Predictive Typestate Checking of Multithreaded Java Programs

Pallavi Joshi
EECS, UC Berkeley
Email: pallavi@cs.berkeley.edu

Koushik Sen
EECS, UC Berkeley
Email: ksen@cs.berkeley.edu

***Abstract*—Writing correct multithreaded programs is difficult. Existing tools for finding bugs in multithreaded programs primarily focus on finding generic concurrency problems such as data races, atomicity violations, and deadlocks. However, these generic bugs may sometimes be benign and may not help to catch other functional errors in multithreaded programs. In this paper, we focus on a high-level programming error, called typestate error, which happens when a program does not follow the correct usage protocol of an object. We present a novel technique that finds typestate errors in multithreaded programs by looking at a successful execution. An appealing aspect of our technique is that it not only finds typestate errors that occur during a program execution, but also many other typestate errors that could have occurred in a different execution. We have implemented this technique in a prototype tool for Java and have experimented it with a number of real-world Java programs.**

I. Introduction

Multithreaded programs often exhibit wrong behaviors due to unintended interaction between concurrent threads. Such errors are very difficult to detect because they happen under very specific thread schedules. Existing research on finding bugs in multithreaded programs has primarily focused on developing techniques to find violations of generic invariants such as data races [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12] and violations of atomicity [13], [14], [15], [16], [17], [18], [19], [20]. However, it has been found that the violations of these low-level invariants do not always imply that there is a bug in a program. For example, a recent tool [7] on classifying harmful races from benign races showed that 90% of real data races are benign. Moreover, many times a multithreaded program may satisfy all these low-level generic invariants; yet, it can violate some functional invariant such as a thread cannot read from a file handler that has already been closed by another thread. Nevertheless, techniques for finding violations of these generic invariants are popular because they do not need a specification from the programmer.

In this paper, we focus on a dynamic technique to detect violations of a class of high-level properties called typestate properties. Typestate [21] is a temporal extension of types where a user can effectively express the usage patterns of many common libraries and application programming interfaces (APIs). For example, a typestate property on an `InputStream` object in Java is that one cannot read from an `InputStream` object after it has been closed. The typestate property of an object type can be conveniently represented by a finite state automaton. A state in the automaton represents a typestate. At any state in an execution, an object can be exactly in one typestate, with the typestate of the object being set to the initial state of the automaton when the object is being created. Each edge of the typestate is labeled by a method defined by the object type. The invocation of a method during an execution changes the typestate of an object according to its typestate automaton. If a typestate in the automaton has no edge corresponding to a method, then an invocation of the method during an execution on an object in the typestate leads to a violation of the typestate property. Dynamically checking a typestate property [22] of an object simply involves checking that the sequence of method calls made on the object is a sequence accepted by the object's typestate automaton.

We present PRETEX, a dynamic and predictive typestate checking technique for multithreaded program executions. PRETEX instruments a multithreaded program to observe various events in a multithreaded execution such as method calls and thread creations. The observed execution need not violate a typestate property (i.e. a successful execution); yet, PRETEX can predict if some other concurrent execution could violate the typestate property. Specifically, PRETEX computes a dependency relation, called *happens-before* relation [23], between various events generated by a multithreaded execution. The happens-before relation, which is also a partial-order relation, is then used to create various interleavings of the observed events so that they do not violate the observed happens-before relation. Each such interleaving represents a potential concurrent execution of the multithreaded program. All the computed interleavings of events are then checked against typestate properties. As such, although the observed execution may not violate a typestate property, PRETEX can predict typestate violations in other concurrent executions that "came close to happening".

PRETEX has the same flavor as some dynamic race detection [3], [6], [24] and atomicity checking [14], [13] techniques. These techniques look at a multithreaded execution and try to predict if a data race or an atomicity violation can happen in some other concurrent execution. [25] is another approach that extracts a causality relation from an execution trace, and uses it to predict data races and atomicity violations. Unlike these techniques, PRETEX focuses on typestate properties whose violations imply a definite bug in a multithreaded program. Other techniques [26] have been proposed to predict violations of safety properties in multithreaded programs which can be expressed using temporal logic. Temporal logic might not be sufficient to express many typestate properties, and therefore these techniques will not be able to check multithreaded programs against those typestate properties.

Predictive typestate checking for concurrent programs poses three problems. First, checking typestate property for each object type is expensive and time-consuming. Second, coming up with the valid typestate property for each object type requires a lot of manual effort. Third, checking typestate property efficiently against all "nearby" concurrent executions could be expensive. PRETEX aims to solve these problems by combining three techniques in three stages. In the first stage, PRETEX performs object race detection [5] to identify the object types whose methods could be concurrently invoked by multiple threads. Racing object types could only cause a typestate violation due to different interleavings in a concurrent execution; therefore, PRETEX only considers these object types in the next two stages. This helps PRETEX to significantly prune the object types whose typestate needs to be checked predictively. Second, PRETEX observes a successful concurrent execution, that is, an execution that does not throw an exception, and tries to infer the likely typestate property of an object type by using an existing dynamic specification mining technique [27]. There are static methods to mine specifications [28] too, which cover all possible ways an object type can be used and not only the ways that were observed during an execution, but are usually not very scalable. Although our inferred typestate properties may not be accurate, they help to reduce the burden of specifications writer who can further help to refine the inferred specifications rather than trying to write them from scratch. Third, PRETEX efficiently checks the inferred typestate properties by constructing an abstract model of a concurrent execution, called computation lattice.

We have implemented PRETEX in a prototype tool for Java. We have applied PRETEX on a number of open-source Java programs containing 500K lines of code. We have detected a previously unknown typestate bug in a Java application `weblech`. Our experiments show that PRETEX can run efficiently on large programs.

The paper makes the following important contributions.

1) It proposes a dynamic technique to predict typestate errors in multithreaded programs. This helps us to improve the coverage of traditional testing.
2) It combines predictive typestate checking with object race detection and specification mining to reduce the runtime overhead and to reduce the burden of writing specifications, respectively.
3) It presents an implementation and its evaluation on a number of real-world Java programs. The results of our experiments are encouraging.

II. Overview

In this section, we give a gentle introduction to predictive typestate checking of multithreaded programs. We explain the technique using the multithreaded program in Figure 1. The example has two threads: `MainThread` and `ChildThread`. `MainThread` creates a new `Socket` object, connects it to an address, and then starts `ChildThread`. `MainThread` obtains an input stream for the socket, and reads from it a number of times. Finally, `MainThread` closes the socket. `ChildThread` obtains an output stream for the socket, and writes a string to it.

The example program is buggy and can throw an exception. Such an exception is thrown if `MainThread` is executed to completion before `ChildThread` executes its first statement. This is because at the completion of its execution, `MainThread` closes `mySocket` and then `ChildThread` calls `getOutputStream` on the closed socket. In fact, such an execution violates the typestate property that the `getOutputStream` method of a `Socket` object cannot be called after calling the method `close` on the same object. However, in a normal execution it is very unlikely that `ChildThread` will be called after the completion of the execution of `MainThread`. This is because the execution of `MainThread` will take a long time due to the presence of the loop and a fair thread scheduler will schedule `ChildThread` long before `MainThread` completes its execution. Nevertheless, the exception can happen under some schedule and the bug in the program should be fixed.

A naïve way to find this bug would be to execute the program many times with the hope that the thread scheduler will create the buggy thread interleaving in some execution. We propose a technique where we can discover this typestate bug by looking at a single successful execution of the multithreaded program. We next explain how we predict the occurrence of the bug by looking at a successful execution (or exception-free execution) where we execute `ChildThread` before `MainThread` calls the method `getInputStream`. The interleaving is shown in Figure 1.

Our technique works in three stages. In the first stage, we compute the types of the objects whose method calls could potentially race with each other. Only the object types that could potentially race are considered for typestate checking in the next two stages. This is because the objects that could potentially race are likely to violate a typestate property due to lack of synchronization. Note that the first stage is only meant for optimization. In the example program, the `Socket` object, `mySocket`, is in race, since the method invocations, `getInputStream`() in `MainThread`, and `getOutputStream`() in `ChildThread`, can occur in either order.

Our second stage is the typestate specification mining stage. We use the typestate properties mined in this stage to predict if they could have been violated in some concurrent execution that was not observed but that could have occurred. In this stage, we infer the likely typestate property of each object type by observing a successful concurrent execution. Specifically, for each type obtained from the previous stage, we obtain the sequence of method calls invoked on each object of that type. We pass these sequences of method calls to an off-the-shelf machine learning procedure [29] to learn an automaton

```
MainThread                                                              ChildThread

InetAddress ad = InetAddress.getByName("testsite.com");
Socket mySocket = new Socket();
mySocket.connect(new InetSocketAddress(ad,80));
ChildThread.start();

                                                        OutputStream os = MainThread.mySocket.
                                                                   getOutputStream();
                                                        PrintWriter out = new PrintWriter(os,true);
                                                        out.println("testString");
InputStream is = mySocket.getInputStream();
BufferedReader ibr =
           new BufferedReader(new InputStreamReader(is));
for (n =0; n < 100; n++)
   String line = ibr.readLine();
mySocket.close();
```

Fig. 1. Predicting an IOException bug that is thrown when getOutputStream() is invoked on a closed Socket object

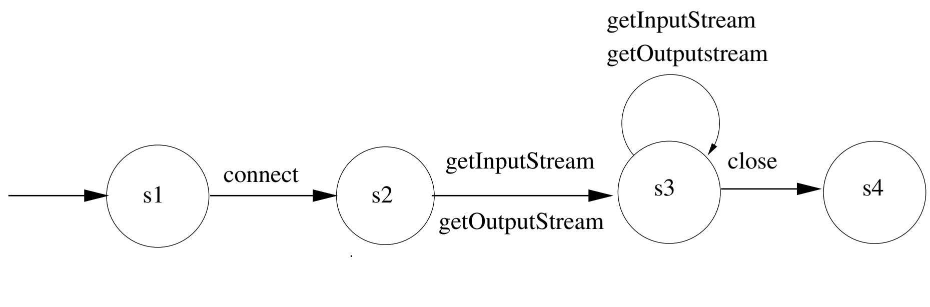

Fig. 2. The likely typestate specification automaton learnt for Socket objects

that contains all the observed sequences. Such an automaton represents the likely typestate property of the object type, i.e. the valid sequence of method calls on that object type.

For example, the sequence of calls on `mySocket`, the only `Socket` object, is `connect`, `getOutputStream`, `getInputStream`, `close` in a successful run where `ChildThread` terminates before `MainThread`. From this sequence, we can infer the likely typestate property that `getInputStream` or `getOutputStream`, cannot be called on a `Socket` object after `close`() has been invoked on it. Typically, in our benchmarks, we have a number of objects for each type in race, and hence, the automaton that describes the union of the sequences observed for these objects gets close to the correct typestate specification automaton for that type. Figure 2 gives a typestate specification automaton that can be inferred for `Socket` objects from our example.

Our third stage is the predictive typestate checking stage. Once we have a likely typestate property for each type and a successful concurrent execution, we predictively check each typestate property against the successful concurrent execution. That is, by looking at the causal dependence among the various events in the successful concurrent execution, we compute other concurrent executions that can be obtained from the successful concurrent execution by reordering independent events. We then check each typestate property against the computed concurrent executions. Since we check each typestate property against a number of concurrent executions that "came close to happening", we call this *predictive* typestate checking. An advantage of predictive typestate checking is that we can predict violations of a typestate property that could potentially happen in other concurrent executions only by analyzing a successful concurrent execution. In order to compute the concurrent executions "near" the successful execution, we compute an abstract model from the sucessful concurrent execution, called the *multithreaded computation lattice*. Each path in such a computation lattice denotes a concurrent execution that could potentially happen if we change the thread interleaving slightly. We then check the typestate property against all paths in this lattice using a dynamic programming algorithm.

Figure 3 gives the lattice for the observed execution in our example. The solid lines describe the observed concurrent execution from which we have computed the computation lattice, whereas the dotted lines trace the concurrent executions that could have been observed. The program state Σ^{30} in the lattice shows that the typestate property of the `Socket` type can be violated. This is because the socket is already closed in this state, when the method `getOutputStream()` is invoked on it. Note that we do not observe this state in the actual concurrent execution, but we predicted this erroneous state by analyzing the computation lattice.

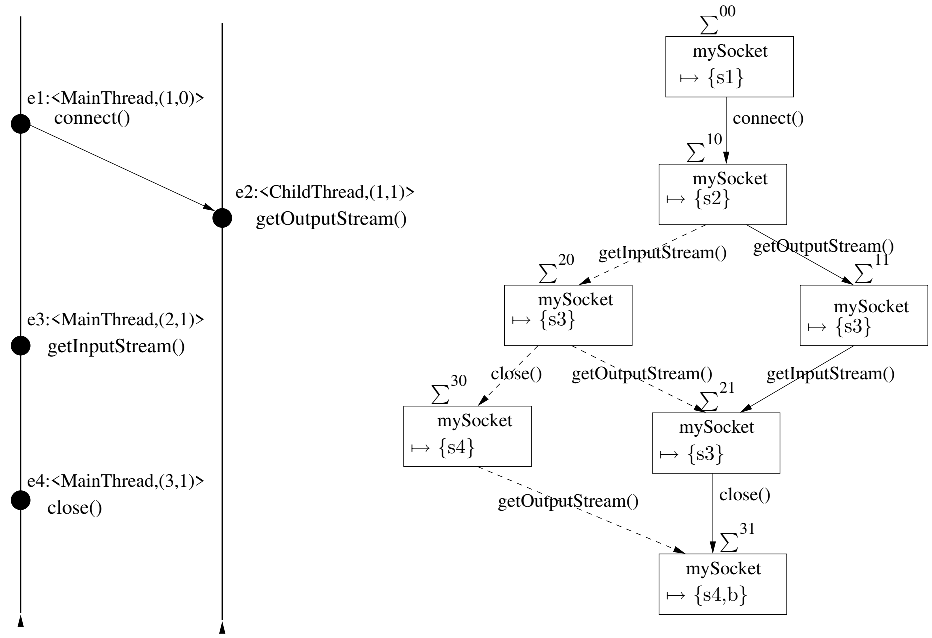

Fig. 3. The multithreaded computation lattice for Figure 1. The left-hand side diagram depicts the partial order observed during program execution. Each event is a pair of the thread name in which it occurred, and the vector clock of the thread when it occurred. The state b in Σ^{31} in the computation lattice is the bad state in the typestate specification automaton for `Socket`. Any transition that is not possible in the specification automaton leads to the bad state b.

III. Predictive Typestate Checking of Multithreaded Programs

We describe the different stages of our technique in this section. The first stage executes the program and finds the types of objects that are involved in race in those executions. The second stage executes the program again, once for each type that was identified to be in race in the previous stage, and obtains the sequence of method calls for each object of that type. PRETEX then constructs a typestate specification automaton for each type of object in race from the method sequences. The third and final stage predictively checks the inferred typestate specifications against a multithreaded execution.

A. Background Definitions

We introduce some standard definitions that we will use to explain the different stages of our technique. The execution of a multithreaded program can be seen as a sequence of events, where an event could be one of the following types.

- MEM(o,t,L,m) denotes that thread t invoked the method m on object o while holding the set of locks L. L is also called as the lockset held by thread t.
- SND(g,t) denotes the sending of a message with unique ID g by thread t.
- RCV(g,t) denotes the reception of a message with unique ID g by thread t.

Given an event sequence $\langle e_i \rangle$, we can define a happens-before relation $\prec$ as follows. $\prec$ is the smallest relation satisfying the following conditions:

- If e_i and e_j are events in the same thread and e_i comes before e_j in the sequence $\langle e_i \rangle$, then $e_i \prec e_j$.
- If e_i is the sending of the message g and e_j is the reception of the message g, then $e_i \prec e_j$.
- If events e_i, e_j, and e_k are such that $e_i \prec e_j$ and $e_j \prec e_k$, then $e_i \prec e_k$.

The happens-before relation $\prec$ is computed at runtime by maintaining a vector clock [30], [31], [24] with every thread. Each thread t_i maintains a vector clock indexed by thread IDs. t_i's vector clock entry for t_j indicates the last event in t_j that could have affected t_i. The details of how we maintain vector clocks is standard and can be found in [24].

We next describe the notion of a multithreaded computation that we use in our predictive checking algorithm. This lattice is computed from the execution of a multithreaded program as in [32], [26], [33], [34]. We denote by ${e_i}^k$, the k-th event that occurred in the i-th thread t_i. Then the program state after events ${e_1}^{k_1}$,${e_2}^{k_2}$,...,${e_n}^{k_n}$ is denoted by $\Sigma^{k_1 k_2 \ldots k_n}$. A state $\Sigma^{k_1 k_2 \ldots k_n}$ is called consistent [34] if and only if for any $1 \leq i \leq n$ and any $l_i \leq k_i$, $l_j \leq k_j$ holds for any $1 \leq j \leq n$ and any l_j such that ${e_j}^{l_j} \prec {e_i}^{l_i}$. In other words, a consistent state is one which can be formed by an interleaving of events that respects the happens-before relation. Let $\Sigma^{00\ldots0}$ be the initial program state. A feasible interleaving of events e_1,e_2,...,e_m generates a sequence of program states Σ^{K_0},Σ^{K_1},...,Σ^{K_m} for which the following two conditions hold. Each Σ^{K_r} is consistent, and for any two consecutive states, Σ^{K_r} and $\Sigma^{K_{r+1}}$, K_r and K_{r+1} differ in exactly one index by one. If the index in which the two differ is i, then the i-th element of K_{r+1} is larger by one than the i-th element of K_r. A sequence of states Σ^{K_0},Σ^{K_1},...,Σ^{K_m} thus identifies an interleaving of events, or a run of the program. We say that Σ leads to $\Sigma^{'}$, written as $\Sigma \rightsquigarrow \Sigma^{'}$ if there is a run in which Σ and $\Sigma^{'}$ are consecutive states.

The set of all program states together with the partial order $\rightsquigarrow$ forms a lattice. For a state $\Sigma^{k_1 k_2 \ldots k_n}$, we call k_1+k_2+...+k_n as its level in the computation lattice.

B. Object race detection

This stage finds the types of objects whose method calls by different threads are in race. Our algorithm to detect the objects that are in race is a combination of the dynamic race detection techniques proposed in [5], [24]. Specifically, at runtime, we check the following condition for each pair of events (e_i,e_j).

$$e_i = \text{MEM}(o_i, t_i, L_i, m_i) \wedge e_j = \text{MEM}(o_j, t_j, L_j, m_j) \\ \wedge (t_i \neq t_j) \wedge (o_i = o_j) \wedge (L_i \cap L_j = \emptyset) \wedge \neg(e_i \prec e_j) \\ \wedge \neg(e_j \prec e_i).$$

The above condition essentially states that two events are in race if they are events on different threads, they are due to method calls on the same object, and the two events are not related by the happens-before relation (i.e. the two events are concurrent). If the above condition holds, we say that the object o_i could be in race, and we record its type. We use vector clock to track $\prec$. The following SND(g,t) and RCV(g,t) events are considered in the above condition. If thread t_1 starts a thread t_2, then events SND(g,t_1) and RCV(g,t_2) are generated, where g is a unique message ID. If thread t_1 calls t_2.`join`() and t_2 terminates, then events SND(g,t_2) and RCV(g,t_1) are generated, where g is a unique message ID. If a o.`notify`() or o.`notifyAll`() in thread t_1 signals a o.`wait`() in thread t_2, then events SND(g,t_1) and RCV(g,t_2) are generated, where g is a unique message ID. Objects which are in race are more likely to result in typestate errors. Therefore, we concentrate on building the typestate specification automaton for such objects in subsequent stages.

C. Inferring likely typestate specifications

Typestate [21] can be used to express the correct usage rules for many application programming interfaces (APIs). For example, one can use typestate to express that a `java.net.Socket` object cannot be read from after it has been closed. A typestate specification uses a finite state automaton (FSA) to encode the correct usage protocol. A state in the FSA is called a typestate, and an object is in one of these typestates at any point of time during program execution. The edges in the automaton are labeled with method names. When a method is invoked on an object, it follows that outgoing edge from its typestate which is labeled with the method name, and transitions to a new typestate (which might be the same as its old typestate). If no such edge exists in its current typestate, then we say that a typestate error has occurred. In this section, we briefly describe how we obtain the typestate specifications for object types that we identified to be in race in the previous stage.

For each type in race, we collect the sequence of method calls invoked on each object of that type. For example, if we find that objects of type `java.net.Socket` are in race, then for each object of type `java.net.Socket`, we record the sequence of method calls that was invoked on it during the execution of the program. Thus, we get as many sequences as the number of `Socket` objects that were used during the execution of the program. For each such set of sequences, we learn a finite state automaton (FSA) that accepts the sequences in the set, and rejects most of those outside the set. The automaton so learnt can be thought of as the typestate specification of that object type because it captures all the different ways objects of that type were used during program execution. Moreover, since the execution does not throw an exception, we can assume that the observed sequences of method calls are valid.

The FSA that we learn is a deterministic finite automaton (DFA), the edges of which are labeled with method names. We use an off-the-shelf PFSA (Probabilistic Finite State Automaton) learner [29]. The learner infers a PFSA that accepts the set of method call sequences presented to it, plus some more sequences that get added as it generalizes. The PFSA learner first constructs a prefix tree or a trie from the set of sequences. Each arc of the trie is labelled with a frequency that reflects how many times that arc was traversed while creating the trie. The trie can be seen as a FSA that accepts the set of sequences from which it was built. Since prefix trees are usually very large in size, the PFSA learner uses the sk-strings method [29] to merge states in the prefix trees. The sk-strings method is a variation of the k-tails method [35] for stochastic automata. It constructs a non-deterministic finite automaton (NFA) by successively merging those states of the trie which are sk-equivalent. Let Σ be the set of method names in the set of sequences, Q be the set of states in the trie, $\delta : Q \times \Sigma^* \rightarrow 2^Q$ be the transition function of the trie, and F_C be the final states of the trie. The set of k-strings of state q is then defined to be the set $\{z | z \in \Sigma^*, |z| = k \wedge \delta(q, z) \subset Q \vee |z| < k \wedge \delta(q, z) \cap F_c \neq \emptyset\}$. Each k-string has a probability associated with it which is equal to the product of the probabilities of the arcs traversed to form that string. The k-strings of a state are arranged in decreasing order of their probabilities. The top n strings, whose probabilities add up to s% or more, are retained and the rest discarded. Two states are said to be sk-equivalent if the sets of the top n strings of both are equal. The process of merging sk-equivalent states is repeated until no more states can be merged. The resulting PFSA accepts a superset of the method call sequences that was presented to it, due to the generalizations performed during merging. The final stage in the learning process converts the NFA into a DFA.

The DFA learned in this stage for each object type can be used as the likely typestate specification for the type. Note that we use this stage to reduce the burden on users so that they do not have to write tedious typestate specification for each type from scratch. However, they can take a look at the inferred typestate automata and refine them as required.

D. Predictive checking against typestate specifications

After we infer the likely typestate specification automata, we predictively check them against a multithreaded execution. Using the multithreaded execution, we generate a mutithreaded computation lattice for each automaton based on method invocation events that are relevant to that automaton. The elements of the lattice consist of program states, where a program state is a mapping from objects of the type whose typestate specification is represented by the automaton to sets of states in the typestate specification automaton. The algorithm to generate the lattice is very similar to the one presented in [34], except for the happens-before relation employed by them. The happens-before relation in [34] considers shared variable reads and writes, and lock acquires and releases, along with the synchronization events, `start()`, `join()`, `wait()`, `notify()` and `notifyAll()`. We do not consider shared variable reads and writes, and lock aquires and releases to avoid the overhead that would be incurred if we kept track of them. The happens-before relation that we employ is, thus, an over-approximation of the exact happens-before relation that exists between the events in a multithreaded execution, but it helps us in verifying more thread interleavings, some of which might be feasible, but not possible under the more conservative happens-before relation in [34].

We generate the lattice on-the-fly during the execution of the program, and analyze it level-by-level. After the states of a level have been analyzed, we discard them. Storing all states in a level may also lead to a number of states exponential in the number of levels being stored. Therefore, we employ the causality cone heuristic [34]. Instead of generating all possible states in a level, the heuristic considers a level to be complete after w states in the level are generated, where w is a pre-determined parameter. However, a level may contain less than w states. The level construction algorithm would get stuck in that case. Also, one cannot determine if there are less than w states in a level unless one sees all the events in the complete computation. This is because the total number of threads is not known until the end of the execution. To avoid this, another parameter l is introduced, which is the length of the current event queue Q. We consider the construction of a level to be complete if we have used all the events in Q for the construction of the states in the current level and the length of queue is at least l, or if we have generated w states in the current level. Algorithms 1 and 2 outline these.

In the functions, `threadId`(e) returns the thread t where the event e has occurred. For a method invocation event MEM(o,t,L,m), `obj`(e) returns o, and `methodId`(e) returns m. We maintain vector clocks for each program state Σ and each event e. The vector clock of a program state Σ reflects the latest event that has occurred in each thread when the program state is reached. The vector clock of an event e is the vector clock of its thread when it occurred. `VC`(Σ) gives the vector clock associated with program state Σ, and `VC`(e) gives the vector clock associated with event e.

The function `constructLevel()` constructs feasible states from the states in the current level and events in the event queue Q. A state Σ in the current level and an event e in the event queue can give rise to a feasible state if and only if Σ has all the information about the events in the current execution of the program and the happens-before relation between them that e has, except for the event e itself. A new state is created by applying the transition represented by the method invocation event e to the set of states that each object is mapped to in the program state.

The function `isNextState()` checks if the program state Σ and the method invocation event e can give rise to a new feasible state. If a new feasible state is possible, then the function `createState()` creates the new state. ρ takes the set of states `obj`(e) is mapped to in Σ and `methodId`(e), applies the transition corresponding to `methodId`(e) in the typestate specification automaton to the set of states, and returns the resultant set of states. If the bad state b is present in the resultant set of states, then we report a typestate error. Any transition that is not possible in a typestate specification automaton is considered to lead to the bad state b. If the new state that is created has a vector clock equivalent to that of a state already in the next level, we merge the two states. We merge the mappings of the corresponding objects in the two states.

We generate states for the next level until the predicate `isLevelComplete` is satisfied. Upon the completion of a level, all the events in the event queue Q which can no longer give rise to feasible states in subsequent levels are discarded. This is done by the function `removeUselessEvents()`, which creates a vector clock VC_{min}, each component of which is the minimum of the corresponding components of the vector clocks of the states in the current level. All events in the event queue which have a vector clock less than or equal to VC_{min} are removed, because they cannot generate feasible program states any more.

IV. Implementation

We have implemented our technique for Java in a prototype tool. We instrument Java bytecode to observe various events. Bytecode instrumentation allows us to analyze any Java program for which the source code is not available. We use the Soot compiler framework [36] to insert probes into the bytecodes of the Java programs. These probes call methods in our analyses which are also written in Java. For the first stage of the analysis, the object race detector, we add instrumentation to keep track of the locksets and vector clocks. For maintaining vector clocks, we instrument the method calls, `start()`, `join()`, `wait()`, `notify()`, and `notifyAll()`. We maintain a database of method invocation events and the locksets and vector clocks associated with them. Moreover, since storing each possible event in the database would stress the memory requirement of the application, we implement the following optimization. Before adding an event to the database, we first search the database to see if such an event already exists. If it does, then we do not add the current event to the database, or else we add it to the database. Since we track the thread dependencies arising out of `start()`, `join()`, `wait()`, `notify()`, and `notifyAll()` and ignore other dependencies present between the threads, a considerable number of events that occurs on an object in a thread occurs with the same vector

Algorithm 1 Level-by-level traversal of the computation lattice

```
while not end of computation do
  Q := enqueue(Q,NextEvent())
  while constructLevel() do
    NOP
  end while
end while

boolean constructLevel()
for each e ∈ Q do
  if Σ ∈ CurrLevel and isNextState(Σ,e) then
    NextLevel := NextLevel ⊎ createState(Σ,e)
    if isLevelComplete(NextLevel,e,Q) then
      Q := removeUselessEvents(CurrLevel,Q)
      CurrLevel := NextLevel
      return true
    end if
  end if
end for
return false

boolean isNextState(Σ,e)
i := threadId(e)
if (∀ j ≠ i : VC(Σ)[j] ≥ VC(e)[j] and VC(Σ)[i]+1 = VC(e)[i])
then
  return true
else
  return false
end if

State createState(Σ,e)
Σ' := new copy of Σ
j := threadId(e)
VC(Σ')[j] := VC(Σ)[j] + 1
Σ' := Σ'[obj(e) := ρ(Σ(obj(e)),methodId(e))]
if b ∈ ρ(Σ(obj(e)),methodId(e)) then
  print 'typestate error observed'
end if
return Σ'
```

Algorithm 2 isLevelComplete predicate

```
boolean isLevelComplete(NextLevel,e,Q)
if size(NextLevel) ≥ w then
  return true
else if e is the last event in Q and size(Q) ≥ l
then
  return true
else
  return false
end if
```

clock and lockset. Therefore, for all of these events, we have to add only a single entry to the database.

For the second stage of the analysis, the method call sequence extractor, we instrument the program to keep track of the method calls invoked on objects of a certain type, which is provided as a parameter to the instrumented program. We run the instrumented program for each type that is identified to be in race in the previous stage. For each such run, we collect the sequences of method calls invoked on objects of that type. We then use an off-the-shelf PFSA builder [29] on the sequences of method calls for each type in race. The edges of the automata are labeled with method call names. These automata are used in the next stage for typestate checking.

For each automaton that we generate, we instrument the program at each point where a method call present in one of the edges of the automaton is invoked. The instrumentation uses the method invocation events to build the levels of the multithreaded computation lattice. We also track thread dependencies by instrumenting the method calls, `start()`, `join()`, `wait()`, `notify()`, and `notifyAll()`.

V. Empirical Evaluation

A. Experimental setup

We evaluated our prototype tool on a number of benchmark programs. We ran our experiments on a laptop with a 2GHz Intel Core 2 Duo processor and 2GB RAM. We considered the following benchmark programs: `hedc`, a meta-crawler application kernel developed at ETH [5], `weblech`, a website download tool, `tornado`, a multithreaded web server, `cache4j`, a fast thread-safe implementation of a cache for Java objects, `jspider`, a web spider engine, `jigsaw 2.2.6`, W3C's web server, and `apache ftpserver`. The eighth column of table I gives the LOC count for these benchmarks. The last column in the table is the number of threads that were spawned for the benchmark applications. All of these were closed programs except for `jigsaw` and `tornado`. For `jigsaw` and `tornado`, we wrote harnesses that spawned a number of threads and queried the web server for different urls.

B. Results

Table I summarizes the average execution time of the various benchmarks for the different stages of the analysis. The second column gives the average execution time of the unmodified benchmark. The third column is the average time taken for obtaining the method call sequences for objects of a particular type. The fourth column gives the average time for the PFSA builder to build a PFSA from a set of method call sequences. The fifth column is the average execution time to run the predictive typestate checker using a single automaton. All of the execution time is in milliseconds. The sixth column gives the total number of typestate errors reported by our tool. An error that is reported more than once is counted only once, and not the number of times it was reported. We manually inspect all the errors that are reported, and provide the number of real errors that we find in the seventh column.

As can be seen from the table, the execution time of an application after being instrumented to print the method call sequences is less than 3 times the execution time of the original application, except in the case of `cache4j`. The execution time of an application after being instrumented to predict typestate errors is less than 2 times the execution time of the uninstrumented application, except for `cache4j`. The overhead of instrumentation is thus very small. The PFSA builder takes almost constant time to build a PFSA from a set of method sequences. We manually examined the errors that

Benchmark	Normal Exec Time (in ms)	Exec Time (method seqs) (in ms)	Exec Time (PFSA) (in ms)	Exec Time (typestate errors) (in ms)	# errors reported	# actual errors	LOC	Threads
tornado	4141	4125	1235	4140	6	0	1326	40
cache4j	4250	90421	1228	99609	3	0	3897	10
hedc	2813	2829	1352	2766	5	0	29948	5
weblech	1079	2641	1353	1609	6	1	35175	3
jspider	641	922	1233	781	7	0	64933	5
ftpserver	4890	8109	152	8125	8	0	127297	40
jigsaw	39031	39000	1352	39000	12	0	381348	30

TABLE I
Execution time for typestate checking

were reported, and found one real error in `weblech` which we describe next.

In `weblech`, the URLs to be examined are queued in an instance of the class `DownloadQueue`. In `Spider.java`, due to insufficient synchronization, a thread might try to retrieve a URL from `DownloadQueue` even if there is no URL in it. The relevant portion of the code is shown below.

```
if(queueSize() == 0 && downloadsInProgress > 0)
{
     ...
     continue;
}
else if(queueSize() == 0)
{
     break;
}
....
synchronized(queue)
{
     nextURL = queue.getNextInQueue();
     downloadsInProgress++;
}
```

When `queue.getNextInQueue()` is called in the above code, the condition that `queueSize()` $\neq$ `0` could have become false. The execution of `queue.getNextInQueue()` can result in an exception being thrown if the size of `queue` is 0. The typestate automaton inferred for `DownloadQueue` by our analysis correctly infers that `size()` should always be called before `getNextInQueue()`. The automaton is shown in figure 4. Using this typestate automaton we could predict the typestate error in the above code.

There are two main sources of false positives in our experiments. We do not track all possible dependencies between the threads, for example, the dependencies due to lock acquires and releases. Therefore, our happens-before relation is an over-approximation of the exact happens-before relation that exists between the events in the observed multithreaded execution. This results in the consideration of thread interleavings that might be infeasible, and hence the typestate errors that are reported to occur in them might not be possible in any real execution of the program. Moreover, since the automata that we build are based on the sequences of method calls that are observed during a certain execution of the program, they do not capture all legitimate sequences of method calls on objects of the type concerned. Thus, some of the errors that are reported are caused by legitimate interactions with the objects that were not observed during the building of the typestate specification automata.

VI. Conclusion

We proposed a dynamic technique to predictively check typestate violations in multithreaded programs. An appealing aspect of our approach is that we can predict a typestate violation by analyzing a multithreaded execution that does not directly violate the typestate specification. This helps us to improve the coverage of traditional testing and check properties that are more high-level than data race and atomicity violation. We also showed how to combine predictive typestate checking with object race detection and specification mining to reduce the runtime overhead and to reduce the burden of writing specifications, respectively. Thus, our technique is fully automated. We presented an implementation and its evaluation on a number of real-world Java programs. The results of our experiments are encouraging.

VII. Acknowledgments

This work is supported in part by the NSF Grant CNS-0720906.

References

[1] A. Dinning and E. Schonberg, “Detecting access anomalies in programs with critical sections,” in *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[2] J.-D. Choi, B. P. Miller, and R. H. B. Netzer, “Techniques for debugging parallel programs with flowback analysis,” *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 491–530, 1991.

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[4] M. Christiaens and K. D. Bosschere, “Trade, a topological approach to on-the-fly race detection in java programs,” in *JavaTM Virtual Machine Research and Technology Symposium (JVM)*. USENIX Association, 2001, pp. 15–15.

[5] C. von Praun and T. R. Gross, “Object race detection,” in *16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*. ACM, 2001, pp. 70–82.

[6] J. D. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and precise datarace detection for multithreaded object-oriented programs,” in *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, 2002, pp. 258–269.

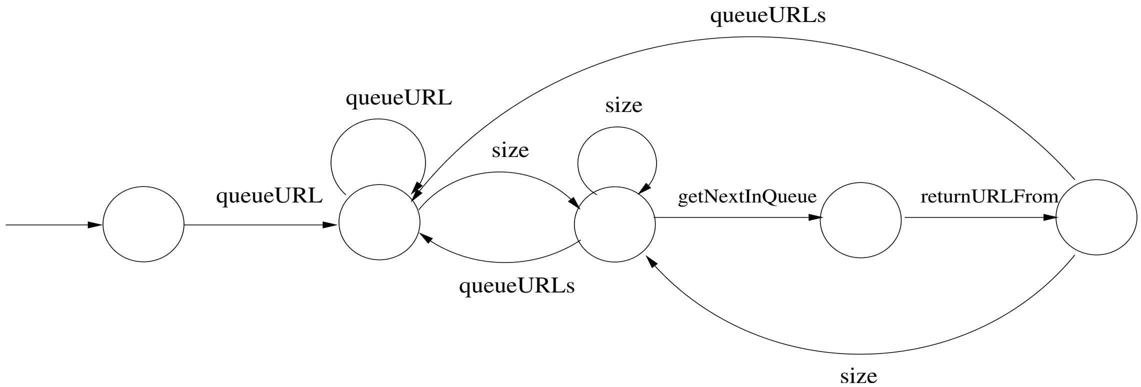

Fig. 4. Typestate automaton inferred for DownloadQueue

[7] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data racesallusing replay analysis," in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, 2007, pp. 22–31.

[8] K. Sen, "Race directed randomized dynamic analysis of concurrent programs," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008, to appear.

[9] N. Sterling, "Warlock: A static data race analysis tool," in *USENIX Winter Technical Conference*, 1993, pp. 97–106.

[10] D. R. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237–252.

[11] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: context-sensitive correlation analysis for race detection," in *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*. ACM, 2006, pp. 320–331.

[12] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 308–319.

[13] L. Wang and S. D. Stoller, "Run-time analysis for atomicity." in *3rd Workshop on Run-time Verification (RV'03)*, ser. ENTCS, vol. 89, no. 2, 2003.

[14] C. Flanagan and S. N. Freund, "Atomizer: a dynamic atomicity checker for multithreaded programs," in *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004, pp. 256–267.

[15] L. Wang and S. D. Stoller, "Accurate and efficient runtime detection of atomicity errors in concurrent programs," in *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, Mar. 2006, pp. 137–146.

[16] R. J. Lipton, "Reduction: A method of proving properties of parallel programs," *Communications of the ACM*, vol. 18, no. 12, pp. 717–721, 1975.

[17] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'03)*, 2003, pp. 338–349.

[18] S. N. Freund and S. Qadeer, "Checking concise specifications for multithreaded software." *Journal of Object Technology*, vol. 3, no. 6, pp. 81–101, 2004.

[19] J. Hatcliff, Robby, and M. B. Dwyer, "Verifying atomicity specifications for concurrent object-oriented software using model-checking," in *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, 2004, pp. 175–190.

[20] C. Flanagan, "Verifying commit-atomicity using model-checking," in *11th International SPIN Workshop*, 2004, pp. 252–266.

[21] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. 12, no. 1, pp. 157–171, 1986.

[22] P. Avgustinov, J. Tibble, and O. de Moor, "Making trace monitors feasible," *SIGPLAN Not.*, vol. 42, no. 10, pp. 589–608, 2007.

[23] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[24] R. O'Callahan and J.-D. Choi, "Hybrid dynamic data race detection," in *ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2003, pp. 167–178.

[25] F. Chen, T. F. Serbănută, and G. Rosu, "jpredictor: A predictive runtime analysis tool for java," in *International Conference on Software Engineering (ICSE'08)*. ACM press, 2008.

[26] K. Sen, G. Rosu, and G. Agha, "Runtime safety analysis of multithreaded programs," in *Proceedings of 4th joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 03)*, ser. ACM, 2003.

[27] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *Symposium on Principles of Programming Languages*, 2002.

[28] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *International Symposium on Software Testing and Analysis*, 2007.

[29] A. V. Raman and J. D. Patrick, "The sk-strings method for inferring pfsa," in *Proceedings of the workshop on automata induction, grammatical inference and language acquisition at the 14th international conference on machine learning (ICML97)*, 1997.

[30] C. J. Fidge, "Timestamp in message passing systems that preserves partial ordering," in *Proceedings of the 11th Australian Computing Conference*, 1988, pp. 56–66.

[31] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the Parallel and Distributed Algorithms Conference*, ser. Elsevier Science, 1988, pp. 215–226.

[32] O. Babaoğlu and K. Marzullo, "Consistent global states of distributed systems: Fundamental concepts and mechanisms," in *Distributed Systems*, S. Mullender, Ed. Addison-Wesley, 1993, pp. 55–96. [Online]. Available: citeseer.nj.nec.com/article/babaoglu93consistent.html

[33] A. Sen and V. K. Garg, "Partial order trace analyzer (pota) for distrubted programs," in *Proceedings of the 3rd Workshop on Runtime Verification (RV03)*, ser. Electronic Notes in Theoretical Computer Science, 2003.

[34] K. Sen, G. Rosu, and G. Agha, "Online efficient predictive safety analysis of multithreaded programs," in *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04)*, ser. Lecture Notes in Computer Science, vol. 2988, 2004, pp. 123–138.

[35] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Transactions on Computers*, pp. 591–597, 1972.

[36] "Soot: A java optimization framework," http://www.sable.mcgill.ca/soot/.