VYRD: VerifYing Concurrent Programs by Runtime Refinement-Violation Detection

Tayfun Elmas Serdar Tasiran
Koç University, Istanbul, Turkey
{telmas,stasiran}@ku.edu.tr

Shaz Qadeer
Microsoft Research, Redmond, WA
qadeer@microsoft.com

Abstract

We present a runtime technique for checking that a concurrently-accessed data structure implementation, such as a file system or the storage management module of a database, conforms to an executable specification that contains an atomic method per data structure operation. The specification can be provided separately or a non-concurrent, "atomized" interpretation of the implementation can serve as the specification. The technique consists of two phases. In the first phase, the implementation is instrumented in order to record information into a log during execution. In the second, a separate verification thread uses the logged information to drive an instance of the specification and to check whether the logged execution conforms to it. We paid special attention to the general applicability and scalability of the techniques and to minimizing their concurrency and performance impact. The result is a lightweight verification method that provides a significant improvement over testing for concurrent programs.

We formalize conformance to a specification using the notion of refinement: Each trace of the implementation must be equivalent to some trace of the specification. Among the novel features of our work are two variations on the definition of refinement appropriate for runtime checking: I/O and "view" refinement. These definitions were motivated by our experience with two industrial-scale concurrent data structure implementations: the Boxwood project, a B-link tree data structure built on a novel storage infrastructure [10] and the Scan file system [9]. I/O and view refinement checking were implemented as a verification tool named VYRD (VerifYing concurrent programs by Runtime Refinement-violation Detection). VYRD was applied to the verification of Boxwood, Java class libraries, and, previously, to the Scan filesystem. It was able to detect previously unnoticed subtle concurrency bugs in Boxwood and the Scan file system, and the known bugs in the Java class libraries and manually constructed examples. Experimental results indicate that our techniques have modest computational cost.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — formal methods, validation; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — mechanical verification, specification techniques; D.2.5 [*Software Engineering*]: Testing and Debugging — debugging aids, diagnostics, monitors, tracing

General Terms Algorithms, Verification

Keywords Runtime Verification, Refinement, Concurrent Data Structures,

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

1. Introduction

Many widely-used software systems, such as file systems, databases, internet services, and standard Java and C# class libraries, have concurrently-accessed data structures at their core [9, 10]. Performance requirements force these systems to use intricate synchronization mechanisms, which makes them prone to concurrency errors. Functional errors in these systems may have serious consequences such as data loss, corruption, or system crash. Therefore, functional correctness is just as important as performance. Concurrency bugs are notoriously difficult to detect and reproduce through testing. This paper introduces a new scalable runtime analysis technique called *refinement checking* for finding concurrency-related errors in industrial-scale software implementations.

Checking refinement consists of verifying that each execution trace of a concurrent implementation is "equivalent" to a trace of its specification. This criterion provides more thorough validation than checking method-local assertions during program execution. We require that specifications execute each method atomically. Thus, if an implementation refines a specification, each implementation trace, in which portions of method executions are interleaved with others, is equivalent to a *specification trace* with atomic method executions. Atomicity, a more widely-studied correctness criterion [7, 6] requires that each implementation trace to be equivalent to some atomic execution *of the implementation*. The distinction can be important for concurrent systems and makes atomicity unnecessarily restrictive in some cases. Consider, for example, an implementation in which a method may terminate exceptionally because resource contention between concurrent threads prevents it from completing its job. In an atomic execution of this implementation, there is only one thread in the middle of a method execution at any given time and thus no resource contention between methods, therefore, method executions never terminate exceptionally. Therefore, executions of this implementation containing exceptional terminations of this method are not equivalent to any atomic execution of the system and will be declared erroneous according to the atomicity criterion. As this example shows, since there is not a (possibly more permissive) specification separate from the implementation, atomicity is sometimes too stringent and may declare certain acceptable system behavior as erroneous. If a specification that allows exceptional method terminations had been used for refinement checking, this execution would not have caused a refinement violation. The ability to use specifications that more closely

model concurrent executions makes refinement a more appropriate correctness criterion for concurrent systems.

In this work, we present a technique for checking refinement at runtime. The specification can be provided separately or a non-concurrent, "atomized" interpretation of the implementation can serve as the specification. We have chosen to investigate runtime checking and sacrifice completeness because of the computational cost and practical difficulty of exhaustively verifying refinement for concurrent software, which requires reasoning about the entire state space of the implementation. The runtime refinement checking techniques presented in this paper are very scalable, require relatively little manual effort, and provide significantly more thorough checking than testing.

A key contribution of this paper is the definition and use of two variations on the notion of refinement appropriate for runtime checking. These definitions, called I/O and "view" refinement, are motivated by our experience with verifying two industrial-scale concurrent programs: Boxwood, a B-link tree data structure built on a novel storage infrastructure [10] and the Scan file system [9, 13]. Both of these systems aim to provide the illusion of atomic, linearly ordered method executions, but are highly concurrent implementations and use intricate synchronization mechanisms and caching to improve performance. Interestingly, neither program's methods can be shown to be atomic using simpler correctness criteria and analysis methods such as reduction [7], commit atomicity [4], or purity [5]. We provide a more thorough discussion of the related work and a comparison of refinement with other correctness criteria in Section 8.

Runtime checking of our simpler correctness criterion, I/O refinement, requires very little instrumentation and logging while still providing a more rigorous check than pure testing. Intuitively, I/O refinement stipulates that for each execution of the implementation, there is an atomic, sequential run (the "witness interleaving") of the specification consisting of the same method calls (and arguments) and return actions (and values). This correctness criterion can be viewed as a variant of linearizability [8] as elaborated on in Section 8.

To enable runtime checking of I/O refinement, the programmer annotates the implementation code so that in every method execution, a unique action is marked as the "commit action". Our analysis uses the order of occurrence of commit actions during execution to generate the witness interleaving. This novel use of commit actions distinguishes I/O refinement from testing. Given the witness interleaving, we can determine exactly which return values are allowed for a particular invocation of a method. In the absence of this information, pure testing is forced to be overly permissive.

Our second correctness criterion, "view" refinement, provides more thorough checking than I/O refinement by means of more visibility into program state. View refinement augments I/O refinement by requiring a particular correspondence between the implementation and specification states when a commit action is taken. Hypothetical view variables $\texttt{view}^I$ and $\texttt{view}^S$ are added to the implementation and specification, respectively. Intuitively, the value of the view variable is a canonical representation of the abstract data structure state when the commit action is taken. View refinement requires that the value of $\texttt{view}^I$ matches $\texttt{view}^S$ for each method execution. The programmer specifies how $\texttt{view}^I$ and $\texttt{view}^S$ are to be computed as a function of the implementation and specification state, respectively.

The visibility `view` provides into program state makes possible the early detection of discrepancies of the implementation state from the specification state. In contrast, in testing and I/O refinement, an implementation error is detected only if and when it causes a discrepancy in the return value of a method. In a particular run, a triggered error may lead to an observed discrepancy long after it occurs, or not at all, whereas, in a different run, the same error would have caused a more serious and easily observable outcome. `view` enables the detection of the error even in the former case. This strength of view refinement comes at the cost of more detailed instrumentation, logging, and programmer effort put into specifying how $\texttt{view}^S$ and $\texttt{view}^I$ should be computed.

```
Insert(x, returnValue)
1 if (status == success)
2   M = M U {x}
3 return returnValue;

LookUp(x)
1 return x in M
```

Figure 1. Specifications of multiset operations

We implemented I/O and view refinement checking as a verification tool called **VYRD** (**V**erif**Y**ing concurrent programs by **R**untime **R**efinement-violation **D**etection). VYRD can perform refinement checking of industrial-scale concurrent data structure implementations. We used the Boxwood project to drive our development of the VYRD tool. VYRD is very effective in detecting concurrency errors. Using it, we were able to detect a subtle error in a cache module in Boxwood that had previously gone undetected[1]. VYRD was also able to catch known concurrency bugs in `java.util.StringBuffer` and `java.util.Vector`. An earlier prototype of VYRD caught several subtle concurrency errors in a Windows NT filesystem [9]. In addition to extensive experimental results about the effectiveness and computational cost of using VYRD, we report the issues identified and solutions implemented while verifying industrial-scale systems. Chief among these are minimizing performance and concurrency impact on the program being verified and incremental computation and comparison of $\texttt{view}^I$ in order to avoid re-traversing the entire program state at each verification step.

The organization of the paper is as follows. We illustrate our runtime verification technique on a concurrent implementation of a multiset described in Section 2. Section 3 formalizes state transition systems and our notion of refinement. Sections 4 and 5 present I/O and view refinement and our technique for checking them at runtime. The refinement checking tool VYRD is described in Section 6. Experimental results from the application of VYRD to Boxwood and other programs are described in Section 7. We discuss related work in Section 8.

2. Example

We will use a multiset data structure as our running example throughout the paper. The first, simple version of multiset supports two operations: `Insert(x)` to insert an element `x` into the multiset, and `LookUp(x)` to check if `x` is an element of the multiset. The specifications of these operations are presented in Fig. 1 where `M` is a state variable that represents the multiset contents. Any invocation of the `Insert` operation is allowed to terminate successfully or exceptionally, but exceptionally-terminating `Insert` operations are required to leave the multiset state unchanged. The multiset implementation (Fig. 2) uses an array `A[0..n-1]` to store the multiset elements. The field `A[i].elt` denotes the element stored in `A[i]`, and initially $\texttt{A}[\texttt{i}].\texttt{elt} = null$ for all `i`. The `FindSlot` subroutine looks for an available slot in the array for a single element `x`. If it finds one, it reserves the slot by setting its content to `x` and returns its index. Otherwise it returns -1.

[1] For reasons that are not related to this error, the cache module had been re-written in a later version of Boxwood. The new cache module does not contain this error.

```
FindSlot(x)
1 for (i=0; i<n; i++)
2   synchronized(A[i]) {
3     if (A[i].elt == null) {
4       A[i].elt = x;
5       return i;
6     }
7   }
8   return -1;
```

```
Insert(x)
1 i = FindSlot(x);
2 if (i == -1)
3   return failure;
4 return success;
```

```
Delete(x)
1 for (i=0; i<n; i++)
2   synchronized(A[i]) {
3     if (A[i].elt = x) {
4       A[i].elt = null;
5       return;
6     }
7   }
```

```
LookUp(x)
1 for (i=0; i<n; i++)
2   synchronized(A[i]) {
3     if (A[i].elt = x)
4       return true;
5   }
6 return false;
```

Figure 2. Implementation of the multiset operations

The idea of I/O refinement is illustrated in Fig. 3. The implementation's actions are shown on the left half of the figure from top to bottom in the order they happen in time. The colors of the boxes representing actions indicate different threads performing them. The execution shows four method calls `LookUp(3)`, `Insert(3)`, `Insert(4)` and `Delete(3)` being concurrently executed by four different threads. Since the four method executions overlap with each other, they could be serialized in any one of 4! ways. A simple but naive method for determining the correctness of the return value of `LookUp(3)` would require evaluating 4! serializations. Clearly, this method would not scale as the number of methods being executed concurrently increases.

Our solution to this problem has two parts. First, we require that the programmer specify a unique *commit action* for each method execution. The concurrent method executions are then deemed to have been serialized in the order of occurrence of the commit actions. Second, we use the sequence of commit actions in an execution of the implementation to drive the execution of the specification. Figure 3 also shows the state transitions of the specification in the right half. At each commit action, the method that is committing and its return value (derived by looking ahead in the implementation's execution) are used to execute the specification. From its current state the specification must take a transition associated with the committing method and its return value. If this is not possible, e.g., if a call to `LookUp(y)` method has a return value of "false" when the specification state indicates that `y` is in the multiset, a refinement-violation is detected. Observe that, in the example in Fig. 3, although the execution of `LookUp(3)` by the gray thread starts before the execution of `Insert(3)` and ends before the execution of `Insert(3)` ends, `LookUp(3)` returns "`true`" since its commit action comes after that of `Insert(3)`.

To see how the notion of I/O refinement improves upon existing testing techniques for concurrent software, consider a verification thread consisting of calls to `LookUp` run after the termination of the threads in Fig. 3. Since the executions of `Insert(3)` and `Delete(3)` overlap, the verification thread would have to consider both possible return values of `LookUp(3)` to be correct. Performing I/O refinement using the order of commit points to obtain a witness serialization, however, we are able to determine that `Delete(3)` occurs after `Insert(3)` and a `LookUp(3)` that occurs after the methods in Fig. 3 should return `false`.

2.1 Inserting a pair

We now present a more complicated version of the multiset example that supports a new operation `InsertPair(x,y)`, whose specification is given below. Here, `M` denotes the multiset contents.

```
InsertPair(x, y, returnValue)
1 if (returnValue == success)
2   M = M ∪ {x,y}
3 return returnValue;
```

When `InsertPair(x,y)` terminates successfully, it adds `x` and `y` to the multiset. If it terminates exceptionally, the multiset state

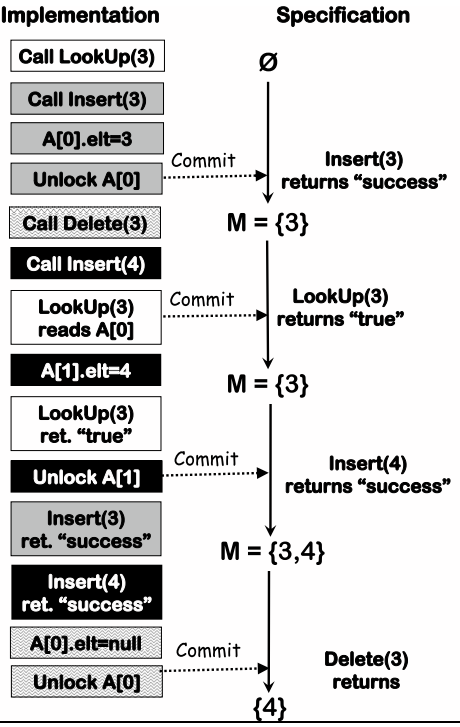

Figure 3. Refinement of multiset. Some actions are not shown to keep the figure simple.

should remain unchanged. In particular, it will be considered a refinement violation if only one of `x` or `y` is inserted into the multiset.

We present this new multiset implementation for two reasons. First, a variety of concurrent data structures, such as file systems and storage systems, implement operations that require multiple resources for successful completion. This feature adds a significant amount of complexity to the implementation. The `InsertPair` operation mimics such operations and is therefore particularly suited for illustrating the power of our approach. Second, this example shows the generality of our approach compared to existing techniques based on reduction (e.g., [7, 6, 5]) which cannot prove the correctness of the implementation of `InsertPair`.

The implementation of `InsertPair` is given in Figure 4. The implementation adds a boolean field `A[i].valid` to each array element to indicate whether the element `A[i]` should be considered a member of the multiset. `InsertPair` first makes sure that there is room for `x` and `y`. If this is not the case, it terminates exceptionally, and frees space it may have allocated. Otherwise, the code block in lines 9-14 atomically adds `x` and `y` to the multiset by setting their `valid` bits. Line 3 of the `LookUp` and `Delete` methods in Fig. 2 must now be modified to read

```
3'    if ((A[i].elt == x) && (A[i].valid))
```

```
InsertPair(x,y)
1  i = FindSlot(x);
2  if (i == -1)
3    return failure;
4  j = FindSlot(y);
5  if (j == -1) {
6    A[i].elt = null;
7    return failure;
8  }
9  synchronized (A[i]) { // begin commit block
10   synchronized (A[j]) {
11     A[i].valid = true;
12     A[j].valid = true;
13   }                    // end commit block
14 }
15 return success;
```

Figure 4. The implementation of `InsertPair`.

The commit point in an execution of the `InsertPair` method that returns *success* is selected to be line 13. The intuition behind this is as follows. Suppose an application thread t is in the process of executing an invocation of `InsertPair` that will succeed. Let `p` and `q` be the locations where this invocation inserts `x` and `y` respectively. Another thread t' can access `A[p]` and `A[q]` either (1) before thread t executes line 10, or (2) after thread t executes line 13. Consider the first case. Thread t holds the locks for `A[p]` and `A[q]` while it sets `A[p].valid` and `A[q].valid` to *true*. Therefore, if thread t' reads the valid bits before thread t executes line 10, it reads them as *false*. Consequently, thread t' observes `x` and `y` as not having been inserted into the multiset. In the second case, since `A[p].elt = x`, `A[q].elt = y`, and `A[p].valid` and `A[q].valid` are *true*, thread t' observes both `x` and `y` as having been inserted into the multiset. Thus, line 13 is where the modified view of the data structure becomes visible to other threads.

To demonstrate refinement checking, a buggy implementation of `FindSlot` is given in Fig. 5. It is possible for two concurrently executing `FindSlot` operations to both conclude at line 2 that the same index `i` is available, since neither have to hold the lock for `A[i]` in line 2. Fig. 6 shows an I/O refinement violation that is caused by this bug. Thread T2 overwrites the value 5 that thread T1 inserted into `A[0]`. If the test program included a `LookUp(5)` after both `InsertPair` operations complete, the specification state would be {5,6,7,8} and require that the return value be `true` while, in the implementation, the return value would be `false`.

```
FindSlot(x)
1 for (i=0; i<n; i++) {
2   if (A[i].elt == null) { // A[i] should be locked
3     synchronized(A[i]) {
4       A[i].elt = x;
5       return i;
6     }
7   }
8   return -1;
```

Figure 5. Buggy implementation of `FindSlot`.

3. Definitions

In this section, we define the notion of I/O refinement and view refinement. This section is not a prerequisite for an intuitive understanding of the remainder of the paper and it can be skipped on a first reading.

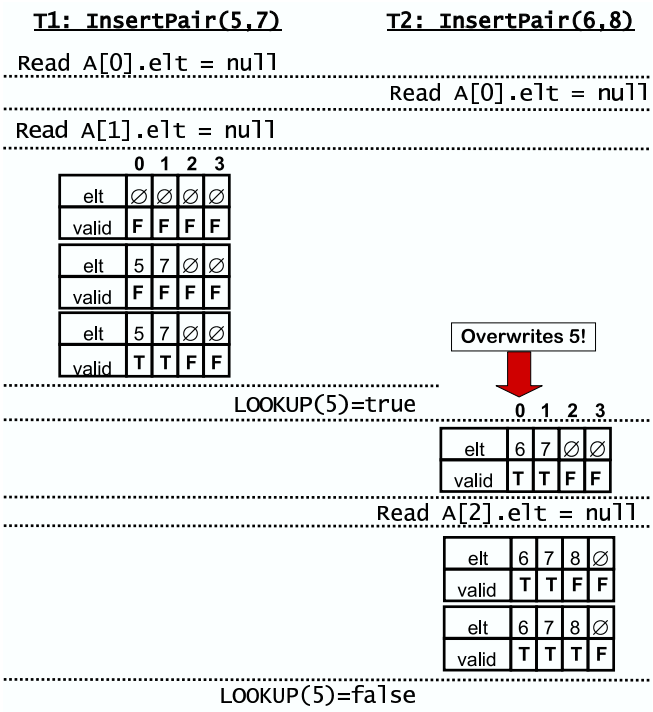

Figure 6. Refinement violations in the buggy version of multiset.

We focus on concurrently accessible implementations of data structures written in object-oriented languages. The data structure makes available a set of operations each of which is implemented as a *public method*. $\mathcal{M}$ denotes the set of public methods. Methods return a single value, and exceptional terminations for methods are modeled by special return values. A method $\mu \in \mathcal{M}$ is called an *observer* if μ's specification does not allow it to modify data structure state. All other methods are called *mutators*. The domain Tid represents the set of thread identifiers and is the union of two disjoint sets, Tid_{app} and Tid_{ds}. Tid_{app} contains identifiers of application threads that call the public methods and Tid_{ds} contains identifiers of worker threads that perform tasks internal to the data structure.

3.1 State transition systems

We use *state transition systems* as the formal semantics of both specification and implementation programs. A state transition system is a tuple $(\mathcal{V}, S, s_0, \delta)$:

- $\mathcal{V}$ is the set of *program variables*.
- S is the set of *states*. Each *state* is an assignment of a value of the correct type to each variable in $\mathcal{V}$. $s_0 \in S$ is the *initial state*.
- δ is the *transition function* from $S \times Actions$ to S, where $Actions$ is the set of *actions* that the system can perform. Each distinct method call, return, and atomic update of a set of state variables is modeled by a unique action. If $\delta(s, \alpha) = s'$, the transition system may perform the action α in state s to change the state to s'. We denote such a transition by $s \xrightarrow{\alpha} s'$.

A *run* of the state transition system is a finite sequence $r = s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} s_n$ for some $n \geq 0$ such that $s_i \xrightarrow{\alpha_i} s_{i+1}$ for all $0 \leq i < n$.

A *call action* is a tuple $\alpha^{call} = (t, \mu, \chi)$, where t is the identifier of the thread performing the method call, $\mu \in \mathcal{M}$ is the public method invoked, and χ is the list of actual method arguments. A

return action is a tuple $\alpha^{ret} = (t, \mu, \rho)$, where t is the identifier of the thread performing the method return, $\mu \in \mathcal{M}$ is the returning public method, and ρ is the value returned by the method. A state in a run is *quiescent* if it does not lie between the call and return action of any method.

3.2 Traces and well-formedness

Traces are defined by designating a subset $\Lambda \subseteq Actions$ of actions as *visible*. Call and return actions of public methods are required to be visible. Given a run r of the state transition system, the Λ-*trace* corresponding to r is the sequence of visible actions, i.e., actions belonging to Λ that take place during that run. When it is clear from the context, we simply refer to a trace and omit mention of Λ.

The sequence of actions associated with a thread t and lying between the call and return action for a public method is called *an execution* ξ of that method. Formally, $\xi = \langle \mu, t, \sigma \rangle$ where μ is the method executed by thread $t \in Tid$ and σ is the sequence of the actions that belong to the particular invocation. ξ is said to have a signature $Sign(\xi) = \langle t, \mu, \chi, \rho \rangle$, where $t \in Tid$ is the thread that executed method $\mu \in \mathcal{M}$, χ is the set of actual parameters and ρ is the return value for this execution. For two method executions ξ and ξ' in a trace τ, we say $\xi \lhd_\tau \xi'$ if the return action of ξ comes before the call action of ξ' in τ.

A sequence of actions by an application thread t is *well-formed* if (i) each call action α_p^{call} is eventually followed by a matching return action α_p^{ret}, and (ii) if α_q^{call} takes place after α_p^{call} but before α_p^{ret}, then α_q^{ret} takes place before α_p^{ret} as well. A trace τ is *well-formed* if for every application thread t, the subsequence of τ corresponding to actions of thread t, denoted by $\tau|_t$, is well-formed. A run is *well-formed* if its corresponding trace is well-formed. In this paper, we restrict our analysis to well-formed runs of state transition systems. A well-formed run is *method-atomic* if after every call action $\alpha^{call}(t, \mu, \chi)$ to a public method μ by an application thread t, no thread other than thread t performs an action until the corresponding return action $\alpha^{ret}(t, \mu, \rho)$ has occurred. A state transition system is *method-atomic* if all of its runs are method-atomic. Each portion of such a run between (and including) corresponding call and return actions is called an *atomic fragment* of the run. A method-atomic state transition system is *deterministic* if whenever two atomic fragments of any two runs have the same starting state and the same method execution signature, they have the same final state as well. Thus, every trace of a method-atomic and deterministic state transition system is produced by a unique run. Note that this definition of determinism allows different possible return values for a method invocation at a given state. It only requires that given the return value, the final state be uniquely defined. We require our specifications to be method-atomic and deterministic state transition systems.

3.3 Refinement between state transition systems

A refinement relation $\preceq_\Lambda$ between state transition systems I and S designates a subset Λ of actions common to I and S to be *visible*. Let the implementation and specification of a concurrent data structure be given by the state transition systems I and S, respectively. I Λ-*refines* S, denoted $I \preceq_\Lambda S$, if for every Λ trace τ of I, there is a Λ trace τ' of S such that (i) $\tau|_t = \tau'|_t$ for all $t \in Tid_{app}$, and (ii) for all method executions ξ and ξ', $\xi \lhd_\tau \xi'$ implies $\xi \lhd_{\tau'} \xi'$. In our simplest notion of refinement, *I/O-refinement*, call and return actions are the only visible actions.

4. Checking I/O refinement at runtime

This section gives more details of our method for checking I/O refinement at runtime. Suppose I has a run r that has been annotated with information about the commit action for each method executed by each thread. Recall that only call and return actions are visible for I/O refinement. Therefore, the corresponding trace τ has the property that for each thread t, $\tau|_t$ is a sequence of pairs of matching call and return actions. We construct a method-atomic interleaving by arranging these call-return pairs in the order of their commit actions. Finally, we check whether the resulting interleaving is a trace of S. Recall that, since S is required to be atomic and deterministic, this ordered sequence of call-return pairs corresponds to a unique run, i.e., uniquely determines the sequence of states that the specification goes through. Therefore, it is straightforward to check whether a given method-atomic interleaving corresponds to a trace of S. We simply execute the specification one method call at a time in the order given by the interleaving. For each method execution ξ, the signature $Sign(\xi)$ (which includes the return value) derived from τ is used to drive the specification. If at any point, it is not possible to execute the specification while conforming to the return action as specified in $Sign(\xi)$, the refinement check is said to fail. We now elaborate on the details of our checking method.

4.1 Selecting commit actions

The order of the commit actions in time must coincide with the application's view of how the state of the data structure transforms over time. Intuitively, the commit action in an execution of a mutator method by a thread is the first action which makes visible to other threads the modified abstract data structure state. An example was provided in Section 2.1. Selection of commit actions for a method from the Boxwood project is given in Section 7.2.5.

In practice, commit actions are specified by designating certain lines in the implementation code to be *commit points*. The programmer must make sure that for each method, exactly one action is marked as the commit action for every execution path through the method code. This is accomplished by associating with each commit point annotation a condition under which it is the commit point. This condition can be typically expressed in terms of thread-local variables.

Commit actions are really hints to refinement checking tools and there is no formal procedure for selecting them. If a certain design pattern is followed, a method for selecting them can be devised. If the user has defined an abstraction function (a "view", as explained in Section 5), a good guess for the commit action is the first lock release following the last modification to a variable in the support of "view". The examples of commit point selection referred to in Sections 2.1 and 7.2.5 above fit this description.

The runtime refinement check described could fail either because the implementation truly does not refine the implementation or because the witness interleaving obtained using the commit actions is wrong. Comparing the witness interleaving with the implementation trace reveals which one of these is the case. If the witness interleaving is wrong, the programmer must re-examine and modify the commit point selection or must refine the conditions associated with the commit point. We have found this iterative process very useful for debugging code that is in development and for improving our understanding of method implementations.

4.2 Off-line refinement checking using a log

It is desirable for a runtime verification tool not to modify the concurrency characteristics of the implementation significantly. This could happen if a large amount of instrumentation overhead is introduced or application threads block waiting for access to the instrumentation module. To interfere minimally with the implementation, we run refinement checking on a separate thread which is informed about the implementation's actions through a *log*. Formally, a log $\mathcal{L}$ is a finite sequence of actions $\mathcal{L} = \alpha_0, ..., \alpha_n$. In practice, the log is a file whose tail is kept in memory for faster

Actions in the log	...	α_0^{cmt}	...	α^{call}	...	α_1^{cmt}	...	α_2^{cmt}	...	α_{n-1}^{cmt}	...	α_n^{cmt}	...	α^{ret}
Specification state		s_0				s_1		s_2		s_{n-1}		s_n		

Figure 7. Checking refinement for observer methods. α^{ret} must be consistent with s_i for some $0 \leq i \leq n$.

access. The implementation threads write entries to the log as they run; the verification thread reads these entries and performs refinement checking. For I/O refinement checking, the entries of the log are required to contain all call, return, and commit actions. Actions must appear in the log in the order they occur in the execution. One way to achieve this is to require that each logged action be performed atomically with the corresponding log update.

4.3 Verifying observer methods

While the commit action of a mutator method is typically associated with an atomic write to a shared variable, the commit action of an observer method is typically associated with a shared variable read. A subset of shared variable reads performed by an observer method determine its return value, as is the case, for example, with a `LookUp` call that returns `true`. However, it is not known before the read is performed whether its outcome will determine the return value. Therefore, precisely marking the commit action of an observer method requires that almost all reads performed by it be logged. This would result in large overhead and significant impact on the concurrency among the operations. Instead, we obviate the need for annotating and logging the execution of observer methods as follows. We only record the call and return actions of observer methods into the log. For one such execution ξ of an observer method μ, let α^{call} and α^{ret} be the call and return action entries in the log, and $\alpha_1^{cmt}, \alpha_2^{cmt}, \alpha_3^{cmt}, ..., \alpha_n^{cmt}$ the commit actions of mutator methods that lie between α^{call} and α^{ret} in the log. Let α_0^{cmt} be the last commit action before α^{call}. We denote by s_i the specification state obtained after atomically executing the method associated with α_i^{cmt} for all $0 \leq i \leq n$. If the return value ρ in ξ of the method μ is consistent with what the specification allows as a valid return value for μ at any of the states $s_0, s_1, ..., s_n$, we declare ξ correct. This amounts to allowing the real commit action of ξ to be anywhere between (and including) α^{call} and α^{ret}. Otherwise, a refinement violation is signaled since a return value of ρ is not consistent with any choice of commit action for ξ. Observe that this check does not result in a combinatorial blow-up in the number of linearizations considered, since the witness interleaving provides a unique ordering of commit points of mutator methods. Because observer methods do not modify the data structure state, the choice of the commit action for an observer method does not influence the checks performed for other methods.

4.4 Using the "atomized" implementation as the specification

If a separate specification does not exist, our technique enables the use of an "atomized version" of the same implementation code as the specification. In an atomized version, the program is forced to have only method-atomic executions. This is accomplished by defining a global lock and requiring that methods must hold this lock during execution and must release it upon termination. The atomized version's methods are also modified so that, in addition to the original set of arguments, they take the return value as an argument and compute the new specification state, in a fashion similar to Figure 1. The idea of using an atomized version as a specification for refinement verification was introduced earlier by us [14] and Flanagan [4] independently. An alternative way to view this verification approach is to decompose the refinement verification task into two: Verifying that the implementation refines the atomized version, and that the atomized version, a sequential program, refines a higher-level specification. The latter problem can then be addressed using methods developed for that purpose.

5. Improving I/O refinement using view variables

As an extreme example of the limited visibility provided by testing and I/O refinement, consider test programs for the multiset example that only call `InsertPair`. Since no observer methods are called, the runtime checks for I/O-refinement would pass trivially for all such tests. For useful testing or I/O refinement checking, frequent calls to observer methods must be performed. But this might reduce the degree to which mutator methods are executed concurrently and make it less likely that an existing error will get triggered. Furthermore, calls to the observer methods might not get scheduled at the most interesting points. This section presents "view" refinement, a correctness condition for concurrent programs which augments I/O-refinement with more visibility into the data structure state. This extra visibility enables us to get more thorough checking from each test run.

The key idea in view refinement is to introduce hypothetical "view" variables $\texttt{view}^I$ and $\texttt{view}^S$ into the implementation and specification, respectively. Intuitively, the values of $\texttt{view}^I$ and $\texttt{view}^S$ extract the contents of data structure from the implementation and specification states. These variables are computed as a function of the data structure state and abstract away information that is not relevant to what applications can observe. For example, for a binary search tree, $\texttt{view}^I$ might be defined as the list of the $(key, value)$ pairs, thus abstracting away the structure of the tree. If a hashtable is given as the specification for the binary tree, $\texttt{view}^S$ might again be the set of $(key, value)$ pairs while the hash function and the collision resolution mechanism are abstracted away. The programmer is asked to specify how the view variables are to be computed. We have found the definition of `view` to be an iterative process which leads to a better understanding of why the program guarantees atomicity of methods.

5.1 Computing `view` using a log

The implementation and specification are not actually modified to perform view refinement checking. Instead, the verification thread separately (possibly off-line) constructs the values of $\texttt{view}^I$ and $\texttt{view}^S$ using the log and the specification run driven by the witness interleaving, respectively. The variables $\texttt{view}^I$ and $\texttt{view}^S$ are initialized to the same value. Each mutator method updates $\texttt{view}^I$ once, atomically with its commit action. $\texttt{view}^S$ is updated once atomically anytime between the call and return of each mutator method. Observer methods do not modify the view variables. During runtime verification, we now also check that the same sequence of updates are performed to $\texttt{view}^I$ and $\texttt{view}^S$. Formally, we declare as visible all commit actions and annotate each of them with the corresponding updated value of $\texttt{view}^I$ or $\texttt{view}^S$.

We now illustrate this method on the multiset example. $\texttt{view}^S$ is selected to be the entire multiset contents `M`. $\texttt{view}^I$ is updated atomically with the commit actions of mutator methods as follows:

```
1 viewI = ∅
2 for (i=0; i<n; i++)
3   if A[i].valid
4     viewI = viewI ∪ {A[i].elt};
```

With the addition of the auxiliary variable $\texttt{view}^I$ to the implementation, we get useful checking even with a test program that has no calls to `LookUp`. This stronger correctness criterion is more likely to expose errors and provide early warnings as the following examples demonstrate. First consider the bug scenario in Fig. 6.

The first call to `LookUp(5)` by thread T1 returns the expected value of `true`. If a second call to `LookUp(5)` had not been performed at the end of the test program by any thread, the error would have gone undetected. This is a likely scenario since T2 is likely to only check the results of its own operations. Computing $\texttt{view}^I$ at the commit point of `InsertPair(6,8)` and comparing it with $\texttt{view}^S$ would have revealed the bug immediately. Suppose that a thread in the test program calls `InsertPair(x,x)` to insert two copies of "x" into the multiset, but, because of an error in the implementation, only one "x" gets inserted into the array `A`. To expose the error through testing or I/O refinement checking, we need an execution that follows this call with `Delete(x)`, followed by `Lookup(x)` with no other calls between them to mutators with "x" as the argument. The probability of this during a test program not written to target this error is low. Even if such a test scenario were exercised, if the insert, delete and lookup operations were far apart in the execution, by the time `Lookup(x)` returns an unexpected result, it would be difficult to locate the original error. View refinement checking detects this error immediately at the commit action of `InsertPair(x,x)`.

5.2 Commit blocks

The computation of $\texttt{view}^I$ given above is a bit oversimplified. Consider the scenario where a thread t_1 that is executing an `InsertPair(`$\texttt{x}_1$`,`$\texttt{y}_1$`)` operation completes line 11 (`A[i].valid = true`). Then, a context switch occurs to thread t_2 which is executing `InsertPair(`$\texttt{x}_2$`,`$\texttt{y}_2$`)`. Suppose that t_2 executes its commit action (line 13). At this point, the $\texttt{view}^I$ computation will include $\texttt{x}_1$ into $\texttt{view}^I$ but not $\texttt{y}_1$, since its valid bit is not yet set. Since lines 11 and 12 of Fig. 4 are protected by locks, it is not possible for another thread to see this "dirty" state where $\texttt{x}_1$ is in the multiset but $\texttt{y}_1$ is not. To handle such cases, intuitively, the computation of $\texttt{view}^I$ needs to "roll back" some shared variable modifications that uncommitted operations have performed. To ease the task of defining and computing $\texttt{view}^I$, we introduce the concept of *commit blocks*. Lines 9 and 13 in Fig. 4 indicate the beginning and the end of the commit block for `InsertPair`. The intent of the commit block is that (i) it can be easily verified to be atomic (by inspection, statically, using a reduction argument, or be checked at runtime using a tool based on reduction, such as Atomizer [6]), and (ii) the task of computing $\texttt{view}^I$ is simplified by assuming that only the committing thread is in the process of executing a commit block. In an actual execution τ, (ii) does not hold, but by using (i), the execution can easily be converted to an equivalent execution $\tilde{\tau}$ where (ii) holds. Then, `view` can be written under the assumption that, at any given commit point, only one thread is in the process of executing its commit block. Conceptually, our refinement checking algorithm first computes $\tilde{\tau}$ from the actual execution τ described in the log, and then performs refinement checking on $\tilde{\tau}$. In fact, relevant portions of $\tilde{\tau}$ are constructed incrementally, on-the-fly during the refinement check.

The choice of the commit block must strike a compromise between simplifying the computation for $\texttt{view}^I$ and verifying that the commit block is atomic. Marking too large a block, for instance, the entire `InsertPair` method as the commit block would make it impossible or too expensive to check that it is atomic. Marking too small a block, for instance, only the commit action, as the commit block does not simplify the computation of $\texttt{view}^I$.

The granularity of checking done using view variables is intermediate between two extremes. At one extreme, one can require a state correspondence only at quiescent states of the implementation as in [13, 4]. But most industrial-scale concurrent data structures are built to be used by large numbers of threads continuously and during any realistic execution, quiescent points are very rare. Checking only at these points might cause errors to be overwritten or to be discovered too late. At the other extreme, asking the programmer write a refinement map that relates implementation and specification states after each action is impractical. Our choice of view variables that are updated only with commit actions strikes a good compromise: A check is performed for each method execution, and the refinement map is easier to write, particularly if commit blocks are used as described earlier.

Computing the value of $\texttt{view}^I$ in the multiset implementation requires an atomic snapshot of the contents of the entire array `A`. We use the log to solve the problem of taking this atomic snapshot and constructing the updated value of $\texttt{view}^I$. Let *supp*(`view`) denote the set of program variables that influence the computation of $\texttt{view}^I$. The set *supp*($\texttt{view}^I$) can be computed by a simple static analysis of the code for updating $\texttt{view}^I$. In addition to all log entries required for I/O-refinement checking, we insert an entry into the log recording each update to a variable in *supp*($\texttt{view}^I$).

6. The Runtime Refinement Checking Tool

We implemented our refinement checking techniques as a reusable library called VYRD. VYRD implements the functionality of a generic refinement checker and has components for instrumenting implementation code for logging and for checking I/O and view refinement using the log.

6.1 Logging issues

The implementation code is instrumented using the helper classes in VYRD to save actions performed and related data to the log at runtime. The logging mechanism of VYRD uses the binary object serialization mechanism of the .Net platform in order to restore record objects as they are saved at runtime. This gives a transparent view of logging and restoring the implementation trace and relieves the programmers from considerable programming effort.

In the industrial systems we applied VYRD to, storage and access for shared variables were managed and their sequential consistency and atomicity of updates was ensured by a separate module. This allowed us to interpret log entries using a sequential memory model. We set aside the verification of the lower-level storage modules as a future task.

6.2 Logging at different levels of granularity

It is possible to log executions at different levels of granularity. An example of fine-grained logging is to record all writes to shared variables, e.g., assignments to an integer-typed variable, inserts to a hashtable, or locking/unlocking using a semaphore. No knowledge of the data structure is necessary to re-construct data structure state using the log in this case. Coarse-grained logging is data-structure specific. If the programmer can ensure statically that a group of actions are performed atomically by a method, the group of actions can be logged as a single entry, which reduces logging contention and overhead. Such a group of actions typically is a lower-level task than a public method, e.g., re-distributing data between two tree nodes, but higher-level than a single shared variable update. Since such tasks are data structure-specific, "replay" methods that re-construct data structure state from the log must be provided by the programmer for each such kind of log entries.

6.3 Programmer-defined view construction

Defining a view for a concurrently accessed data structure requires knowledge of internals of the implementation. Roughly speaking, view variables must extract the abstract data structure contents by abstracting away the rest of the program state. Since the way abstraction is done is specific to the data structure, the programmer is asked to describe how `view` is to be computed from program state. Currently, `view` is defined by writing a method that computes it using the log. In many examples, including Boxwood, `view`

can be written using shared state only. Once we understood a data structure implementation, even for the industrial examples we studied, we found that it only took a few days of effort to define `view`.

6.4 Incremental computation and comparison of views

The size of the program state for an industrial-scale concurrent data structure could be huge, e.g., the entire contents of a hard disk for a file system. To avoid re-traversing the entire program state at each commit action we compute $\texttt{view}^I$ incrementally. Typically, since $\texttt{view}^I$ represents data structure contents, its value has a certain structure, and one can speak of portions of that structure, e.g., a subset of linked-list entries or files that are modified between two commit points. As we replay the actions of the implementation and the methods in the specification, we perform a dependency analysis of the portions of view variables on the variable updates that took place between two commit actions. We then only re-compute and compare only the relevant parts of the view variables.

7. Experience with Vyrd

We ran VYRD on several programs to test its efficacy in catching errors. In the experiments reported in this section, logging was done at the finer, shared variable write level.

7.1 Test harnesses

To produce traces for refinement checking, we developed test programs for each data structure implementation intuitively aiming to trigger concurrency errors. All test cases on a particular data structure start from the same initial state. Each test program first generates a random pool of keys to be shared by all threads as arguments for method calls. Then the program creates a number of threads (reported in the tables of experimental results) each of which, using arguments randomly chosen from the pool, issues a given number (also reported in tables) of random method calls to the same data structure instance concurrently. The pool is reduced gradually over time to focus more concurrent method calls on a smaller region of the data structure. In implementations with compression mechanisms, the compression thread is either triggered automatically by mutator methods, or, otherwise, it is run continuously.

7.2 Boxwood

`Boxwood` [10] provides scalable storage infrastructure for applications. Boxwood is written in C# and consists of roughly 30K lines of code (LOC). A block diagram for Boxwood is given in Fig. 10. A B-link tree module (`BLinkTree`) is built on Boxwood's data store abstraction: Each shared variable is a byte-array identified by a unique handle, and is stored and managed by the `Chunk Manager` module. Shared variables have version numbers that are incremented after each write. The `Cache` module improves the performance of `BLinkTree` and is intermediate between `BLinkTree` and `Chunk Manager`.

We concentrated on verifying the `BLinkTree` (≈3KLOC) and `Cache` (≈1KLOC) modules, assuming that `Chunk Manager` was implemented correctly. We followed a modular approach to verifying `BLinkTree` and `Cache`. We treated `Cache` as a separate data structure that works in collaboration with `Chunk Manager` and has `BLinkTree` as its client. The verification of `BLinkTree` was performed assuming that the `Cache+Chunk Manager` combination works correctly.

7.2.1 Verification of `Cache` + `Chunk Manager`

`Cache` has public methods for reading and writing shared variables, flushing all dirty variables in the cache to `Chunk Manager`, and a "revoke" method for writing a single variable from `Cache` to `Chunk Manager`. Each variable accessed via `Cache` is addressed by the same unique handle that is also recognized by `Chunk Manager`. We verified the correctness of the abstract data store provided by the `Cache+Chunk Manager` combination assuming that `Chunk Manager` is implemented correctly.

We specified $\texttt{view}^I$ for `Cache+Chunk Manager` as the set of (*handle*,*byte-array*) pairs stored in them. To extract $\texttt{view}^I$, for each *handle*, if there exists a cache entry associated with *handle*, *byte-array* is taken from the cache entry, otherwise, it is taken from `Chunk Manager`. We also verified the following invariants about `Cache` at runtime: (i) If a clean cache entry exists for *handle*, `Cache` and `Chunk Manager` must contain the same corresponding *byte-array*. (ii) a cache entry must be in either the clean or dirty entries list. For `Cache` we logged add and delete operations to hashtables and byte-array-copy operations from/to the cache entry buffers. Logging at this level of granularity was necessary for detecting the concurrency error described below.

```
WRITE(handle, buffer)
 1  RECLAIMLOCK.BEGINREAD()
 2  LOCK(clean)
 3  ce ← GET-CLEAN-ENTRY(handle)
 4  de ← GET-DIRTY-ENTRY(handle)
 5  UNLOCK(clean)
 6  if ce = null and de = null
 7    then
 8         RECLAIMLOCK.ENDREAD()
 9         te ← MAKE-NEW-ENTRY(handle)
10         RECLAIMLOCK.BEGINREAD()
11         COPY-TO-CACHE(buffer, te)
12         LOCK(clean)
13         ADD-TO-DIRTY-LIST(handle, te) ▷ Commit point 1
14         UNLOCK(clean)
15  elseif ce = null
16    then
17         LOCK(clean)
18         REMOVE-FROM-CLEAN-LIST(handle)
19         COPY-TO-CACHE(buffer, ce)
20         ADD-TO-DIRTY-LIST(handle, ce) ▷ Commit point 2
21         UNLOCK(clean)
22    else
23         COPY-TO-CACHE(buffer, de) ▷ Commit point 3
24  RECLAIMLOCK.ENDREAD()

COPY-TO-CACHE(buffer, entry)
1  for i ← 0 to LENGTH-OF(buffer) − 1
2     do
3        entry.data[i] ← buffer[i]

FLUSH()
 1  LOCK(clean)
 2  while more entry in dirty list
 3     do
 4        te ← NEXT-DIRTY-ENTRY()
 5        if te is old enough to flush
 6          then
 7               BOXWOOD-ALLOCATOR-WRITE(te.handle, te.data)
 8               ADD-TO-VICTIMS-LIST(te)
 9  while more entry in victims list
10     do
11        te ← NEXT-VICTIM-ENTRY()
12        REMOVE-FROM-DIRTY-LIST(te)
13        ADD-TO-CLEAN-LIST(handle, te)
14  UNLOCK(clean) ▷ Commit point
```

Figure 8. Pseudocode fragment for the `Boxwood Cache`.

7.2.2 Bug caught in `Cache`

The error is due to a method call (the call to COPY-TO-CACHE in line 23 of Fig. 7.2) not being protected by the proper lock (LOCK($clean$)). As a result, while a byte array in the cache is being overwritten in-place (the COPY-TO-CACHE method), it is possible for a cache flush to be triggered. The cache flush may be interleaved so that a corrupted byte-array that contains partly old and partly new data is written to `Chunk Manager`. At this point, the cache entry is marked "clean" since it is believed to have been written to `Chunk Manager`. Invariant (i) for `Cache` is violated at this point, and a view refinement violation occurs. It is a lot harder for I/O refinement and testing to detect this error: The corrupted state must lead to an error in the return value of a method as described in the following scenario: Before another write to this cache entry occurs, it is evicted from `Cache` but is not written back to `Chunk Manager` since it is marked "clean". Then the block is read back into the cache, which retrieves the corrupted buffer from `Chunk Manager`. In our experiments, both I/O refinement and view refinement detected this error but I/O refinement required a much longer test run.

7.2.3 Verification of `BLinkTree`

The `BLinkTree` module in `Boxwood` is a highly concurrent implementation of a B-link tree data structure [12]. A B-link tree stores a set of $(key, value)$ pairs and provides the INSERT, DELETE, and LOOKUP methods. An internal compression thread works concurrently with data structure operations and re-arranges the tree structure without modifying the set of $(key, value)$ pairs. In addition to the data structure operations, we also checked that atomic updates that the compression thread performs to the program state do not modify `view`. We found no errors in the original code for `BLinkTree`. Bugs were inserted manually in order to collect experimental data for performance measurements.

7.2.4 Defining $\texttt{view}^I$ for `BLinkTree`

$\texttt{view}^I$ was defined to be the sorted list of all the $(key, data)$ pairs in the tree, along with their version numbers. Each $(key, data)$ pair is stored at a data node pointed to by a leaf pointer node (Fig. 10). The list was computed by a left to right traversal of the leaf pointer nodes and adding to the list each data node they point to. The non-data nodes form an indexing structure for the `BLinkTree` and are used for accessing the data nodes efficiently but their structure is abstracted in the computation of $\texttt{view}^I$. Although the indexing structure is manipulated arbitrarily by many threads concurrently, it keeps all the data nodes always visible and accessible even when uncompleted operations are modifying the indexing structure.

7.2.5 Selecting commit actions

The `BLinkTree` mutator methods described in [12] follow a pattern in which the effect of each method is reflected in the data structure state by a single write to a particular data or leaf node variable, while remaining writes re-structure the tree. For INSERT (See Figure 8) and DELETE, we identify this single write and designate it as the commit action. In INSERT, there are four possible commit points for different executions of INSERT depending on what exit path is taken through the method code. These paths are distinguished by whether the key being inserted was already in the tree (line 15), whether the leaf node that is being inserted into needs to be split into two nodes (lines 39 and 46) , and whether the leaf node is also the root node of the tree (line 52). Each of these cases is identified using a boolean expression in terms of method-local variables. The appropriate leaf or data node write operation is then marked as the commit action.

```
INSERT(key, data)
 1  ▷ ptr is the pointer to the current parent node
 2  ▷ p is the pointer to the current node
 3  p ← null
 4  completed ← false
 5  stack ← MOVE-DOWN-AND-STACK()
 6  repeat
 7        ptr ← POP(stack)
 8        repeat
 9              found ← true
10              LOCK(ptr)
11              A ← READ-NODE(ptr)
12              if A is a leaf pointer node and key is in A
13                then
14                      OVERWRITE(A, data)
15                      ▷ Commit point 1
16                      UNLOCK(ptr)
17                      RETURN
18              if key > HIGHVALUE(A)
19                then
20                      UNLOCK(ptr)
21                      found ← false
22                      ptr ← GET-RIGHT-POINTER(A)
23              elseif key < LOWVALUE(A)
24                then
25                      UNLOCK(ptr)
26                      found ← false
27                      ptr ← GET-LEFT-POINTER(A)
28          until found
29        if p = null
30          then
31              ▷ this is the first write and we
32              ▷ are going to write a data node
33              p ← CREATE-DATA-NODE(key, data)
34        if A is safe to insert d
35          then
36              p ← INSERT-INTO-SAFE(p, A)
37              if A is a leaf pointer node
38                then
39                    ▷ Commit point 2
40              completed ← true
41        elseif A is not the root
42          then
43              p ← INSERT-INTO-UNSAFE(p, A)
44              if A is a leaf pointer node
45                then
46                    ▷ Commit point 3
47          else
48              p ← INSERT-INTO-UNSAFE-ROOT(p, A)
49              completed ← true
50              if A is a leaf pointer node
51                then
52                    ▷ Commit point 4
53    until completed
```

Figure 9. The implementation of the `Insert` subroutine in BLink-Tree and the conditional commit point annotations.

7.3 The Scan file system

A previous prototype of VYRD had been applied to the Scan file system [13] which consisted of some 5KLOC. It detected several previously undetected, serious concurrency bugs while running Scan's own performance benchmarks. Interestingly, these bugs were also in the cache module of Scan and were very similar to those found in Boxwood's `Cache`. More details on this example can be found in [13].

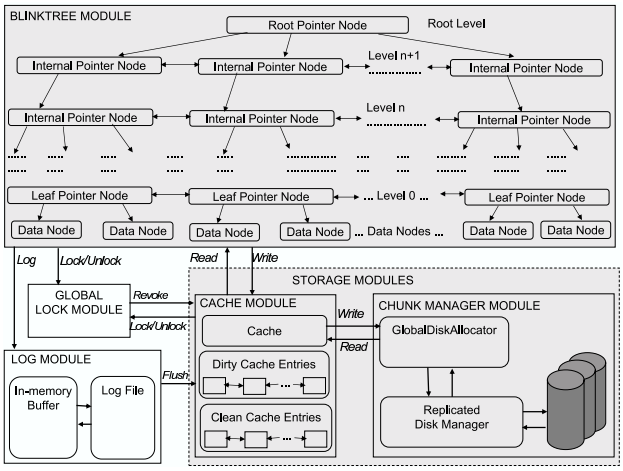

Figure 10. Boxwood block diagram

Table 1. Time to detection of error

Program	Error	#Thrds	I/O Ref.	View Ref.	run-time ratio
`Multiset-Vector`	Moving acquire in FindSlot	4	1308	25	1.03
		8	773	21	
		16	758	10	
		32	820	6	
`Multiset-BinaryTree`	Unlocking parent before insertion	4	3648	736	1.38
		8	930	217	
		16	330	76	
		32	262	78	
`java.util.-Vector`	Taking length non-atomically in `lastIndexOf()`	4	219	219	2.83
		8	58	58	
		16	52	52	
		32	25	25	
`java.util.-StringBuffer`	Copying from an unprotected `StringBuffer`	4	195	90	3.46
		8	152	63	
		16	124	19	
		32	29	17	
`BLinkTree`	Allowing duplicated data nodes	2	2198	405	1.27
		4	4450	483	
		8	3332	611	
		10	2763	342	
		16	1069	301	
		25	3692	515	
		32	2111	715	
Cache	Writing an unprotected dirty cache entry	4	521	14	16.9
		8	805	8	
		10	599	10	
		16	302	29	
		25	539	26	
		32	311	34	

7.4 Microbenchmarks

7.4.1 Multithreaded Java libraries

We worked on the `java.util.Vector` and `StringBuffer` classes, concurrency errors in which had been reported previously [6, 7]. Both examples consisted of approximately 1KLOC. Both I/O and view refinement were able to easily detect the errors.

7.4.2 Multiset

We developed two different Java implementations of `Multiset`: one with an `Vector`-based representation (≈300 LOC), the other based on a binary search tree (≈1KLOC). We included compression threads and a delete operation in both implementations. VYRD was a valuable debugging aid during the development of the multiset implementations. The concurrency errors used in runtime measurements were inserted manually and were all detected.

7.5 I/O refinement vs. view refinement

To compare the effectiveness of I/O and view refinement checking, on the examples described above, we recorded the number of methods executed before the first error was detected by each technique. Table 1 reports the average of results from large numbers of repetitions of the same experiment. The last column presents the ratio of the CPU time required for running VYRD in view refinement mode to that of running it in I/O refinement only mode on the same trace. View refinement's superiority in early detection is apparent from these results. The additional cost of checking view refinement must be traded off with its ability to detects errors much earlier in the trace.

The only error in `java.util.Vector` was in an observer method and typically manifests itself without corrupting data structure state. Therefore view refinement was no better at detecting it than I/O refinement in our tests.

7.6 Computational overhead measurements

Table 2 compares the overhead during runtime due to logging only with the runtime of the unmodified program, assuming I/O or view refinement will be run using the log later. All numbers reported are CPU seconds on a 2.4 GHz Intel Pentium PC with 1GB of RAM. In the first three examples and in `Cache`, in which the mutator methods perform a large number of shared variable writes, the overhead of logging information required for view refinement is significantly larger than that for I/O refinement. This is because these examples require logging at very fine granularity. In `java.util.Vector`, `java.util.StringBuffer` and `BLinkTree`, the difference between logging overhead for I/O and view refinement is much less.

Table 3 presents results on the computational cost of checking view refinement and compares it with the computational cost of running the program and logging information. All running times are CPU seconds on the same machine. The impact of instrumentation and logging, and the computational cost of checking view refinement are tolerable, especially for the industrial examples.

8. Related work

Many correctness criteria for concurrent systems (e.g., atomicity in the work of Flanagan and Qadeer [7] and Wang and Stoller [16] require that non-atomic (interleaved) executions of the implementation be "equivalent" to an atomic, sequential run of the implementation. The definition of equivalence used in these criteria does not make reference to the specification for the system. I/O-refinement and view refinement are different from these criteria in that they do not require the existence of such an atomic, sequential run *of the implementation*. Therefore, we believe that they rule out fewer useful practical implementations.

The choice of the definitions for I/O and view refinement were motivated by our experience with verifying two industrial-scale concurrent programs, neither of which could be proven using using reduction (e.g. as in [7]), the concept of "commit atomicity" as in [4], or by proper selection of "pure" code blocks as defined in [5]. The intuition behind this fact is illustrated by the following simple example inspired by Boxwood. Consider a tree implementation of a set data structure in which data is stored at the leaves only,

Table 2. Overhead of logging

Implementation	Program	I/O Ref.	View Ref.
`Multiset-Vector`	15.4	0.39	3.69
`Multiset-BinaryTree`	1.1	0.135	0.53
`Vector`	0.20	0.09	0.12
`StringBuffer`	0.92	0.18	0.24
`BLinkTree`	56.2	2.42	2.63
Cache	1.8	1.67	3.31

Table 3. Running time breakdown

Program	#Thrd/ #Mthd	Prog. alone	Prog.+ logging only	Prog.+ logging and VYRD	VYRD alone (off-line)
`Multiset-Vector`	10/50	15.4	23.4	106.8	82.5
`Multiset-BinaryTree`	10/50	1.1	4.4	7.8	3.1
`Vector`	20/200	0.2	0.32	2.46	2.03
`StringBuffer`	10/30	0.92	1.16	2.1	1.85
`BLinkTree`	10/600	56.2	58.9	213.18	157.32
Cache	10/500	1.8	5.11	9.5	4.45

while internal nodes (the "indexing structure") enable search. In the following, let p denote a shared variable a write to which changes the abstract data structure contents, i.e., a write which changes the contents of a leaf. Let q denote a shared variable that represents an internal node of the tree. In the example below, writes to q re-arranges the shape of the data structure, for example, by re-balancing the tree. Suppose all writes to p and q are lock protected. Consider the interleaving of actions by two threads t_1 and t_2 concurrently executing two methods μ_1 and μ_2 respectively:

$$\begin{array}{lcccc} t_1\text{:} & W(p) & & & W(q) \\ t_2\text{:} & & W(p) & W(q) & \end{array}$$

Because t_1 and t_2 perform multiple, lock-protected writes to the same set of shared variables that do not commute with each other, it is not possible to convert this execution using reduction or purity to a method-atomic execution of the implementation. However, since only the writes to p modify the abstract data structure state, refinement checking can be performed using the witness ordering of μ_1 followed by μ_2 and will not report any refinement violations.

I/O refinement can be viewed as a variant of linearizability [8], where the "sequential specification" for the data structure implementation is given in the form of an executable specification, and may contain non-determinism in order to allow termination scenarios not possible in a sequential execution.

Checking refinement as a verification approach is well-studied (See [1, 11] among many others). Runtime checking of conformance to a state invariant derived from an object model has been investigated [3]. Runtime checking of property annotations inserted into implementation code has been studied [2]. Runtime analysis methods for correctness criteria such as atomicity have been developed [6, 16]. Refinement checking has recently been integrated with simulation-based validation of hardware designs [15]. The commit-atomicity work of Flanagan [4], published simultaneously with an earlier version of the work described here [14], is closely related but is a very restricted case of the work presented here. In commit atomicity, the specification is required to be the atomized version of the implementation, refinement checking is done only at quiescent points rather than at each commit point, and a complete match between specification and implementation states is required whereas we, by using view variables, can declare more intuitively equivalent program states as matches. Our work is the first general tool that can check at runtime that an industrial-scale concurrent data structure implementation refines a specification.

9. Conclusion

Run-time checking of refinement promises to be a powerful verification approach with reasonable computational cost. In this paper, we presented two notions of refinement and techniques for checking them. We applied these techniques to the Boxwood project and presented experimental results demonstrating the efficacy of and the trade-offs offered by our techniques.

Acknowledgments

The work reported in this paper was supported by Microsoft Research, Redmond and Silicon Valley. We would like to thank Jim Larus, Roy Levin and Sriram Rajamani for making our collaboration possible. We also thank the Boxwood group, particularly Chandu Thekkath and Lidong Zhou for their support with understanding and running Boxwood. Serdar Tasiran would like to thank Martin Abadi for stimulating discussions.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proc. 3rd Annual Symposium on Logic in Computer Science*, pp. 165–175. IEEE Computer Society Press, 1988.

[2] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In *Electronic Notes in Theoretical Computer Science*, Vol. 89. Elsevier, 2003.

[3] M. L. Crane and J. Dingel. Runtime conformance checking of objects using Alloy. In *Electronic Notes in Theoretical Computer Science*, Vol. 89. Elsevier, 2003.

[4] C. Flanagan. Verifying Commit-Atomicity Using Model-Checking. In *SPIN '04: The SPIN Workshop on Model Checking of Software*. Springer-Verlag, 2004.

[5] 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.

[6] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM Symposium on Principles of Programming Languages*, pp. 256–267, 2004.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN 2003 Conf. on Programming language design and implementation*, pages 338–349. ACM Press, 2003.

[8] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[9] M. Ji and E. Felten. Scan-based scheduling and layout in a reliable write-optimized file system. Technical Report TR-661-02, Princeton University, Department of Computer Science, 2002.

[10] 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`

[11] S. Park and D. L. Dill. Verification of cache coherence protocols by aggregation of distributed transactions. *Theory of Computing Systems*, 31(4):355–376, 1998.

[12] Y. Sagiv. Concurrent operations on b-trees with overtaking. *Journal of Computer and System Sciences*, 3(2), Oct. 1986.

[13] S. Tasiran, A. Bogdanov, and M. Ji. Detecting concurrency errors in file systems by runtime refinement checking. Technical Report HPL-2004-177, HP Laboratories, 2004.

[14] 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.

[15] S. Tasiran, Y. Yu, and B. Batson. Using a formal specification and a model checker to monitor and guide simulation. In *Proceedings of the 40th Design Automation Conference*, pages 356–361. ACM, 2003.

[16] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *Electronic Notes in Theoretical Computer Science*, Vol. 89. Elsevier, 2003.