Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 144 (2006) 41–56
www.elsevier.com/locate/entcs

VyrdMC: Driving Runtime Refinement Checking with Model Checkers

Tayfun Elmas[1] Serdar Tasiran[2]

College of Engineering
Koç University, Istanbul, Turkey

Abstract

This paper presents VyrdMC, a runtime verification tool we are building for concurrent software components. The correctness criterion checked by VyrdMC is refinement: Each execution of the implementation must be consistent with an atomic execution of the specification. VyrdMC combines testing, model checking, and Vyrd, the runtime refinement checker we developed earlier. A test harness first drives the component to a non-trivial state which serves as the starting state for a number of simple, very small multi-threaded test cases. An execution-based model checker explores for each test case all distinct thread interleavings while Vyrd monitors executions for refinement violations. This combined approach has the advantage of improving the coverage of runtime refinement checking at modest additional computational cost, since model checkers are only used to explore thread interleavings of a small, fixed test program. The visibility and detailed checking offered by using refinement as the correctness criterion differentiate our approach from simply being a restricted application of model checking. An important side benefit is the reduction in program instrumentation made possible if VyrdMC is built using a model checker with its own virtual machine, such as Java PathFinder [8]. We are investigating the use of two different model checkers for building VyrdMC: Java PathFinder, an explicit-state model checker and Verisoft, a "stateless" model checker [7].

Keywords: Concurrent Software, Runtime Verification, Refinement Checking, Model Checking

1 Introduction

Many software applications are built as collections of interacting, concurrently-accessed software components. Verifying industrial-scale concurrent software is particularly hard because of the added complexity due to different possible

[1] Email: telmas@ku.edu.tr
[2] Email: stasiran@ku.edu.tr

1571-0661 © 2006 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2006.02.003

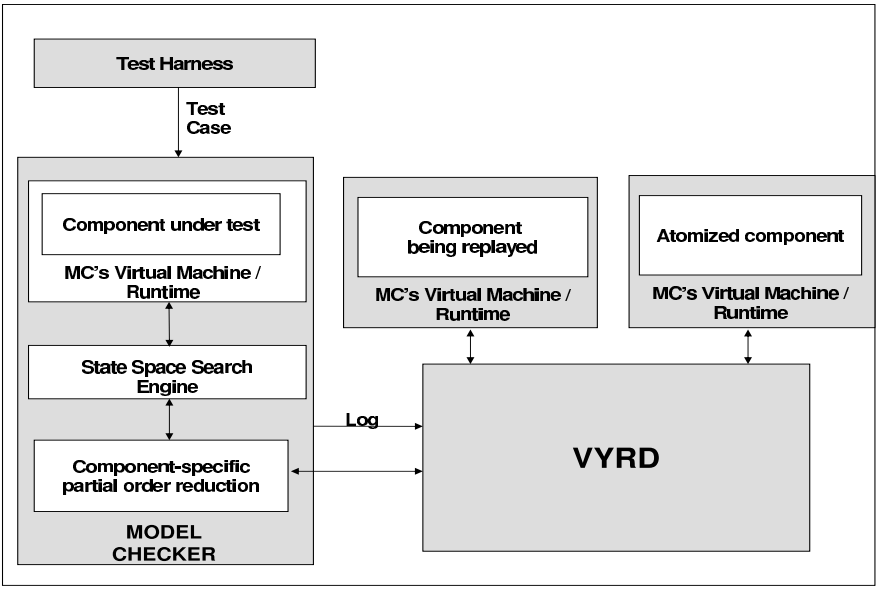


Fig. 1. The architecture of VyrdMC

interleavings of threads accessing shared data. This paper proposes a hybrid technique that integrates the strengths of runtime refinement verification and model checking into a validation tool for concurrent software.

Execution-based model checking of such systems has limited applicability due to the state-space explosion problem if abstractions are not performed. Model checking using abstractions is good at verifying local properties. However, more global specifications such as refinement require most implementation details to be included in the implementation model and abstraction techniques do not accomplish much in this case. Pure testing, while computationally practical, suffers from low coverage and limited observability. The latter weakness was addressed by our previous work on runtime refinement checking [5,6].

Refinement formalizes the requirement that method invocations by concurrent threads appear to be executed atomically in a linear order and are consistent with an executable specification. Each execution of the implementation must be "equivalent" to an execution of the specification, which can be given as a separate program or be obtained by constructing an "atomized" version of the implementation. In [5,6], we investigated two notions of refinement: I/O and view refinement. The latter of these requires a particular correspondence between the implementation and specification states, which enables detection of concurrency errors as soon as they cause a discrepancy in the implementation state. Experimental results on industrial-scale designs showed that the added observability of view refinement makes the early de-

tection of concurrency errors much more likely.

While it provides significantly better verification than testing, runtime refinement checking also suffers from low coverage. If concurrency errors are not triggered by a test program, refinement checking is no better than testing. In our experience, to trigger a particular concurrency error, tests with many concurrent threads, each performing a large number of method calls, were often needed, while the same error could have been triggered with many fewer threads and method calls if the correct (but unknown a priori) thread interleaving were enforced. This suggested that very small test programs might be able to reveal concurrency errors if all thread interleavings are systematically explored and view refinement checking is performed. This observation motivated the technique presented in this paper.

In the proposed tool, called VyrdMC, an execution-based model checker guides the testing of concurrent software by exploring all distinct thread interleavings for a given fixed test program. We focus on model checkers that execute programs as is, without performing any abstraction. Clearly, such model checkers could be used to drive programs with the purpose of performing any runtime check. What distinguishes our approach (depicted in Figure 1) is the combination of the strengths of the following three approaches:

(i) Testing is used to drive Vyrd to non-trivial, sufficiently large and complicated "anchor" states. At anchor states, the specification and implementation are consistent. Starting at the anchor state, the test program issues a very small number of threads with few operations each. In each test case, the operations and their arguments are chosen so that they focus on the same portion of the state of the program and are intuitively likely to contend for access to resources and to trigger concurrency errors.

(ii) Our refinement checking tool Vyrd provides improved observability, which makes it possible to detect errors triggered by the small test cases starting from the anchor state. During the course of such short tests, errors are unlikely to propagate to a method's return value.

(iii) Model checking tackles the coverage problem by generating all qualitatively distinct execution traces starting from the anchor state. The model checker needs to only explore different thread interleavings since everything else is fixed. This very restricted application of model checking and the choice of very small test programs makes our technique computationally feasible even for industrial-scale programs.

Previous work, notably [2], has explored techniques that blend these techniques in different ways. Our approach is distinct in that it is able to handle industrial-scale software as is, without need for abstraction. The use of re-

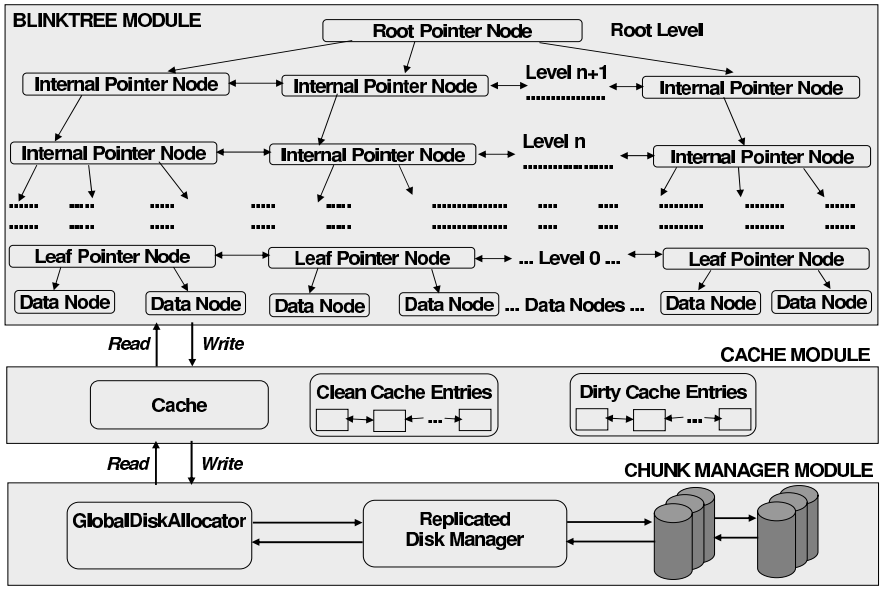


Fig. 2. The layered architecture of the components in Boxwood

finement as a correctness criterion that is both less restrictive than and as comprehensive as atomicity and linearizability further distinguishes our approach.

A side benefit of using a model checker such as Java PathFinder is the re-use of its runtime environment – essentially a virtual machine (VM) – to automate monitoring and logging tasks that are required for refinement checking and that had to be performed manually previously. An example is the detection and recording of updates to shared variables. Automating this phase makes refinement checking a lot more easily usable. Vyrd also uses the runtime environment of the model checker for its replaying mechanism, i.e., driving independent instances of the implementation and the specification for verification purposes. We discuss the benefits of and issues that arise when implementing VyrdMC using two kinds of execution-based model checkers: an explicit-state model checker like (JPF) [8], and a "stateless" model checker like Verisoft [7].

This paper is organized as follows. Section 2 introduces our motivating example, BLinkTree: a concurrently-accessible B-link tree implementation. Section 3 describes our testing strategy. How Vyrd checks refinement and is adapted to run in conjunction with the model checker is explained in Section 4. Section 5 explains how thread scheduling is controlled by the two categories of model checkers. Section 6 presents complexity reduction techniques we are investigating in VyrdMC. Section 7 concludes the paper and outlines future work.

2 Motivating Example: The BLinkTree Module

This section introduces a component of an industrial-scale storage infrastructure project, Boxwood [3], which drove the development of our runtime verification framework. Boxwood consists of three modules as seen in Figure 2: BLinkTree, Cache and Chunk Manager. The BLinkTree module is an implementation of the concurrent operations for the B-link tree data structure presented in [4]. The information associated with each BLinkTree node is stored in a shared variable, which is a variable-length byte-array. BLinkTree uses the Cache module to store and retrieve byte-arrays. Cache makes its data persistent through the distributed Chunk Manager module.

The BLinkTree stores `(key,data)` pairs, and provides methods to add, delete and look up entries. In addition, an internal compression thread works concurrently with data structure operations and re-organizes the tree by redistributing the contents between pairs of neighboring nodes. In order to improve performance, BLinkTree operations use a very relaxed locking scheme which allows operations from different methods to overtake each other.

The complicated B-link tree structure, the large number of state variables required for a non-trivial B-link tree instance, and highly concurrent operations makes the verification of the BLinkTree module challenging. Exhaustive state-space search techniques are made impractical for this purpose because of the state-space explosion problem. Abstraction techniques would not be of much help since verifying the effects of mechanisms such as overtaking and compression would require the inclusion of almost all implementation state in the abstract model.

In our earlier work, we showed that refinement checking using the Vyrd tool to be an effective verification approach for BLinkTree without requiring any abstraction [6]. However triggering concurrency errors in BLinkTree and other industrial-scale software typically required running many long tests with large numbers of threads because the thread scheduling mechanisms of the language runtimes were used with no modification. In each of these cases, we observed that the concurrency errors could be triggered with few concurrent threads each containing a single method call and starting from almost any non-trivial state provided that threads were scheduled in one particular way. This need to exercise more control on and systematically explore the thread interleavings to increase the coverage of runtime refinement checking motivated the technique described in this paper.

3 The Test Harness

This section explains the test harness, which is designed to keep the resource requirements of our technique small.

3.1 Test Cases

The construction of the test cases is largely manual. Each multi-threaded test case starts from a small number of pre-constructed instances of a concurrently-accessed software component, $\mathcal{C}$, denoted by $\mathcal{C}_1, \mathcal{C}_2, ..., \mathcal{C}_k$. During the construction of each $\mathcal{C}_i$, refinement checking is performed, and the corresponding specification state $\mathcal{S}_i$ is computed. For the instances used in each test case, Vyrd uses the $(\mathcal{C}_i, \mathcal{S}_i)$ pairs as the starting point.

Each test case $\mathcal{T}$ consists of a small number of threads (typically two or three) each performing one method call on a component. The arguments to the method calls are selected to focus method executions on the same portion of the component state.

For example, a single BLinkTree instance that includes instances of all tree node types may serve as a non-trivial starting point to the test case. Each test case is then generated by selecting a number of `Insert`, `LookUp` and `Delete` method invocations, each run in a separate thread. The `key` arguments to the methods are selected in order to make them work on the same pointer/data node, or neighboring nodes.

3.2 Modeling Executions

We model executions as sequences of atomically executed code fragments each of which is called an *action*. The sequence of actions that are taken during an execution are called its *trace*. Actions fall into the following two categories:

Atomic updates to a component's shared variables: These are updates to the fields of shared objects that are part of the component being verified. Fields can be of primitive or reference types. It is assumed that such updates are guaranteed to be atomic by the language and the runtime environment.

Method calls to other components: The component under test, $\mathcal{C}$, may be making use of other components, which, for the purposes of verifying $\mathcal{C}$, are considered to have correctly implemented atomic operations. The developer assumes that all modifications to and observations of the instances of these components are performed atomically and thus they are called *assumed-atomic objects*. Then each method call to an assumed-atomic object is treated an action. This enables refinement checking layered components using a limited form of assume-guarantee reasoning as described in Section 6.2.

Example: During verification of BLinkTree, the byte arrays allocated from

Chunk Manager and temporarily stored in Cache are the shared objects. While verifying BLinkTree, Cache and Chunk Manager are assumed to be implemented correctly, thus, each method of Cache and Chunk Manager operating on a byte-array is considered to be an atomically executed action. While verifying the Cache module, on the other hand, each cache entry is treated as a shared object and each byte-code instruction applied to a field of a cache entry is considered an atomic action.

3.3 Execution Tree

Because of the way test cases are constructed, the only non-determinism in executions of the implementation is due to different thread interleavings. The possible traces for a given test case are each of finite length under the assumption that methods terminate. Violations of this assumption, i.e., deadlocks or cycles in the state space, can be detected by the model checker at the same time runtime refinement checking is being performed. In the rest of this paper, we assume that the component being verified is free of deadlocks and that each of its methods terminate.

We can then represent the set of possible executions of a test case $\mathcal{T}$ by a finite *execution tree*. An execution tree $\mathcal{E}_\mathcal{T}$ is a directed acyclic graph each vertex of which is labeled with a state of $\mathcal{C}$ and each branch which is labeled by an action. The root of is the initial state of $\mathcal{C}$. Each trace τ corresponds to a unique *path* in $\mathcal{E}_\mathcal{T}$ from the root to a leaf.

4 Runtime Refinement Checking with Vyrd

This section gives an overview of the runtime refinement checking tool Vyrd. Figure 3 shows the building blocks and the phases of the verification technique implemented in Vyrd. The reader can refer to [6] for details. Vyrd analyzes traces generated by the component being tested and checks their conformance to a specification. Specifications are required to execute methods atomically, sequentially. Vyrd allows the use of an "atomized version" of the component (one where a single global lock serializes all method executions) as a specification.

An instance of $\mathcal{S}$, the executable specification for $\mathcal{C}$, is driven by Vyrd to serve as a reference as described below. Vyrd also creates an additional instance of $\mathcal{C}$, denoted by $\mathcal{C}_{replay}$, and replays the recorded actions on it.

Vyrd communicates with the program under test via a log. A log is a data structure into which a sequence of entries with information on the actions performed is written in the order they occur. In Vyrd, the logging mechanism is implemented manually for each component, and the component under test is instrumented to notify the log when an action that needs to be recorded

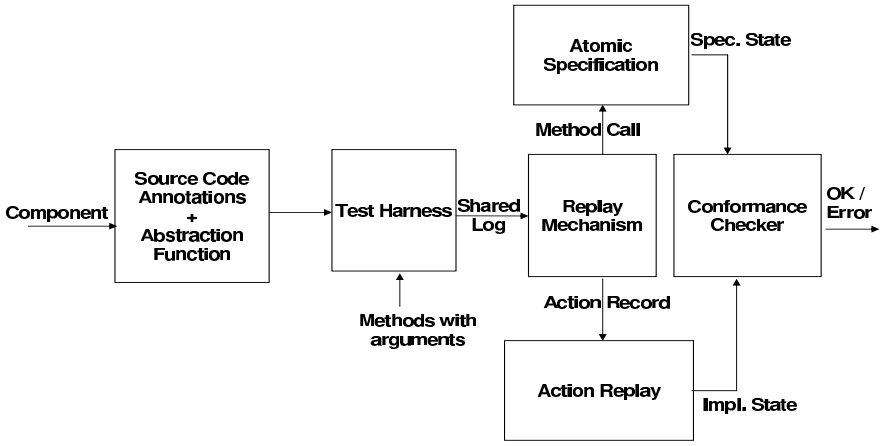


Fig. 3. The architecture of Vyrd

takes place. The log is implemented in order to minimize the impact on the concurrency behavior of the component and to de-couple the verification thread in order to enable off-line operation.

The replaying mechanism of Vyrd reads sequentially from the log the actions performed on $\mathcal{C}$ during testing and applies the same actions on $\mathcal{C}_{replay}$ in the same order. This replayed version's actions and abstracted state are compared at designated points (commit points) for each method invocation with those of $\mathcal{S}$. The linear order in which methods are invoked in $\mathcal{S}$ is the order of the method commit points in the log. The I/O refinement criterion requires that the sequence of method calls and returns per thread match for the specification and implementation traces. A stronger correctness criterion that requires a match between the abstract states of $\mathcal{C}_{replay}$ and $\mathcal{S}$ at commit points in addition is called view refinement. Vyrd signals an error if any refinement violations are detected and reports execution traces for $\mathcal{C}$ and $\mathcal{S}$ that lead to the error.

View refinement provides more observability into program state than I/O refinement. Discrepancies in implementation state immediately lead to view refinement violations, while, during a particular execution, the discrepancy may be missed by I/O refinement if it does not happen to lead to a different method return value. In [6], view refinement was shown to result in better error detection using much small test cases whose execution trees can be handled by the model checker easily.

5 Using Model Checkers to Drive Refinement Checking

Using model checkers serves two purposes in VyrdMC. First, a model checker traverses the paths in an execution tree $\mathcal{E}_{\mathcal{T}}$. If the model checker has support for partial-order reductions, then only paths whose concurrency behaviors are

qualitatively distinct are explored. Second, a model checker provides a runtime environment which helps VyrdMC in automating identification and handling of actions during trace generation and while replaying them for refinement verification. Currently, in the Vyrd tool these are performed via manual source or byte-code instrumentation.

We are working on two versions of VyrdMC, one around Java PathFinder, the other around Verisoft. In order not to limit the discussion of the techniques by the current features supported by these two model checkers, we describe the key features of and issues in VyrdMC using two generic hypothetical model checkers, *ExplicitMC* and *StatelessMC* instead. *ExplicitMC* and *StatelessMC* embody the distinguishing features of Java PathFinder and Verisoft, respectively. We assume that both *ExplicitMC* and *StatelessMC* are capable of (*i*) traversing the execution tree of a test case by performing a depth-first (DFS) or breadth-first (BFS) search, (*ii*) executing test programs as is without performing any abstraction, and (*iii*) making use of partial order reduction techniques to prune equivalent paths [9].

5.1 The Explicit State Model Checker: ExplicitMC

ExplicitMC is an explicit state model checker which stores states that it visits. The entire representation of the state, which contains all of the memory space including the heap and the stacks of the threads, is stored. *ExplicitMC* does state space exploration through a backtracking mechanism. It stores at certain points during execution the states to be returned to. Once the execution reaches a pre-defined depth, *ExplicitMC* backtracks to a previously saved state **s** and takes a previously unexplored branch in the execution tree from **s**. The new direction is taken by running an action from a thread previously not chosen by the model checker at that state. Note that the run-time of this algorithm is proportional to the number of explored actions in the execution tree since each action in the tree is executed only once.

ExplicitMC has a virtual machine (VM) that interprets the byte code instructions of the program under test one at a time, in the same way JPF has an underlying JVM that interprets Java *.class* files. The VM manages all the memory space of the program including all the objects and fields. During testing, the operation of VM is intercepted to identify the shared objects to take necessary actions at each invocation of a primitive instruction or a method call on those objects. The JPF API exports classes and methods for accomplishing this.

5.2 The Stateless Model Checker: StatelessMC

StatelessMC is a "stateless" model checker, i.e., it stores no state representations in memory but only information about which transitions and paths of the execution tree have been traversed so far. One example of such a model checker is Verisoft [7], which was developed for verifying `C/C++` programs.

While traversing the execution tree $\mathcal{E}_{\mathcal{T}}$, *StatelessMC* starts executing the program from an initial state, the root of $\mathcal{E}_{\mathcal{T}}$, and it drives the execution until it reaches a state where it takes an unexplored branch. The state-less search mechanism is slower than the the mechanism of *ExplicitMC* since it executes many branches in $\mathcal{E}_{\mathcal{T}}$ multiple times. However, because it does not store states, it requires considerably less amount of memory, especially when the program being verified has a large number of variables.

Unlike *ExplicitMC* , *StatelessMC* has no VM, so it instruments the source code or the byte code to control the execution of the program under test and to produce calls to the model checker. This instrumentation is done automatically and the instrumentation mechanism can be modified in order to insert extra calls by the model checker to the refinement checker.

5.3 Monitoring Executions

Since runtime environments for *ExplicitMC* and *StatelessMC* differ, their handling of shared variables and actions differs as well. If *ExplicitMC* is used as the model checker, the VM in *ExplicitMC* can be used to track the shared objects. Once an object is created during a method execution of $\mathcal{C}$, the VM marks that object as a shared object and monitors actions on it. *ExplicitMC* monitors byte-code instructions of the VM on the shared objects. Each invocation of such an instruction is intercepted just after it is executed by the VM and information is recorded in the shared log. The assumed-atomic objects are also tracked and the method calls to those objects are run atomically by the VM. JPF provides a mechanism for performing calls to VyrdMC code after relevant events such as those listed above.

StatelessMC extracts the expressions or instructions that create and manipulate the shared objects via assignments to their fields. However, since *StatelessMC* does not provide a virtual machine for replaying actions, the logging and replaying mechanisms are implemented separately, outside *StatelessMC* . Implementation source code is instrumented manually for logging actions and for proper handling of method calls to assumed-atomic objects.

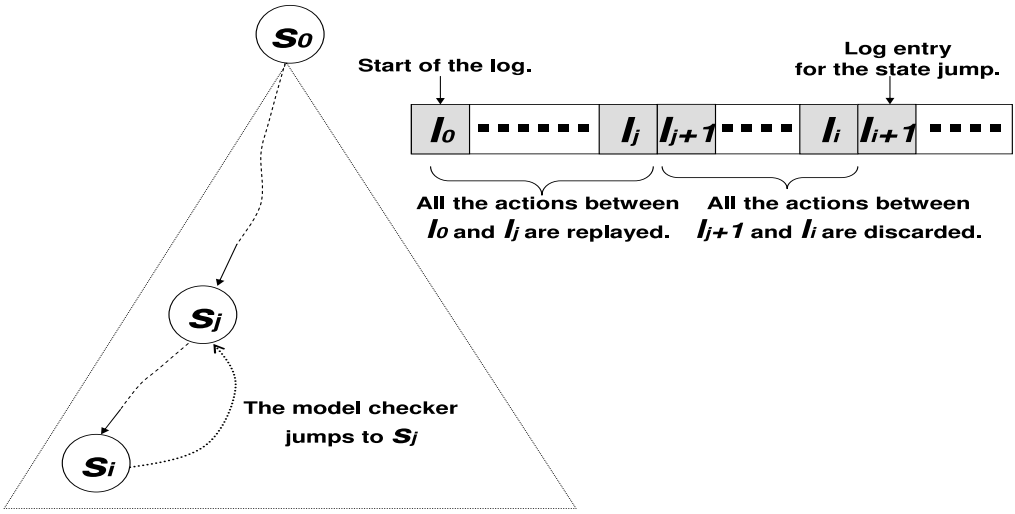


Fig. 4. An execution tree and the contents of the corresponding log.

5.4 Coordinating State-Space Exploration and Vyrd

Vyrd communicates with the program under test through a shared log, therefore, its operation remains mostly the same. The following modifications are applied to Vyrd (*i*) to coordinate its operation with the state space exploration process, and (*ii*) when possible, to automate the replay mechanism of Vyrd using the runtime environment of the model checker.

Vyrd was built to check refinement for a single linear trace that starts from the initial state and contains no backtracking or jumps. It assumes that each state transition in the trace is caused by the execution of an action and each point in the log corresponds to a unique state obtained by executing the actions in the log up to that point. Running the design in a model checker breaks this assumption. Model checkers may move to a previously visited state that is not necessarily reachable from the current state by taking an action of the program.

To coordinate Vyrd with the model checker's traversal of the execution tree, first, nodes in the execution tree are given unique ID's as follows. At every internal node of the tree that has more than one child, the children are numbered (starting from 0) in the order they are visited by the model checker. Then, each tree node n is uniquely labeled by a string of integers consisting of the numbers of the children on the path leading to n from the root of the execution tree.

A new kind of entry representing jumps in the state space is added to the log. As a result, the log consists of two kinds of entries:

- *Actions:* These are actions performed by the program. This corresponds to traversing an edge in the execution tree in the parent to child direction. Each such action is labeled by the number of the child.
- *Jumps:* These correspond to jumps performed by the model checker. A jump entry contains the unique ID of the position in the log that corresponds to the state being jumped to.

Figure 4 shows a simple execution tree and the contents of the corresponding log. Suppose that the model checker jumps from node n_i where the program is at state $\mathbf{s}_i$ to a previously visited node n_j with corresponding program state $\mathbf{s}_j$. Suppose that σ_i and σ_j are the integer strings that uniquely identify n_i and n_j. When the jump occurs, a log entry λ_{i+1} is inserted at position $\mathbf{l}_i+1$ that indicates that Vyrd should restore its state to what it was when n_i was first encountered. If *ExplicitMC* is being used, Vyrd accomplishes this by (i) resetting all of its internal state including the states of $\mathcal{C}_{replay}$ and $\mathcal{S}$ states to their initial states, and (ii) replaying the log from the beginning until after the jump log entry λ_i, skipping over actions that do not lie on the path from the root to $\mathbf{l}_j$. Since the execution path up to $\mathbf{s}_j$ has been checked for refinement violations in a previous execution, Vyrd does not perform any checking until it reaches $\mathbf{s}_j$. It resumes refinement checking as usual starting from $\log_{i+1}$.

If *StatelessMC* is used to traverse the execution tree, there is no need to re-start reading of the log from the beginning since *StatelessMC* performs the jump by restarting the execution from the initial state. In this case Vyrd continues with the next entry ($\mathbf{l}_{i+2}$) in the log after resetting its internal state.

In order to keep memory requirements low, we chose to restart Vyrd when jump entries are encountered instead of caching Vyrd's internal state for each node of the execution tree visited. While this possibly results in increased run time, since the test cases we focus on are small and since Vyrd keeps a lot of history information required to check refinement, we believe this is a reasonable trade-off.

5.5 Replaying Actions from The Log with ExplicitMC

Vyrd uses the VM implementation of *ExplicitMC* as the runtime environment to facilitate replaying of the implementation instance $\mathcal{C}_{replay}$ and the specification instance $\mathcal{S}$. A separate VM instance, denoted by VMMC^I, is created for $\mathcal{C}_{replay}$. Since testing and replaying are performed in the same type of VM, VMMC^I can directly apply each record of action (a byte-code instruction or method) on $\mathcal{C}_{replay}$ in a straightforward manner.

For driving the sequential specification, there are two alternatives:

- $\mathcal{S}$ is run in a separate instance of VM, denoted by VM^S. In this case, the

control mechanisms included in VM^S for the state space exploration are disabled since the method is run sequentially without any branching.

- For model checkers such as Java PathFinder implemented in Java-like languages, the VM that the model checker runs the component being verified is itself is run on an underlying native virtual machine or runtime environment. In this case it is less costly in runtime to run $\mathcal{S}$ on this lower level virtual machine.

For Vyrd to be able to check refinement, VMMC^I and VM^S must provide mechanisms for tasks such as replaying logged actions on objects and computing user-defined abstraction functions on $\mathcal{C}_{replay}$ and $\mathcal{S}$. JPF's virtual machine provides this mechanisms. If other virtual machines are used, they must be

5.6 Replaying Actions from The Log with StatelessMC

In *StatelessMC* $\mathcal{C}_{replay}$, $\mathcal{S}$ and the other related objects are created and managed in the same address space as the one Vyrd is run on. $\mathcal{C}$ and $\mathcal{S}$ while replaying and computing the abstract states of them during verification.

StatelessMC requires separate implementations of operations that enable replay. Actions corresponding to updates to fields of shared objects Vyrd can be replayed in a straightforward manner by performing the same update in $\mathcal{C}_{replay}$. If the action replayed is a method call to an assumed-atomic object, Vyrd replays the action simply by calling the method with the same arguments on the corresponding object.

In a similar way to $\mathcal{C}_{replay}$, an instance of $\mathcal{S}$ is kept and manipulated by Vyrd in order to produce the expected atomic behavior. Vyrd manipulates and observes $\mathcal{S}$ by executing the methods of the specification sequentially at the points where the refinement check is performed.

6 Optimizations

While VyrdMC uses model checkers to explore a very restricted state space, the size of this space, related to the number of distinct thread interleavings for two concurrent method executions, can still be exponential in the program size. In this section, we introduce several ideas that reduce the computational cost of VyrdMC to alleviate this problem.

6.1 Exploiting Purity

The idea of "pure" code blocks was introduced by Flanagan and Qadeer in [1]. Intuitively, these are blocks of code that are executed atomically and that do

not result in any net modification of a shared variable when they terminate normally. The concurrent data structures we worked on in [6] make extensive use of code blocks that fit this description. For VyrdMC to be computationally viable, thread interleavings that differ only in their scheduling of pure code blocks need to be considered equivalent. In other words, the model checker must consider blocks marked as pure to be independent from any other code block while applying partial order reduction techniques.

To see how exploiting purity affects the performance of model checker, we had an experience in which two `InsertPair` were executed concurrently on "multiset" [6] instances of size 100, 1000 and 10000. The test programs were driven by Java PathFinder and the executions caused by interleaving of pure blocks that looks for empty slots in the multiset for insertion were eliminated during the state search. Consequently the number of states and the transitions did not change as the size of the multiset got bigger. However, the memory used by the model checker to represent multiset states and the computation time to manage the on-memory representation of the state space increased as size of the the multiset increased. Regardless of the increase in the runtime of the model checker, the verification time were no more than a few minutes, which is still reasonable to check scenarios with several concurrent methods.

6.2 One-pass Assume-Guarantee Reasoning on a Layered Architecture

This section describes a variant of assume-guarantee reasoning that can be applied to systems with a layered architecture in order to reduce the computational complexity of runtime refinement verification. Consider a system consisting of components $\mathcal{C}_1$, $\mathcal{C}_2$,...,$\mathcal{C}_n$ where $\mathcal{C}_1$ is the highest-level layer and $\mathcal{C}_n$ is the lowest. We say that the system has a layered architecture if the following two conditions hold:

- Any layer $\mathcal{C}_i$ has no dependency on higher levels $\mathcal{C}_1$,...,$\mathcal{C}_{i-1}$,
- $\mathcal{C}_i$ only makes calls to methods of $\mathcal{C}_{i+1}$

We perform assume-guarantee reasoning for such systems as follows. While checking refinement for $\mathcal{C}_i$, return values of calls to methods of $\mathcal{C}_{i+1}$ are determined by executing $\mathcal{S}_{i+1}$, the sequential specification for $\mathcal{C}_{i+1}$. This amounts to assuming that, for the execution under consideration, the behavior of $\mathcal{C}_{i+1}$ conforms to $\mathcal{S}_{i+1}$, i.e., does not result in a refinement violation. This assumption is checked while refinement verification is performed for $\mathcal{S}_{i+1}$ for the same execution. Currently, we use an atomized version of $\mathcal{C}_i$ for $\mathcal{S}_i$. Since $\mathcal{S}_i$ executes methods atomically in a linear order, its calls to the next layer's methods are naturally executed atomically as well. Equivalently, in $\mathcal{S}_i$, $\mathcal{C}_i$'s calls to methods of $\mathcal{C}_{i+1}$ are replaced by calls to $\mathcal{S}_{i+1}$, the atomized version of

$\mathcal{C}_{i+1}$.

At runtime, while the actions of the components are being logged, we want to run the system as is and avoid running atomized versions of layers along with the system. We use the following approach to perform refinement checking for all layers in one pass. While checking refinement, for each layer $\mathcal{C}_i$, Vyrd drives an instance $\mathcal{S}_i$ of its atomized version to serve as the specification. Equivalence of the traces of $\mathcal{C}_i$ and $\mathcal{S}_i$ is checked each time a method commits in $\mathcal{C}_i$.

As an example, consider the BLinkTree module: It uses Cache as a lower layer component to store its data and Cache uses Chunk Manager for persistent storage. While checking refinement, Vyrd drives one implementation and one specification instance for each of BLinkTree, Cache and Chunk Manager for replaying purposes. Vyrd applies the conformance check as usual when any method execution of BLinkTree, Cache or Chunk Manager commits using appropriate implementation and specification instances.

7 Conclusions

We introduced a runtime refinement checking framework, VyrdMC, that combines a very simple test harness, a model checker, and our refinement checking tool Vyrd. The model checker improves the coverage of testing by controlling thread interleavings to generate quantitatively distinct execution traces. Vyrd analyzes resulting execution traces for I/O refinement and view refinement and provides improved observability and more thorough verification.

The paper discussed several issues around two types of model checkers, an explicit state and a stateless model checker, as part of the proposed framework. Among them are how Vyrd is kept synchronized with the state space exploration process, and how the underlying runtime environment of the model checker is exploited by Vyrd. We proposed optimizations that reduce the computational complexity of VyrdMC.

We are implementing our framework on the execution-based JPF and Verisoft model checkers. We believe that the integrated framework will be an easy-to-use and powerful tool for the discovery of interesting refinement errors in concurrent software

References

[1] C. Flanagan, S. Freund, and S. Qadeer. Exploiting Purity for Atomicity. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*. ACM Press, 2004.

[2] C. Flanagan. Verifying Commit-Atomicity using Model-Checking In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software*, April 2004. Volume 2989 of Lecture Notes in Computer Science. Springer-Verlag.

[3] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004), San Francisco, CA, USA, December 2004, pages 105-120.*
http://research.microsoft.com/research/sv/Boxwood

[4] Y. Sagiv. Concurrent Operations on B-trees with Overtaking. *Journal of Computer and System Sciences*, 3(2), Oct. 1986.

[5] S. Tasiran and S. Qadeer. Runtime Refinement Verification of Concurrent Data Structures. In *Proc. Runtime Verification '04 (ETAPS '04). Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[6] Tayfun Elmas, Serdar Tasiran, Shaz Qadeer. VYRD: VerifYing Concurrent Programs by Runtime Refinement-Violation Detection. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI), Chicago, Illinois, USA, June 12-15, 2005.

[7] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174–186, Paris, January 1997.

[8] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model Checking Programs. In IEEE International Conference on Automated Software Engineering (ASE), September 2000.

[9] P.Godefroid. Partial Order methods for the Verification of Concurrent Systems-An approach to the State-Explosion Problem. volume 1032 of Lecture Notes in Computer Science. Springer-Verlag, January 1996.

[10] Cormac Flanagan, Patrice Godefroid. Dynamic Partial-order Reduction for Model Checking Software. POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Long Beach, California, USA. pages 110–121, 2005.

[11] Eitan Farchi, Yarden Nir, Shmuel Ur. Concurrent Bug Patterns and How to Test Them. 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, page 286.

[12] Alex Groce and Willem Visser. Heuristics for Model Checking Java Programs. International Journal on Software Tools for Technology Transfer, Volume 6, Number 4, pages 260–276, August 2004.