encountered, the tag of this object is removed. During mining, when we encounter a field assignment event whose subcomponent object is not *null* and has input specifications, we create a monitor for the subcomponent object if it does not have a monitor but has the tag, or abstract it to *≠null* and do not monitor it if it does not have a monitor and does not have a tag. This fast solution may omit monitoring some subcomponent

objects but can avoid much unnecessary monitoring overhead. These two approaches do not need a pre-traversal of traces.

The algorithm CompositeMiner shows the pseudo code to mine specifications of composite components by leveraging specifications of subcomponents. The input includes target types of objects of which we intend to mine specifications, a trace, and a set of specification FSMs of subcomponents. The input of target types are optional. If they are not specified, the miner infers specifications of all objects in the trace. The output includes a set of FSMs, each for a target type of composite objects, and possibly some logs of violations of input specifications. In this algorithm, we use the notation $f.t$ to denote the type of the field f, abs to denote the state abstraction function. For brevity, we write $abs(o)$ to denote the abstract state of the object o gained by applying the state abstraction function abs on the concrete value of each of the fields of o. The function $t : \{objects\} \rightarrow \{types\}$ is to map an object to its type. The initialization of variables is omitted from the algorithm for space limit. The algorithm actually initializes an empty FSM for each composite object when it is first encountered and an empty stack for each thread when it is first encountered. The first and last method invocation events of a composite object in the input trace define the initial and final states of the FSM model of the composite object. We omit these details from the algorithm for brevity. The stack stores pairs $\langle e, tran \rangle$, where e represents a method invocation event and *tran* represents a transition in the FSM model for this event.

The main loop of the algorithm traverses the trace T and processes events one by one in their order. The fast approach to determine subcomponent objects to be monitored is employed in this algorithm. The algorithm can be adapted slightly to incorporate the approach that all subcomponent objects are monitored. For a field assignment event (lines from 13 to 21), if the subcomponent object has input specifications, we conditionally create a monitor for it and update abstract states of its composite object with the current state of the monitor. Otherwise, the null-abstraction is used for the subcomponent object. For a method invocation event, if the receiver o is a target composite object (lines from 2 to 4), we add a transition to SC_o with its source state as the current abstract state of o and its destination state as *null*. Then, we call the procedure Stack to possibly pop previous events from and then push this event into the stack of the thread th. The destination state of this transition is determined by the procedure Stack when this method call exits. If the receiver o is a subcomponent object with input specifications (lines from 5 to 12), we first call the procedure Stack to maintain the stack of the thread th. If there is a monitor for o, we advance the monitor by the method m. If we cannot perform the advancement, a violation of input specifications is encountered. Then we set the current state of o to be $\neq$*null*, destroy this monitor and log this violation for later inspection. After all events in the trace are processed, we call the procedure Stack to pop all remaining events in each of the stacks of the threads and set the destination states of the corresponding transitions (line 22). At last, we unionize FSMs of all objects of the same type to get a single FSM model for the type (lines 23 and 24). The union of two FSMs consists of the union of the states and the union of the transitions of these two FSMs into one FSM.

Algorithm: CompositeMiner

Input:
TYPES = {t_1, …, t_n}: types of composite objects, optional
T: Trace
S = {S_t | *t* is the type of a subcomponent}: deterministic FSMs for subcomponents
Output:
SC = {SC_t | $t \in TYPES$}: specifications for composite objects
L: logs of violations of specifications in *S*

```
1:  foreach e ∈ T in the order in T do
2:      if e = <m, o, me', th> ∧ t(o) ∈ TYPES then
3:          add a transition tran for m from abs(o) to null to SC_o
4:          Stack(e, tran)
5:      if e = <m, o, me', th> ∧ S_t(o) ∈ S then
6:          Stack(e, null)
7:          if the monitor mo exists for o then
8:              advance mo by m
9:              if a violation v encountered then
10:                 abs(o) ← ≠null
11:                 destroy mo
12:             L ← L ∪ {v}
13:     if e = <cn, f, cs, th> then
14:         if S_f.t ∈ S then
15:             if cn has a monitor mo then
16:                 update state of cs with current state of mo
17:             else if cn is newly created then
18:                 create a monitor mo for cn
19:                 update state of cs with current state of mo
20:             else update state of cs with abs(cn)
21:         else update state of cs with abs(cn)
22: Stack(null, null)
23: foreach t ∈ TYPES do
24:     SC_t = union of all SC_o with t(o) = t
25: return SC

    Procedure Stack(event: me = <m, o, me', th>, Transition: tran)
    with me' = <m', o', me'', th'>

26: if s_th is not empty then
27:     let (me_t = <m_t, o_t, me_t', th_t>, tran_t) be top element of s_th
28:     let me_t' = <m_t', o_t', me_t'', th_t'>, tran_t)
29:     if me' = me_t then
30:         s_th.push(me, tran)
31:     else
32:         while m_t' ≠ m' do
33:             s_th.pop()
34:             if tran_t ≠ null then
35:                 set destination state of tran_t as abs(o_t)
36:             if s_th is not empty then
37:                 let (me_t = <m_t, o_t, me_t', th_t>, tran_t) be top element of s_th
38:             else break
39:         if m_t' = m' and s_th is not empty then
40:             s_th.pop()
41:             if tran_t ≠ null then
42:                 set destination state of tran_t as abs(o_t)
43:         s_th.push(me, tran)
44: else s_th.push(me, tran)
45: if me = null then
46:     foreach s_th do
47:         while s_th is not empty do
48:             s_th.pop()
49:             if tran_t ≠ null then
50:                 set destination state of tran_t as abs(o_t)
51:             if s_th is not empty then
52:                 let (me_t = <m_t, o_t, me_t', th_t>, tran_t) be top element of s_th
```

The procedure Stack simulates runtime stacks of the training program and determines when method invocations in the trace exit. For a thread th, we write s_{th} to denote the stack of th. If the stack for the thread of the input event is empty (line 44) or the event at its top is the calling event of the input event (lines 29 and 30), we simply put the input event into the stack. Otherwise, we pop the top event and set

the destination state of the transition. This repeats until we reach the remaining top event of the stack that is called by the calling event of the input event (lines from 32 to 38). We pop this top event (lines from 39 to 42) and push the input event (line 43). When the end of the trace is reached, we pop all remaining events in each of the stacks of the threads and set the destination states of the transitions (lines from 45 to 52). To illustrate the Stack procedure, consider the trace fragment in Figure 4. The order of method invocation events coming into and out of the stack of the *main* thread is as follows: me_1 pushed, me_1 popped, me_2 pushed, me_2 popped, me_3 pushed, me_3 popped, me_4 pushed, me_5 pushed, me_5 popped, me_4 popped, me_6 pushed, me_7 pushed, me_7 popped, me_8 pushed, me_8 popped, me_9 pushed, me_9 popped, me_6 popped.

The algorithm traverses the input trace *T* only once. Every method invocation event is pushed into and popped out of a stack only once, respectively. Each field assignment event is processed directly. So, if *T* has *m* method invocation events and *n* field assignment events, the algorithm has the time complexity of $O(2m + n)$. In contrast, the commonly used *kTail* algorithm [10] and typical PFSA learners [12] have the running time quadratic and cubic to the length of the input trace, respectively

III. Implementation

In this section, we present the implementation of the trace collector and the implementation of the specification miner with the collected traces as input.

A. Trace Collection

To obtain the trace to mine specifications, we used the C programming language to write a tracing agent based on Java Virtual Machine Tool Interface (JVMTI)[7]. JVMTI is convenient to trace programs in many aspects. E.g., it makes it easy to access the call stack and to attach a unique tag to every object. The key benefit of the tracing agent is that for both single-threaded and multiple-threaded applications, events are issued and recorded when they actually occur during runtime, that is, the order of events is preserved globally. The tracing agent is attached to the Java Virtual Machine and writes the flow of events to a plain text file. The tracing agent records three types of events: *method entry*, *method exit* and *field modification*. Table I presents the event types and recorded information. A *method entry* event is issued when a method enters. A *method exit* event is issued when a method exits. A *field modification* event is issued when some value is assigned to a field of an object.

We can configure what events are to be traced by providing a package name through the option to indicate that the tracing agent will record events from all classes in this package. Because we aim to mine API specifications of objects, only *method entry* and *method exit* events of public constructors and public, instance methods are traced. Our tracing agent is based on JVMTI that allows a much less complex and thus less error-prone implementation. The downside of this approach is that the tracing agent incurs significant runtime overhead. However, our specification miner is modular and is not bound to this tracing agent. Any traces that adhere to the trace definition in section II.A can be fed into our specification miner.

[7]http://download.oracle.com/javase/6/docs/technotes/

TABLE I. Types of events and information traced by the tracing agent.

Event Type	Traced Information
Method Entry	The thread name; the stack depth of this invocation; the name and signature of the called method; the type and identifier of the receiver.
Method Exit	The thread name; the stack depth of this invocation; the name and signature of the called method.
Field Modification	The class of the outer object; the identifier of the composite object; the declaring class of the field, the name and type of the field; the new value.

B. Specification Mining

The traces collected by the agent do not strictly adhere to the trace definition in section II.A. The *method entry* event does not contain its calling event. However, the tracing agent records extra *method exit* events that explicitly tell when the called method of a *method entry* event exits. For a *method entry* event, its corresponding *method exit* event is the first *method exit* event after it in the trace such that the threads and thread depths of these two events are equivalent, respectively. For a *method entry* event e, a *method entry* event e' is (directly) called by it if (1) e' is between e and e's corresponding *method exit* event in the trace, (2) e' has the same thread as that of e, and (3) the stack depth of e' is larger by one than that of e.

For a *method entry* event of the constructor of a composite object, we create an `ObjectState` object to represent the abstract state of the created composite object. The `ObjectState` has a corresponding field for each of the fields of the composite object. When a *field modification* event that modifies the field of this object is encountered, we update the corresponding field of the `ObjectState` object with the abstract value of the new value. The `ObjectState` is also updated when the monitors of its subcomponent objects advance to new states. The *field modification* event also captures the initialization of a field at its declaration. In this way, an `ObjectState` object maintains the abstract state of the corresponding composite object. `ObjectState` objects are used to determine the source and destination states of a transition of mined FSM models during the mining process.

IV. Empirical Validation

We conducted a series of experiments to evaluate the effectiveness of our approach to mine better specifications of composite objects and its overhead.

A. Experimental Setup

We applied our approach to mine specifications for classes from three packages and their sub-packages of the Oracle Java JDK 6 system library: `java.lang`, `java.util`, and `java.io`, totally 17 packages. Classes in these packages obey important API specifications and they are widely used as experimental targets in the literature [11][12][13]. Training programs used here are the 11 benchmark programs from the DaCapo benchmark suite 2006-10-MR2 [26], which ensures a controlled and reproducible execution of all benchmarks. Considering that numerous events were produced, the execution time of each program was limited to at most two hours.

TABLE II. INPUT SPECIFICATIONS AND THEIR SUBCOMPONENTS.

Name	Description	Regular Expression [a]	Subcomponents
CollectionItr	Collection should not be changed while being iterated	createIterator(c, i) next$(i)^*$ updateCollection$(c)^+$ next(i)	79
MapItr	Map should not be changed while being iterated over its keys or values	createCollection(m, c) (updateMap(m) | updateCollection(c))* createIterator(c, i) next$(i)^*$ (updateMap(m) | updateCollection(c))$^+$ next(i)	24
Close	Stream should not be used after it is closed	close(s) (read(s) | write(s))	56

[a] Here matches mean violations. For an event $m(p_1, p_2)$, p_1 represents the receiver of the method m, and p_2 if any represents its return. An event here may represent a few different methods with the similar functionality of the class. For example, updateCollection(c) can be add, remove, clear, et al.

All experiments were carried out on a machine of Win7 and 6G RAM, 3.0 GHz Intel Core i5-2320 CPU with the 64-Bit Server VM of the Oracle Java SE 1.6.0_27. We collected the data by repeating each run 10 times and the geometric mean were computed as the final result. In the specification mining experiments, we monitored all subcomponent objects with input specifications.

Bodden [22] used eight well-known finite-state properties of Java system classes in his typestate analysis experiments to determine whether benchmark programs from the DaCapo suite [26] violated these properties. To validate our approach, we used six well-known finite-state properties of Java system classes as input specifications that cover the eight properties used by Bodden [22][8]. After analyzing the traces, we found that only three of them have experienced our approach, that is, there were some objects that obey these properties and were assigned to fields of some composite objects. These three properties and the number of classes in the target packages that obey them are presented in Table II. They are also expressed as regular expressions in Table II for clarity. These properties are clearly stated in the Java API documentation. They are well known in the literature and are often used as subjects for the analysis and verification of finite-state properties [22][30]. In addition, these three properties are also targets of various specification miners. For example, the properties CollectionItr and MapItr can be (partially) recovered by [12][11], and the property Close can be (partially) recovered by [7][31]. We manually created the FSM models[9] for these properties, which did not take much time. The FSM for the Close property is similar to the model in Figure 1 (b). There are more than 150 subcomponents considered, which means our approach can be intensively experienced.

B. Overview of Results

Table III presents the number of events of the trace, the number of mined FSMs, the time cost to mine each trace, and the time increase compared with the null-abstraction approach for each benchmark program. The *total/average* cell of the *time increase* column is the average of the time increases of the 11 benchmark programs. Because different benchmarks may use objects of the same class, we mined several models for a class, each from a benchmark program. In such cases, we unionized these models into one FSM as the final specification for the class. In total, 154 FSMs were mined. A FSM has 3.4 states and 7.4 transitions in average. Our approach is very fast. The analysis time is roughly linear to the length of the input trace. Mining one trace of tens of millions of events typically took around 3 minutes and none of the input traces exceeded 10 minutes. We also implemented the null-abstraction approach for comparison. Compared with the null-abstraction approach, the extra execution time is around 10% with the average as 6.79%. This overhead is acceptable considering the improved precision obtained by our approach.

In the input traces, there are twenty five composite classes whose objects have some field values as the subcomponent objects with input specifications. These twenty five composite classes come from three packages and can be divided into two domains: *I/O* (java.io and java.util.zip) and *Collection* (java.util). To validate mined specifications, we employed two skilled Java programmers to manually inspect the resulting FSM models of these twenty five composite classes mined by the null-abstraction approach and by our approach. The mined models of these twenty five classes can be freely accessed[10]. Each programmer inspected all these models independently and at last compared the inspection results. If there were some conflicts, they performed further inspection to get the coincidence. The main reference of the inspection was the Java documentation and the source code of the target classes. This task took about three days.

The quality of models for these twenty five classes mined by our approach is satisfactory. For example, eighteen out of the twenty five classes are streams. Except for the OutputStreamWriter, the PrintStream and the InputStreamReader, the Close property is successfully recovered for each of them. The completeness of the mined properties depends on the methods called by the benchmark programs. For some of these classes, the mined property is not complete because some methods that consist of this property are not invoked.

C. Enhancing Models: Qualitative Evaluation

Models for fifteen out of the twenty five classes produced by the null-abstraction approach have no overgeneralization[11]. All of these fifteen classes have some direct or indirect state-indicating fields such as the *out* field of the BufferedWriter that is initialized by its constructor and is set to be *null* by its close method. Models for the left ten classes produced by the null-abstraction approach are overgeneralized. Models for three out of these ten classes

[8] Our Close property covers his Reader and Writer properties (all Readers and Writers obey the Close property) and our CollectionItr property covers his FailSafeEnum and FailSafeIter properties (all Collections and Vectors obey the CollectionItr property).

[9] Unlike the regular expressions in Table II that are for the readers to understand these properties, these models encode the valid API usage.

[10] https://sourceforge.net/projects/tsminer/.

[11] All of them exhibit some incompleteness.

TABLE III. OVERVIEW OF BENCHMARK PROGRAMS AND EXPERIMENTAL RESULTS

Benchmark Program	Events	FSM Models	Execution Time (minutes)	Time Increase
antlr	47,598,159	80	2.19	5.29%
bloat	60,267,427	82	5.68	18.70%
chart	59,093,129	145	4.56	4.35%
eclipse	57,351,123	91	4.28	6.29%
fop	38,251,808	89	1.63	4.78%
hsqldb	67,276,475	82	2.62	1.55%
jython	56,232,181	77	8.81	4.45%
luindex	50,191,882	83	4.85	3.19%
lusearch	38,366,642	79	2.03	6.28%
pmd	56,156,929	92	7.98	5.11%
xalan	41,277,812	82	1.82	6.27%
total/average	572,063,567	154	46.45	6.79%

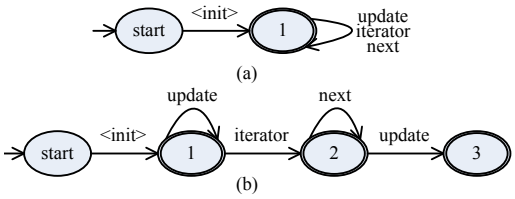

Fig. 5. Model for the `HashSet`: the initial one (a) mined by the null-abstraction approach and the enhanced one (b) mined by our approach.

produced by our approach are the same as that of the null-abstraction approach. Our approach failed to remove the over-generalization of models of these three classes. The reason for this is not the limitation of our approach. As for the `OutputStreamWriter` and the `PrintStream`, we did not observe their `close` invocations during the runs of the benchmark programs. As for the `InputStreamReader`, no specification was provided for its field *sd* because the type of this field is `sun.nio.cs.StreamDecoder` that is beyond the target packages. Models for the left seven of the ten classes are enhanced by our approach. These seven classes are listed in Table IV.

The first six out of the seven classes presented in Table IV are streams and they obey the Close property. For the initial model of each of these classes produced by the null-abstraction approach, if there is a `close` transition that does not end in the *closed* state, our approach exactly adds one additional *closed* state as the destination for this `close` transition. The added *closed* state has no non-`close` incoming or outgoing transitions, which exactly reflects the Close property. One such enhanced model is for the `BufferedOutputStream` presented in Figure 2 (b), compared with the initial model in Figure 2 (a). The model of the `HashSet` was enhanced by our approach through a `HashMap` subcomponent that obeys the MapItr property. To recover the CollectionItr property of the `HashSet`, we added the `next` method of the `Iterator` returned by its `iterator` method to its alphabet of methods[12]. The model mined for the `HashSet` is presented in Figure 5. The `update` method denotes methods that modify contents of the `HashSet`, such as `add`, `remove` and `clear`. The null-abstraction approach produced the useless model in Figure 5 (a) that violates the CollectionItr property. Our approach successfully recovered the CollectionItr property of the `HashSet` by exploring the MapItr property of its subcomponent.

[12]Our approach currently focuses on single object specifications. To recover specifications of multiple objects, the expansion of the alphabet to include methods of multiple objects is requisite.

D. Enhancing Models: Quantitative Evaluation

Although we can easily see that the quality of mined specifications is enhanced by our approach compared with those mined by the null-abstraction approach, we try to quantitatively evaluate the enhancement. In the literature, the measurement of *precision* (the percentage of mined behavior that is correct) and *recall* (the percentage of correct behavior that has been mined) are often used [10][18][19]. After the manual inspection, we observed that no recall is lost for all of the enhanced models compared with their corresponding initial ones for the case of our benchmark programs. This resulted from the fact that our approach just added necessary states to models of composite objects to make them obey specifications of their subcomponents. Theoretically speaking, it is possible for our approach to add redundant states to models of composite objects and thus lead to recall loss. However, we did not observe this in our experiments.

Considering the fact that our approach did not lose recall for these models, we adopted a convenient way to evaluate the precision enhancement. We define the *precision enhancement* as the percentage of the behavior of the initial model that are rejected by the enhanced model. This rejected behavior is erroneous behavior that is incorporated in the initial model but removed from the enhanced model by our approach. To perform this evaluation, we applied the trace generation algorithm TraceGen [19] to randomly generate normal traces from the initial model and then simulated these traces on the corresponding enhanced model. A *normal trace* is a sequence of transition labels (methods) that form a path starting from the initial state to a final state of the FSM model. It represents normal behavior of the object that is accepted by its model. To generate a normal trace, we start from the initial state of the model and randomly choose an outgoing transition to reach the next state. This repeats until we reach a final state. The precision enhancement is the ratio of the number of traces that are rejected by the enhanced model to the number of all generated traces from the initial model.

For example, for the initial model presented in Figure 2 (a) of the `BufferedOutputStream`, we can generate a set of two normal traces that cover all of its transitions at least once:

T_1: `<init>`, `write`, `write`, `close`.

T_2: `<init>`, `write`, `close`, `write`, `close`.

We can see that T_1 represents normal behavior while T_2 represents erroneous behavior. Then we simulate these two trace on the enhanced model presented in Figure 2 (b). T_1 is accepted, while T_2 is rejected. So, for this set of normal traces, the precision enhancement is 1/2 = 50%.

TABLE IV. PRECISION ENHANCEMENTS OF MINED SPECIFICATIONS BY OUR APPROACH.

Subject Class	Enhancement
`BufferedOutputStream`	18.21%
`BufferedReader`	6.13%
`DataInputStream`	4.85%
`DataOutputStream`	45.23%
`FileReader`	26.05%
`FileWriter`	47.83%
`HashSet`	27.02%

We configured the TraceGen algorithm to generate sets of normal traces such that each transition in the initial model had to be covered at least 3 times. We computed the precision enhancement by repeating each experiment 10 times with 10 different sets of normal traces, and the average of the results is presented in Table IV. The results show that our approach can significantly remove overgeneralized behavior from the initial model and thus effectively improve the precision of the mined models. The enhancements for the `DataOutputStream` and the `FileWriter` are significantly high because their initial models have very few (only 2) states. The more overly general (fewer states) the initial models are, the more effective our approach can be. In total, for the 7 enhanced FSM models, our approach gains an average precision enhancement of 25.05%.

V. RELATED WORK

Object-oriented typestate systems are proposed [24][25]. Typestates are intended for specifying correct sequences of method invocations. Because typestates reflect how state changes of objects can affect valid method invocations, a *typestate* is an abstraction of a set of concrete object states and can be characterized by values of all fields of an object. Typestates are mapped onto fields of the class by defining a predicate on field values for each typestate, called a *state invariant*. State-based specification mining techniques are based on these ideas. They try to automatically recover state invariants. Different approaches for state-based mining utilize different state abstraction functions. For example, PACHIKA [14] mines specifications for Java classes by abstracting field values of objects; ReAjax [27] mines a state machine for an Ajax web application by abstracting states of its Document Object Model elements. These state abstraction functions do not consider the possible available specifications of subcomponents and typically use the null-abstraction for subcomponents.

Dallmeier et al. [8][13] propose to mine specifications of Java classes by abstracting values of object fields (or returns of observer methods [8]). Because of the coarse null-abstraction, they can produce overly general models such as those presented in Figure 2 (a) and Figure 5 (a). Our approach explores benefits of available specifications of subcomponents and can mine more precise models such as those shown in Figure 2 (b) and Figure 5 (b). Dallmeier et al. [13] propose to automatically generate test cases to enrich mined specifications. Their approach is effective to increase the number of transitions but has a limited power of discovering new states due to the state abstraction function used. For example, the generated test cases may cover the transition of the `close` from the state *2* to the state *3* in Figure 2 (b), but the mined model is still the one presented in Figure 2 (a). This is because the states *1* and *3* are indistinguishable under the null-abstraction. Our approach complements theirs.

Dallmeier et al. [14] also propose to mine *deep models* that also consider states of transitively reachable objects rather than just a $\neq$*null* state for reference fields. The *depth* parameter is introduced to define the number of considered indirections when including states of deeper reference fields. Models produced under the null-abstraction have the depth of 0 and models with the depth of 1 also consider fields of reference fields of the target object. Reference fields beyond the depth are still subject to the null-abstraction. To discover all useful states of the composite object, this approach depends on the existence of state-indicating fields of the object or those of its descendant fields, which cannot be guaranteed in general. In addition, considering unrelated fields can lead to unnecessary states that complicate mined specifications. In general, we do not know what is the best state abstraction function for specific classes without further knowledge. We believe that there is no best and general state abstraction function even for the primitive numeric types. Moreover, to keep the approach to be scalable, the depth must be small (Dallmeier et al. [14] mine models with the depth of 1), and choosing an appropriate depth for the general specification mining technique is difficult. In contrast, our approach directly leverages the abstraction encoded within available specifications of subcomponents to mine more precise specifications of composite objects.

The Daikon tool [15] can infer program invariants that are boolean expressions on variables and constants at specific program points. Typical program invariants include class invariants, preconditions and postconditions of methods. Although states of subcomponents of composite objects can be considered and abstracted through *derived variables*, invariants are mined from invariant templates, which does not consider existing specifications of subcomponents.

Whaley et al. [6] propose to extract a separate submodel for each class field which consists of only these methods that refer to this field. Their dynamic model extractor records a set of pairs of method calls and do not inspect objects' states. Their static model extractor finds pairs of methods such that calling the second after the first will certainly raise an exception. Inspired by this, Alur et al. present the JIST tool [23] that produces interface specifications for Java classes. The interface specification consists of sequences of method calls that will not reach exceptional states (in which *exceptional predicates* hold). JIST utilizes sophisticated techniques such as predicate abstraction and symbolic model checking to statically reason about exceptional states of the target class. In contrast, our approach works on concrete program executions and tries to generalize concrete state of objects to infer meaningful API specifications. In addition, JIST does not explore benefits from available specifications of subcomponents.

Ghezzi et al. [17] present the SPY approach to mine specifications of Java data container classes. Concrete return values of inspector methods are used to label states of the target object. Specifications mined by SPY cannot be represented by a finite number of FSMs, so they build a set of graph transformation rules to generate specifications intensionally. SPY can mine very precise but at the same time very complex API specifications. Our approach inspects objects' internal states and utilize abstract states of subcomponents encoded within their specifications to mine models of moderate size with balanced precision and recall. Furthermore, our approach

can control the complexity of mined models through input specifications. Simple input specifications tend to produce less complex models of composite objects.

Lo et al. [10] present a steering mechanism to improve the precision of specifications mined by the *kTail* algorithm. They first infer simple temporal properties of two events. Then, they use inferred simple temporal properties to guide the *kTail* algorithm such that two equivalent states are merged only if the merging does not violate any temporal properties. This approach does not consider explicit states of programs. The inferred simple properties may capture some constraints of subcomponents, but more benefits of available specifications of subcomponents are not explored.

Wu et al. [31] propose to mine resource releasing specifications in the form of (*resource-acquiring*, *resource-releasing*) API method pairs from source code and API documentation. Techniques are proposed to mine multiple object specifications [11][12] that can include method invocations from more than one objects. Our approach currently focuses on specifications of single objects. However, this is not the inherent limitation. Our approach can be easily extended to mine specifications of multiple objects.

VI. Conclusions and Future Work

This paper presents an approach to mine precise specifications of composite components by leveraging available specifications of subcomponents. We implemented our approach based on the state-based specification mining techniques. Experiments show that our approach can make important states of composite objects distinguishable and that specifications of subcomponents can be used to effectively improve the precision of mined specifications of composite components. The quality of input specifications of subcomponents may influence mined specifications of composite components. In future work, we plan to evaluate our approach with imperfect input specifications, such as these produced by other specification miners. In addition, we plan to conduct further empirical studies with subjects from other libraries and explore the possibility of leveraging input specifications of other forms, such as the invariants over variables, to improve mined specifications.

Acknowledgment

This work was supported by the National Natural Science Foundation of China under Grant Nos. 91318301, 91118007 and 61120106006, and the National High Technology Research and Development Program of China (863 program) under Grant Nos.2011AA010106, 2012AA011201, and the Program for New Century Excellent Talents in University in China.

References

[1] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. ACM Comput. Surv., 41(2):1–76, 2009.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002, pp. 1–3.

[3] M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. Statically checking API protocol conformance with mined multi-object specifications. In *ICSE*, 2012, pp. 925–935.

[4] M. Pradel and T. R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *ICSE*, 2012, pp. 288–298.

[5] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL*, 2002, pp. 4–16.

[6] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA*, 2002, pp. 218–228.

[7] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *TACAS*, 2005, pp. 461–476.

[8] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with adabu. In *WODA*, 2006, pp. 17–24.

[9] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE*, 2006, pp. 282–291.

[10] D. Lo, L. Mariani, and M. Pezze. Automatic steering of behavioral model inference. In *ESEC/FSE*, 2009, pp. 345–354.

[11] M. Pradel and T. R. Gross. Automatic Generation of Object Usage Specifications from Large Method Traces. In *ASE*, 2009, pp. 371–382.

[12] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *ICSE*, 2011, pp. 591–600.

[13] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. IEEE Transactions on Software Engineering, 38(2):243–257, 2012.

[14] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE*, 2009, pp. 550–554.

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. Science of Computer Programming, 69:35–45, 2007.

[16] M. Gabel and Z. Su. Testing mined specifications. In *FSE*, 2012, pp. 4:1–4:11.

[17] C. Ghezzi, A. Mocci, and M. Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE*, 2009, pp. 430–440.

[18] C. Le Goues and W. Weimer. Measuring code quality to improve specification mining. IEEE Transactions on Software Engineering, 38(1):175-190, 2012.

[19] D. Lo and S.-C. Khoo. Quark: Empirical assessment of automaton-based specification miners. In *WCRE*, 2006, pp. 51–60.

[20] M. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. IEEE Transactions on Software Engineering, (99):1, 2012.

[21] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering, SE-12(1):157-171, 1986.

[22] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE*, 2010, pp. 5–14.

[23] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *POPL*, 2005, pp. 98–109.

[24] R. DeLine and M. Fahndrich. Typestates for objects. In *ECOOP*, 2004, pp. 465–490.

[25] K. Bierhoff and J. Aldrich. Lightweight object specification with typestates. In *ESEC/FSE*, 2005, pp. 217–226.

[26] 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. In *OOPSLA*, 2006, pp. 169–190.

[27] A. Marchetto, P. Tonella, and F. Ricca. State-based testing of ajax web applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, 2008.

[28] A. Mesbah and A. van Deursen. Invariant-based automatic testing of AJAX user interfaces. In *ICSE*, 2009, pp. 210–220.

[29] L. Mariani, A. Marchetto, C. D. Nguyen, P. Tonella, A. Baars. Revolution: automatic evolution of mined specifications. In *ISSRE*, 2012, pp. 241–250.

[30] F. Chen and G. Roşu. Parametric trace slicing and monitoring. In *TACAS*, 2009, pp. 246–261

[31] Q. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Iterative mining of resource-releasing specifications. In *ASE*, 2011, pp. 233–242.