Combining Static Analysis and Model Checking for Software Analysis

Guillaume Brat
Kestrel/NASA Ames Research Center
Moffett Field, CA 94035, USA
brat@email.arc.nasa.gov

Willem Visser
RIACS/NASA Ames Research Center
Moffett Field, CA 94035, USA
wvisser@email.arc.nasa.gov

Abstract

We present an iterative technique in which model checking and static analysis are combined to verify large software systems. The role of the static analysis is to compute partial order information which the model checker uses to reduce the state space. During exploration, the model checker also computes aliasing information that it gives to the static analyzer which can then refine its analysis. The result of this refined analysis is then fed back to the model checker which updates its partial order reduction. At each step of this iterative process, the static analysis computes optimistic information which results in an unsafe reduction of the state space. However, we show that the process converges to a fixed point at which time the partial order information is safe and the whole state space is explored.

1 Introduction

In industrial settings, software verification consists almost entirely of testing. Formal analysis, be it based on static analysis or model checking, is not considered practical for software applications. Fortunately, this situation is slowly changing and more resources are devoted to improving the practicality of such analysis tools. For example, the Java PathFinder (JPF) model checker has been applied to the verification of critical avionics software [2, 11, 13].

JPF is a model checker which operates on principles similar to the SPIN model checker [7], i.e., given a closed environment for software, it performs a systematic exploration of the state space of the program by executing it. Therefore, JPF has to deal with issues such as generating an environment to close a system, deriving finite models from infinite state spaces, and curbing the state explosion problem (so that exhaustive exploration can be performed). This work focuses on alleviating the state explosion problem (i.e., the model checker runs out of memory before it can explore the whole state space) by using partial order reduction, which eliminates the exploration of redundant paths due to the interleaving of independent transitions.

JPF relies on abstraction and static analysis to reduce the size of state spaces. Abstraction is used prior to model checking; it generates a smaller program (meaning that it yields a smaller state space than the original program) that is a safe (it preserves the behaviors that are relevant to the property under consideration) approximation of the original program. Static analysis is also used prior to model checking to slice the original program [6] (yielding a smaller, safe program), and also, compute information needed to perform partial order reduction during model checking. JPF has the following important (for this work) characteristics:

- it is an explicit-state model checker for Java programs, i.e., it uses a custom Java virtual machine to explore states in a DFS manner and verify properties;
- it prunes on-the-fly the state space using partial order reduction on safe transitions;
- it relies on static analysis to compute safe transitions.

This paper focuses on partial order reduction, and the interactions between static analysis and model checking. Since static analysis is used prior to model checking, partial order analysis is subject to the following limitations of static analysis: no knowledge of the values taken by variables at run time (even though we make limited use of symbolic evaluation through the Bandera toolset [4]), approximations inherent to using the traditional dataflow framework (e.g., widening or k-limiting), and, the fact that the precision of static analysis is directly conditioned by the precision of its aliasing algorithm. Unfortunately, most practical alias analyses are quite imprecise. JPF however can compute precise aliasing since it explores the state space by executing the program under all possible interleavings. Therefore, our driving idea is to research how we can interface the static analyzer and the model checker so that we can take advantage of one's strengths to cancel the other's weaknesses.

In this paper, we show that the static analyzer and the model checker can operate concurrently and reduce significantly the size of the explored state space. The static an-

alyzer can use dynamic information (i.e., alias sets) computed by JPF during state exploration and statically compute partial order information that JPF can use to prune the state space. The aliasing information given to the static analyzer is a subset of the real alias set; therefore, static analysis may produce incorrect information about safe statement which JPF cannot trust to definitely discard execution paths. However, this information can be used to pick paths (that are not discarded by the current partial order information) during state exploration. As JPF covers more of the state space, it computes a safer alias set (i.e., closer to the real alias set), and in turn, the partial order information computed by the static analyzer is safer (i.e., more relevant paths get included in the exploration). Eventually, this iterative analysis reaches a fixed point (the alias set is the complete alias set for the current environment), and, the model checker knows exactly what paths can safely be discarded. The innovativeness of our work is twofold:

1. we use static analysis and model checking concurrently to increase their precision, and
2. we perform a safe analysis by computing (less and less) unsafe information.

The paper is organized as follows. Section 2 describes static analysis and partial order reduction. In Section 3, we formalize our approach and prove that it yields safe solutions. We then describe an example, discuss some practical considerations, and present our conclusions.

The rest of the paper uses the following notations. Statements (also referred to as transitions) are denoted by lower case letters such as $n, m, p, \ldots$. The set of all statements is called T. The thread of execution for a given statement n is given by $\theta(n)$. The state of the system is usually referenced by the lower case letter s. The state reached from a state s after statement n has been executed is denoted by $n(s)$. The set of all possible states is called S. The set $enabled(s)$ represents the set of statements that can be executed at state s, i.e., the transitions that are enabled in state s. For a given statement n, $def(n)$ ($ref(n)$) is the set of variables defined (used) at n.

2 Static Analysis in JPF

Java PathFinder [2, 11, 13] is an explicit state model checker that takes compiled Java programs (i.e., byte code class-files) and analyzes all paths through the program for deadlock, assertion violations and linear time temporal logic (LTL) properties. JPF is built on a custom Java Virtual Machine (JVM) and therefore does not require any translation to an existing model checker's input notation. Since JPF is custom-made for Java model checking, it allows an aggressive attack on the state-explosion problem inherent in most complex Java programs. Importantly, the JPF model checker has full control over which (Java) statements to execute in every state, and moreover, has the complete state of the JVM at its disposal at all times during execution. These two characteristics allows the implementation of the concepts in this paper: partial-order reductions (only execute certain transitions in each state) and calculation of alias information (present in the data-portion of the JVM state).

2.1 Partial Order Reduction

To perform a correct verification of an asynchronous system (in our case, a multi-threaded Java program), the model checker has to explore all possible interleavings of concurrent transitions (i.e., concurrent statements) in the system. Unfortunately, the interleaving model yields a combinatorial explosion in the number of states that need to be explored. The goal of partial order reduction is to use the commutativity of concurrent transitions to reduce the state space that needs to be explored by a model checker.

For example, a system consisting of two threads with three transitions each, such that transitions in one thread are independent from transitions in other threads, yields a space consisting of 16 states and up to 20 different paths. However, any such two paths differ by at most nine commutations of transitions (on different threads). If the property checked by the model checker on this system is not sensitive to those transitions, then the state space can be reduced to only one path, i.e., seven states.

As described in [3], two transitions are *independent* if the execution of one does not disable the other (and vice versa) (*enabledness* condition) and they result in the same state regardless of their execution order (*commutativity* condition).

Definition 1 *two transitions n and m are independent at a given state s if the following two conditions are satisfied:*

1. $n, m \in enabled(s) \Rightarrow n \in enabled(m(s))$
2. $n, m \in enabled(s) \Rightarrow n(m(s)) = m(n(s))$

These conditions define an independence relation between pairs of transitions (statements) that is symmetric and anti-reflexive. In [8], Holzmann and Peled extend this definition of independence with the concept of *global independence*.

Definition 2 *Two transitions n and m are globally independent if and only if they are independent in every state where they are simultaneously enabled.*

We need two other conditions to perform path elimination using independence. First, transitions have to be *invisible* with respect to the checked property, i.e, its execution from any state does not change the value of the propositional variables in the property. Second, eliminating a path

should not eliminate another path that branched out of one of the intermediary states.

Partial order reduction is usually implemented using the concept of *ample* sets. An ample set at state s (denoted $ample(s)$) is a subset of $enabled(s)$. When operating in a partial order reduction mode, the model checker will explore only paths, from a given state, that start with transitions in the ample set rather than the enabled set. In other words, given a state s, partial order reduction results in the elimination of all the paths starting with transitions in $enabled(s) \setminus ample(s)$. According to [3], the following conditions yield a correct ample set (s is a given state):

C0 $ample(s) = \emptyset \Leftrightarrow enabled(s) = \emptyset$;

C1 along every path in the full state graph that starts at s, the following condition holds: a transition that is dependent on a transition in $ample(s)$ cannot be executed without a transition in $ample(s)$ occurring first;

C2 if s is not fully expanded, then every transition in $ample(s)$ is invisible; and,

C3 a cycle is not allowed if it contains a state in which some transition is enabled but is never included in $ample(s)$ for any state s on the cycle.

Note that $C1$ implies that the transitions in $enabled(s) \setminus ample(s)$ are independent of those in $ample(s)$.

Verifying condition **C1** is as hard as checking reachability for the full state space (see Theorem 11, page 154 of [3]). Moreover, the full state space is not available when ample sets are computed during on-the-fly model checking as it is the case in JPF. Therefore, practical implementations of partial order reduction need to use conditions that are easier to check, even if they yield less reduction [3].

JPF relies on another concept based on *safe* transitions [8]. In essence, a transition is *safe* if it is independent on any transition of any other thread. A partial order reduction scheme that uses only safe transitions in the ample set is guaranteed to yield correct results.

Definition 3 *Given a property P, statement n is safe if it is invisible with respect to P and globally independent from any m such that $\theta(n) \neq \theta(m)$.*

Practically, at any given state s, JPF looks for a safe transition in $enabled(s)$ (the set of enabled transitions at s). If it finds a safe transition, say t, JPF executes it and explores the graph it creates. During backtracking, JPF ignores the paths originated in the other transitions ($enabled(s) \setminus \{t\}$). If it does not find any safe transition in the enabled set, all transitions are explored. In other words, the ample set ($ample(s)$) is either the full enabled set ($enabled(s)$) or a singleton containing one of the safe transitions in $enabled(s)$. Contrary to SPIN, JPF relies on a static analyzer to perform the dependence analysis needed to identify safe statements.

2.2 Static Analysis

This section describes how JPF uses static analysis to compute partial order information. We designed our analysis based on the dependences defined in the Bandera toolset [4]. Hatcliff *et al.* defined six types of dependences [6]. There are three intra-thread dependences which are usually found in sequential programs, namely: *data*, *control* and *divergence* dependences. Since these dependences relate statements within the same thread, they cannot be used to identify independent statements (see page 157 of [3]).

Hatcliff *et al.* also define three types of dependences (*interference*, *synchronization*, and *ready* dependences) that capture concurrency issues. The interference dependence captures the fact that shared variables can escape the scope of a given thread. We give the formal definition of interference dependence because it is useful when it comes to computing partial order information.

Definition 4 *A statement n is interference-dependent on a statement m if*

1. *$\theta(n) \neq \theta(m)$, and*
2. *$def(m) \cap ref(n) \neq \emptyset$.*

Note that $def(m)$ and $ref(n)$ need to take into account the possible presence of aliases. Second, it is obvious that, if n is interference-dependent on m, then n and m are not globally independent (they do not commute) with respect to partial order reduction. So, computing interference dependences gives important information about independence with regard to partial order reduction. However, interference dependence is not quite restrictive enough to identify safe statements. Therefore, we take a conservative approach and consider that any statement defining or using (possibly via an alias) a shared variable is unsafe. Our problem then becomes analyzing statically a program to identify shared variables. This requires alias analysis.

Both the synchronization and ready dependence are irrelevant to our discussion on partial order reduction. Synchronization dependence exists to make sure that the monitors enclosing a statement that is in the slice are also in the slice. Since the statements involved in synchronization are in the same thread, they are already considered dependent with regard to partial order reduction, and we do not need to compute if statements are synchronization dependent. The ready dependence states that a statement n is ready-dependent on a statement m if m's failure to complete can make $\theta(n)$ block before reaching or completing n. According to this definition, the execution of m does not disable n; quite the contrary, it enables n. Therefore, ready dependence is irrelevant to the notion of dependence with regard to partial order reduction.

If neither synchronization nor ready dependences are useful to provide information about independence with regard to partial order reduction, we still need to study the synchronization means in Java and analyze their impact on statement independence. For that, we take a closer look at the synchronization commands identified in [6]: namely, *enter-monitor*, *exit-monitor*, *wait*, *notify*, and *notify-all*. The first command (*enter-monitor*) is the only one that can disable statements on different threads. Indeed, the execution of *enter-monitor* prevents other threads to access the lock, potentially disabling transitions on those threads (note that it disables only statements that are attempting to acquire the same lock; once again it is a shared object problem). The *exit-monitor*, *notify*, and *notify-all* commands release locks; therefore, without considering possible exception problems, these commands do not disable other statements; they rather enable some. Finally, the *wait* command does block the execution of some statements, but only on the same thread, which means that it can block only statements that have already been deemed dependent; therefore, we do not need to pay special attention to *wait* commands.

The static analyzer does not really compute what statements are safe. It identifies unsafe statements, and then mark the other statements as safe. Here is the formal definition of an unsafe statement:

Theorem 1 *A statement n is unsafe if*

1. *n is interference dependent on any other statement,*
2. *n is an enter-monitor statement, or*
3. *n is an invoke statement on a synchronized method.*

Conditions 2 and 3 are stronger that they need to be. An *enter-monitor* statement is unsafe only if there is another *enter-monitor* statement on another thread attempting to grab the same lock. Condition 2 takes a conservative approach and mark any *enter-monitor* statement as unsafe. Similarly, condition 3 takes a conservative approach by marking all invocations of synchronized methods as unsafe.

2.3 Implementation

Partial order reduction is performed on-the-fly by JPF based on information computed by the static analyzer. As illustrated in Figure 1, in the current implementation, static analysis is done before model checking. The static analyzer identifies a set of safe statements, which JPF uses to eliminate redundant paths. The static analyzer is run only once (before any run of the model checker) and therefore it provides conservative results; it only computes a subset of the ideal set of safe statements. This guarantees that all interleavings that can affect the result of the verification process are explored by JPF. The precision of the analysis directly affects the number of paths that JPF can ignore. Therefore, approximations made by the static analyzer have a direct impact on the level of reduction achieved by JPF.

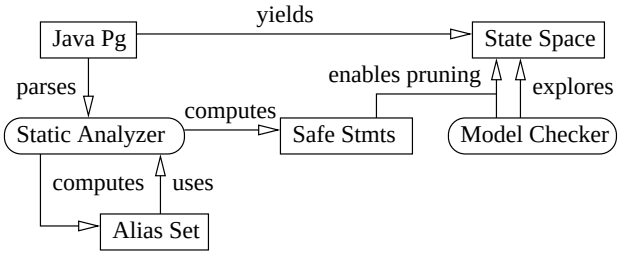

Figure 1. One-time partial order reduction.

Partial order analysis depends on aliasing and call graph analyses; hence, it suffers from the approximation introduced by those analyses. Unfortunately, aliasing analysis is a complex problem [10]. For example, it has problems handling possibly infinite data structures. Practical algorithms rely on k-limiting schemes that differentiate aliases in lists only up to a depth of k [10]. Moreover, scalable algorithms typically do not differentiate between different calling contexts [1, 12]. This leads us to believe that only limited partial order reduction can be achieved by using static analysis as a pre-processing step.

Our driving principle is based on two observations. First, JPF can compute precise aliasing (and call graph) information while the static analyzer can only compute approximate aliasing information. Therefore, JPF should be the one computing the aliasing information. Second, JPF lacks a global view of the synchronization in the system; it usually considers only one path of execution at a time and it does not realize the synchronization problems until it bumps into them. The static analyzer has a more abstract view of the system, and given the proper aliasing information, it does a good job at computing partial order information. Therefore, it should be the one computing the independence sets. Therefore, it seems that, by working on different aspects of the problem, they could help each other and do a better job than they do now. This paradigm is illustrated in Figure 2.

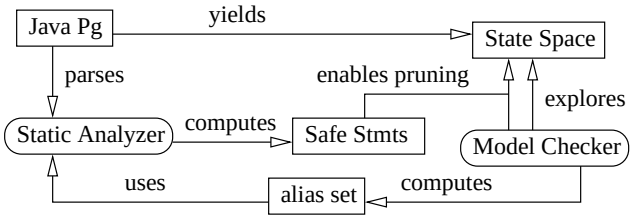

Figure 2. Iterative partial order reduction.

Theoretically, this paradigm is attractive, but, practically, it seems to be only realizable at the expense of safety (i.e.,

the verification is optimistic and does not offer guarantees even if it terminates). Indeed, JPF can only compute partial alias sets as it explores the state space. Only when it has completely explored the state space can it compute precise alias sets. Therefore, it seems that, in our scheme, static analysis would always be based on unsafe alias sets, and hence, produce only overly optimistic sets of safe statements. This implies that JPF could potentially ignore relevant paths, and therefore, miss some errors. Fortunately, this is not so as we prove in the next section.

At each new iteration, JPF adds new aliases to the alias sets it computed during previous iterations. These new aliases may lead the static analyzer to re-classify safe statements as unsafe (but never unsafe statements as safe). Therefore, at each new iteration, the set of safe statements is a subset of the safe statements at previous iterations. Each time a statement is re-classified from safe to unsafe, it forces JPF to consider additional paths. The iteration process starts with no alias sets (or one resulting from a must-alias analysis) and optimistic sets of safe statements. The process goes on until JPF has safely identified all the aliases in the program. At that point, JPF gives an exact alias set to the static analyzer, which in turn can compute an exact set of safe statements. That set is then used by JPF to identify exactly all the relevant remaining paths. The overall verification process then finishes once JPF has covered all relevant paths. All of this is valid because JPF never completely discards a path until the whole verification process is over.

The innovative aspect of our method is that it performs an overall safe analysis out of iterative unsafe analysis steps. Moreover if no approximations are introduced by the static analyzer when computing the set of safe statements, then it results in an optimal reduction of the state space. The state spaces explored by JPF during each iteration (besides the last one) are only partial state spaces. Therefore, stopping at these stages could yield incorrect results. The overall process converges towards an optimal (in terms of partial order reduction) state space.

In the next section, we describe a framework that formalizes this approach. We then use it to prove the correctness of this approach. In particular, we prove that the iteration process cannot falsely detect that an alias set is complete (which could cause an early termination of the state exploration and hence yield incorrect results).

3 Formal Framework

3.1 Aliasing

In this section, we define aliasing and establish results on the alias analysis done by JPF. Formally, an alias at program point t is a pair (u, v) of references ($(u, v) \in R^2$ where R is the set of references in the program) that point to the same store location. In other words, u and v give access to the same object. We can represent all aliases in a program as a mapping of program points to alias sets as follows:

$$\begin{array}{llll} \text{A:} & \text{T} & \rightarrow & 2^{R \times R} \\ & \text{t} & \mapsto & \{(u, v) \in R^2\} \end{array}$$

where $A(t)$ represents the set of aliases holding at program point t. We define $\mathcal{A}$ as the set of all possible alias mapping for a given program. In the rest of the paper, we will refer to $A \in \mathcal{A}$ as the alias set of the program. We also define a natural partial order on $\mathcal{A}$ as follows:

$$\forall (A, B) \in \mathcal{A} : A \subseteq B \Leftrightarrow (\forall t \in T : A(t) \subseteq B(t)).$$

In the presence of dynamic information about finite programs, $\mathcal{A}$ is finite. The number of references is bounded, and therefore, there is a finite number of alias combinations.

We assume that at any given time during the analysis, we can determine the set of states S^e that have been explored by JPF. We also assume that JPF can reliably compute the set of aliases holding at any given state. Let f_a ($f_a : S \rightarrow \mathcal{A}$) be the function that allows JPF to extract aliases from a set of explored states. Thus,

$$f_a(S^e) = A$$

where S^e is the set of explored states and A is the set of aliases for the current set of explored states. It is obvious that f_a is monotonically increasing.

$$\forall (X, Y) \in S^2 : X \subseteq Y \Rightarrow f_a(X) \subseteq f_a(Y) \qquad (1)$$

Indeed, exploring more states does not eliminate aliases, it only adds more aliases.

3.2 Safe Statement Analysis

We assume that the static analyzer has a consistent means of computing the set of safe statements given an alias set. Let f_s ($f_s : \mathcal{A} \rightarrow T$) be such a function. Thus,

$$f_s(A) = I$$

where A is the current set of aliases in the program and I is the set of safe (independent) statements that can be inferred given A. Obviously, f_s is monotonically decreasing.

$$\forall (X, Y) \in \mathcal{A}^2 : X \subseteq Y \Rightarrow f_s(Y) \subseteq f_s(X) \qquad (2)$$

Adding an alias may create a new dependence among statements, but it does not remove any. Therefore, it can only change a safe statement into an unsafe statement and can never change an unsafe statement into a safe statement.

Furthermore, let f_r be the function ($f_r : T \rightarrow S$) that, given a set I of safe statements, can compute the largest set of states $S^M = f_r(I)$ that can be explored by JPF. It is obvious that, at any given time, the set of state explored by JPF (say S^e) is a subset of S^M.

$$S^e \subseteq S^M \subseteq S$$

3.3 Fixed Point Analysis

The analysis uses an iterative process which sees the static analyzer compute independence sets and JPF bounds on state exploration and alias sets. The process goes as follows (note: indices relate to iterations):

1. given an alias set A_{k-1}, the static analyzer computes an independence set $I_k = f_s(A_{k-1})$ while JPF is exploring a confined state space ($S^e_{k-1} \subset S^M_{k-1}$);
2. then JPF uses I_k to limit its exploration to a new confined state space ($S^M_k = f_r(I_k)$) by performing partial order reduction;
3. JPF computes a set of aliases A_k given the state space S^e_k it has explored ($A_k = f_a(S^e_k)$ and $S^e_k \subseteq S^M_k$);
4. the process goes back to Step 1 unless a fixed point has been reached ($A_{k+1} = A_k$).

To show that a fixed point is reached, we define $C = (\mathcal{A} \times T \times S \times S)$ where any $c \in C$ consists of an alias set, a set of safe statements, and confined and explored state spaces. We also define a partial order relation $\sqsubseteq$ over C:

$$\forall c_1 = (A_1, I_1, S^M_1, S^e_1), c_2 = (A_2, I_2, S^M_2, S^e_2) \in C : \\ c_1 \sqsubseteq c_2 \Leftrightarrow \\ (A_1 \subseteq A_2) \wedge (I_1 \supseteq I_2) \wedge (S^M_1 \subseteq S^M_2) \wedge (S^e_1 \subseteq S^e_2)$$

The fact that $\sqsubseteq$ defines a partial order derives directly from the fact that $\subseteq$ is a partial order relation. Therefore, $(C, \sqsubseteq)$ is a partial ordered set (or poset). It is also easy to show that

Lemma 1 *$(C, \sqsubseteq)$ is a complete lattice.*

Proof: First, note that C is finite because T, S, and $\mathcal{A}$ are finite. Therefore, we only have to show that the supremal ($\sup Y$) of a set $Y \subseteq C$ is in C (and similarly for the infimal of Y, $\inf Y$). This is trivial since $\sup C = (A, \emptyset, S, S) \in C$ and $\inf C = (\emptyset, T, \emptyset, \emptyset) \in C$.

■

We now define a function f ($f : C \rightarrow C$) that summarizes the effects of an iteration. Thus,

$$\forall c = (A, I, S^M, S^e) \in C : \\ f(c) = (g \circ f_a(S^e), f_s(A), f_s \circ f_r(A), S^e)$$

It is also trivial to show that, for any iteration k,

$$(A_k, I_k, S^M_k, S^e_k) = f((A_{k-1}, I_{k-1}, S^M_{k-1}, S^e_{k-1}))$$

and that, using (1) and (2), f is monotonically "increasing".

$$\forall (c_1, c_2) \in C : c_1 \sqsubseteq c_2 \Rightarrow f(c_1) \sqsubseteq f(c_2) \qquad (3)$$

Therefore, we can show that

Theorem 2 *f has a supremal fixed point*

Proof: All we have to show is that

- f is a monotone function (see (3))
- f is defined over a complete lattice $(C, \sqsubseteq)$ (see Lemma 1). Then, according to Theorem 2.1 in [9], f admits a supremal fixed point.

■

At this point, we have shown that our iterative verification process will converge to a fixed point. We still have to show that this fixed point corresponds to a correct analysis.

Theorem 3 *After termination of the iterative process, all relevant paths have been explored.*

Proof: Let $P(S)$ be the set of paths in the full state space S. Let $P(S_A)$ be the set of paths in the explored states space. Let T be the set of all possible transitions. Let $s(p)$ be a function that returns the final state of a path p.

Assume that there exist paths that should have been explored but have been discarded by JPF based on the partial order information given by the static analyzer. Let p be the first such path that could have been encountered by JPF.

$$p \in P(S) \setminus P(S_A).$$

Let $p' \in P(S_A)$ be the explored path that "caused" p to be discarded. Then, there exist two transitions t and t' and three sub-paths p_1, p_2, and p'_2 such that

- $p = p_1.t.p_2$,
- $p' = p_1.t'.p'_2$,
- $t, t' \subset enable(s(p_1))$, and
- t' was misclassified as safe.

If t' is not safe, then either t' is visible by the property being checked or it is not globally independent.

It is not possible for t' to be visible by the property because it would imply that an alias at t' had been missed even though t' was executed by JPF; this violates the assumption that alias discovery on states visited by JPF is correct.

Then, t' must not be globally independent. Moreover, since p should have been explored, it means that t and t' are not independent at state $s(p1)$. This implies that t and t' access a shared variable (or lock) through aliases and that at least one of these aliases is never uncovered during the analysis. However, such aliases result from the execution of the last transition in p_1 (i.e., the transition preceeding t in p and t' in p'). Since JPF has executed that transition, these aliases would have been found by JPF (again, using the assumption that alias discovery on states visited by JPF is correct).

Therefore, there are no aliases accessing a shared variable from t and t', and t' is globally independent.

Therefore, p was rightly discarded by JPF. We can repeat this reasoning for every (wrongly discarded) path and prove that all the relevant paths have been explored by JPF.

■

4 Example

The example in Figure 3 is a simple multi-threaded program that exhibits a subtle deadlock due to incorrect initialization of variables. Essentially the deadlock is due to the count variables in *Task1* and *Task2* not being initialized to zero - the important statements that would need to be interleaved in order to find this error are the assignments to the count variables just prior to the while loop in each thread (i.e., statements labeled u_1 and u_2 in Figure 3).

```
class Main {
  static void main(String[] args) {
    Event new_ev1 = new Event();
    Event new_ev2 = new Event();
    Task1 task1 =
      new Task1(new_ev1,new_ev2);
    Task2 task2 =
      new Task2(new_ev1,new_ev2);
    task1.start();
    task2.start();
  }
}

class Event {
  int count = 0;
  synchronized void
      wait_for_event(int rcount) {
    if (rcount == count)
    try {wait();}
    catch(InterruptedException e){};
  }
  synchronized void signal_event() {
    count = (count + 1) % 3;
    notifyAll();
  }
}

class Task1 extends Thread {
  Event event1,event2;
  int count = 0;
  Task1(Event e1, Event e2) {
    this.event1 = e1;
    this.event2 = e2;
    this.start();
  }
  void run() {
u1 : count = event1.count;
    while(true) {
      event1.wait_for_event(count);
      count = event1.count;
      event2.signal_event();
    }
  }
}

class Task2 extends Thread {
  Event event1,event2;
  int count = 0;
  Task2(Event e1, Event e2) {
    this.event1 = e1;
    this.event2 = e2;
    this.start();
  }
  void run() {
u2 : count = event2.count;
    while(true) {
      event1.signal_event();
      event2.wait_for_event(count);
      count = event2.count;
    }
  }
}
```

Figure 3. Java program with deadlock.

During the first iteration, the static analyzer does not have any aliasing information. Let us examine the class $Task1$. Even though the analyzer can determine that the fields $event1$ and $event2$ are probably aliased to objects on different threads, it has no concrete proof that it is happening (because it has no alias information at this point). Therefore, the analyzer has no choice but to mark all the statements dealing with those variables as safe, except for the statements invoking the synchronized methods $wait_for_event$ and $signal_event$. In these cases, the analyzer can infer that these invocations will ask for locks (even though it does not know which locks) and we mark such calls as unsafe (without checking what object is used as a lock). A similar analysis is done for the class $Task2$. All statements in class $Event$ are marked as safe (and will remain so since no aliases can be generated in the $Event$ class). All statements in class $Main$ are marked as safe.

During the first exploration done by JPF, the following aliases (among others) will be uncovered. The references $Main.new_ev1$, $Task1.event1$, and $Task2.event1$ will be aliased when $task1$ and $task2$ start running. Similarly, the references $Main.new_ev2$, $Task1.event2$, and $Task2.event2$ will be aliased when $task1$ and $task2$ start running. Since the statements u_1 and u_2 (in bold in the figure) have been marked as safe, they are not interleaved and JPF does not find the deadlock. Still, JPF passes important alias information to the static analyzer.

During the second iteration, some alias information, including the aliases concerning the fields $event1$ and $event2$ in both $Task1$ and $Task2$ will be known to the static analyzer. Using this information, the analyzer can now mark as unsafe any statement accessing the fields $event1$ and $event2$ in both $Task1$ and $Task2$ (including statements u_1 and u_2). This results in only one safe statement in each of the thread: the condition of the while loops.

This new information is then communicated to JPF which realizes it needs to take into account some additional interleaving. This results in JPF interleaving statements u_1 and u_2 and in the deadlock being found.

	no p.o.r.	p.o.r. u_1, u_2 safe	p.o.r u_1, u_2 unsafe
States	866	339	432
Transitions	1489	459	632
Deadlock	found	not found	found

Figure 4. State space reduction figures.

Table 4 shows the results of running JPF without partial order reduction and with different information about safe statements. Column 1 shows the results when JPF does not use any partial order reduction. Column 2 shows the results obtained by JPF with partial order reduction using an overly optimistic set of safe statements. Column 3 shows the results obtained by JPF with partial order reduction using a correct set of safe statements. Since the example is quite small, JPF can find the deadlock without partial order reduction, but, to do so, it has to visit twice as many states as when it performs partial order reduction with the correct safe statements. When the information about safe statements is incorrect (as in the first iteration) the deadlock is not found. But the discovery of aliases by JPF leads to a correct set of safe statements, and, JPF eventually finds the deadlock after visiting roughly 100 more states.

5 Practical Considerations

Our technique relies on the fact that JPF always finishes the exploration of a graph once it has started it. This is not true in general either; explicit state model checkers often run out of memory before they can finish exploring the whole space. If it is the case, our iterative technique will perform an unsafe analysis. However, in its normal processing mode, JPF would have run out of memory anyway. So, the result of its analysis would be unsafe too. Therefore, our method does not do worse than the normal processing mode. We might even argue that it would cover a "more relevant" state space before it runs out of memory.

A final important consideration is the fact that JPF, as most explicit-state model checkers, stops at the first error it finds and returns with an error trace (in JPF's case, a full execution trace). Therefore, by constraining the model checker to stay within a confined state space (at each iteration), we restrict the possibility of the model checker to get lost in some uninteresting part of the state space. It would be interesting to extend the static analysis so that the model checker is guided towards critical areas first.

6 Conclusion

We have described an iterative process to perform software verification using static analysis and explicit-state model checking. Static analysis identifies safe statements which are used by the model checker to perform partial order reduction (thus increasing the chances of exhaustive exploration). The model checker performs state exploration (using partial order reduction) and computes precise alias sets used by the static analyzer to identify safe statements. This work is innovative for the following reasons.

- It describes a framework where static analysis and model checking work concurrently to improve the coverage and the precision of software verification; we are aware of only one other work [5] using model checking and static analysis concurrently; but it is based on abstract interpretation and abstract model checking.
- It describes an iterative process that computes unsafe intermediate results until it converges to a safe result; traditional research ideas in static analysis and explicit-state model checking try to improve on safe solutions by reducing the impact of the approximations.
- This method yields a solution that is much closer to the optimal solution (both in terms of aliasing and partial order reduction) than traditional methods.

However, we pay the price of losing some of the generality in the results of the static analysis. Since we incorporate dynamic information, the results of the static analysis cannot be re-used for different inputs (environment) of a program as it was the case in the traditional linear process.

Our future plans are to improve the current implementation, to experiment with this technique on many more Java programs, and to define a rationale for the duration of the iteration steps. Future improvements include the use of on-demand (or compositional) algorithms in the static analysis. We expect that, as we experiment with other Java programs, we might have to refine the distribution of work between the static analyzer and the model checker. We will also consider other types of interactions between the model checker and the static analyzer, and possibly other techniques.

References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder - a second generation of a java model checker. In *Workshop on Advances in Verification*, july 2000.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[4] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera : Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22nd International Conference on Software Engineering*, Limeric, Ireland., June 2000. ACM Press.

[5] P. Cousot and R. Cousot. Parallel combination of abstract interpretation and model-based automatic analysis of software. In *Proceedings of the First ACM SIGPLAN Workshop on Automatic Analysis of Software, AAS'97*, pages 91–98, 1997.

[6] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with jvm concurrency primitives. In *Proceedings on the 1999 International Symposium on Static Analysis*, pages 1–18, 1999.

[7] G. Holzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[8] G. Holzmann and D. Peled. An imporovement in formal verification. In *Proceedings on the Formal Description Techniques Conference*, pages 197–211, 1994.

[9] R. Kumar and V. K. Garg. *Modeling and Control of Logical Discrete Event Systems*. Kluwer Academic, 1995.

[10] W. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers University, 1992.

[11] J. Penix, W. Visser, E. Engstrom, A. Larson, and N. Weininger. Verification of time partitioning in the deos real-time scheduling kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, june 2000.

[12] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of theACM Conference on Principles of Progamming Languages*, 1996.

[13] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the International Conference on Automated Software Engineering*, sep 2000.