Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 113 (2005) 163–179
www.elsevier.com/locate/entcs

Runtime Refinement Checking of Concurrent Data Structures

Serdar Tasiran

Koç University, Istanbul, Turkey

Shaz Qadeer

Microsoft Research, USA

Abstract

We present a runtime technique for checking that a concurrent implementation of a data structure conforms to a high-level executable specification with atomic operations. The technique consists of two phases. In the first phase, the implementation code is instrumented in order to record information about its execution into a log. In the second phase, a verification thread runs concurrently with the implementation and uses the logged information to check that the execution conforms to the high-level specification. We pay special attention to reducing the impact of the runtime analysis on the concurrency characteristics and performance of the implementation. We are currently applying our technique to Boxwood [1], a distributed implementation of a B-link tree data structure.

Keywords: Runtime verification, concurrent data structures, refinement, atomicity

1 Introduction

For data structure implementations that can be accessed by concurrent application threads, verifying refinement with respect to a high-level executable specification is more thorough than property verification alone. In this work, we present a technique for checking refinement at runtime.

Proving refinement using static techniques requires reasoning about the entire state space of the implementation. Traversing every implementation state is infeasible or impossible for most non-trivial systems. To apply theorem-proving techniques to refinement proofs, one must devise and prove a representation invariant about the program state. These are tedious and com-

doi:10.1016/j.entcs.2004.01.028

putationally costly tasks. Furthermore, when modifications are made to an implementation during its development, invariants may need to be modified and re-proven as well. As a result, for practical programs, it is not feasible to check refinement statically.

Compositional methods have been explored to divide refinement checks into smaller, computationally managable subproblems. While there is well-studied theory for compositional reasoning, modular proofs of refinement are difficult to carry out in practice. When applying a verification tool on a component of a design, an abstract model of the component's environment needs to be devised and verified. Such environment assumptions and interface specifications may not have been made explicit in the design description, and thus may need to be guessed and verified. Furthermore, coordination of proof sub-tasks as the program evolves is difficult and error-prone.

Checking refinement during runtime reduces both the computational and the human effort required. The computational effort is reduced simply because, instead of the entire state-space, only the states along execution paths of test cases are examined. The human effort required for verifying a component of a design is reduced because the component is run as part of the actual program implementation and all executions are naturally legal executions of the component's environment. Thus, there is no need to devise or verify an environment model, i.e., a test stub for the component. These advantages have motivated us to investigate runtime refinement checking as a verification paradigm.

We use state transition systems to give semantics to the implementations and specifications of concurrent data structures. Section 2 formalizes state transition systems and our notion of refinement. We illustrate our runtime verification technique on a concurrent implementation of a multiset described in Section 3. Sections 4 and 5 present our technique for checking refinement at runtime. We discuss related work in Section 6 and conclude in Section 7.

2 Preliminaries

2.1 State transition systems

We focus on concurrently accessible implementations of data structures written in object-oriented languages. The data structure makes available a number of operations each of which is implemented as a method. When it is necessary to distinguish these methods from methods only internally accessible by the data structure, we refer to them as *public methods*. Several application threads can issue calls to methods concurrently and portions of method executions can be interleaved for better performance. Throughout this paper, the domain *Tid*

represents the set of thread identifiers. Tid is the union of two disjoint sets, Tid_{app} and Tid_{ds}. The set Tid_{app} contains identifiers of applications threads that call the public methods provided by the data structure. The set Tid_{ds} contains identifiers of worker threads in the implementation used to perform tasks internal to the data structure.

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

- $\mathcal{V}$ is the set of *program variables*. These variables include, for example, the heap-allocated data shared among the threads as well the thread-local stacks, one for each thread.
- 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. If $\delta(s, a) = s'$, the transition system may perform the action a in state s to change the state to s'. We denote such a transition by $s \xrightarrow{a} s'$.

A *run* of the state transition system is a finite sequence $r = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ for some $n \geq 0$ such that $s_i \xrightarrow{a_i} s_{i+1}$ for all $0 \leq i < n$.

The set $Actions$ consists of *visible* and *invisible* actions. Actions corresponding to calls and returns of public methods are required to be visible actions. A *call action* is a tuple $(t, \text{Call}, name, args)$, where t is the identifier of the thread performing the method call, *name* is the name of the invoked public method, and *args* is the list of method arguments. A *return action* is a tuple $(t, \text{Return}, name, rets)$, where t is the identifier of the thread performing the method return, *name* is the name of the returning public method, and *rets* is the list of values returned by the method. Given a run r of the state transition system, the *trace* corresponding to r is the sequence of visible actions that take place during that run.

A sequence of call and return actions by an application thread t is *well-formed* if it satisfies the following three properties:

(1) The first action is a call action.

(2) Every call action $(t, \text{Call}, name, args)$ must be followed by a return action $(t, \text{Return}, name, rets)$.

(3) If the return action $(t, \text{Return}, name, rets)$ is not the last action in the sequence, it must be followed by a call action $(t, \text{Call}, name', args')$.

A trace τ is *well-formed* if for every application thread t, the subsequence of τ corresponding to actions of thread 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. 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* of that method.

A well-formed run is *atomic* if after every call action $(t, \text{Call}, name, args)$ to a public method *name* by an application thread t, no thread other than thread t performs an action until the corresponding return action $(t, \text{Return}, name, rets)$ has occurred. Thus, every atomic run is a concatenation of *fragments* of the run, such that only one application thread performs actions in any particular fragment. Each fragment begins with a call action by an application thread t and ends with the corresponding return action by thread t. A state transition system is *atomic* if all of its runs are atomic.

The call action at the beginning and the return action at the end of a fragment form the *signature* of the fragment. An atomic state transition system is *deterministic* if whenever two fragments of any two runs have the same first state and the same signature, they have the same last state as well. Thus, every trace of an atomic and deterministic state transition system is produced by a unique run.

2.2 Refinement between state transition systems

Let the implementation of a concurrent data structure be given by the state transition system $\mathcal{I} = (\mathcal{V}^{\mathcal{I}}, S^{\mathcal{I}}, s_0^{\mathcal{I}}, \delta^{\mathcal{I}})$, and the specification be given by the state transition system $\mathcal{S} = (\mathcal{V}^{\mathcal{S}}, S^{\mathcal{S}}, s_0^{\mathcal{S}}, \delta^{\mathcal{S}})$. For a trace τ and a thread t, let $\tau|_t$ denote the projection of τ onto actions associated with thread t. The transition system $\mathcal{I}$ *refines* $\mathcal{S}$ if for every trace τ of $\mathcal{I}$, there is a trace τ' of $\mathcal{S}$ such that $\tau|_t = \tau'|_t$ for all $t \in Tid_{app}$.

Note that the definition of refinement depends on the set of actions of the state transition system chosen to be visible. In our notion of refinement, called *I/O refinement*, call and return actions are the only visible actions. Intuitively, this definition of refinement expresses the requirement that the return values of method calls be consistent with those in some (atomic) execution of the specification. Of course, we could make more actions of the implementation visible and require them to be matched against corresponding actions in the specification. A larger set of visible actions will result in more extensive runtime checking than that allowed by I/O refinement. Section 5 describes one such choice of visible actions that we have found useful in practice.

3 Example

We will use a multiset data structure as our running example throughout the paper. In this section, we give the specification and implementation of the multiset. The multiset data structure supports two operations:

- InsertPair(x,y) to insert the elements x and y into the multiset.
- LookUp(x) to check if x is an element of the multiset.

The specifications of these operations are presented in Figure 1 where M is a state variable that represents the multiset contents. Any invocation of the

```
InsertPair(x, y)
1  status ← success or failure
2  if status = success
3     M ← M ∪ {x, y}
4  return status
```

```
LookUp(x)
1  return (x ∈ M)
```

Fig. 1. The specification of the multiset operations

InsertPair operation is allowed to fail or succeed. The specification nondeterministically makes this choice, and updates the specification state accordingly. The nondeterministic choice is made visible via the return value of the method.

The following multiset implementation uses an array A[1..n] to store the multiset elements. Each array element has two fields:

- A[i].content is the multiset element stored in A[i]. It has value *null* if no element is stored at location i.
- A Boolean variable A[i].valid indicates whether the element in position i is valid.

Initially, A[i].content = *null* and A[i].valid = *false* for all i. The implementation makes use of an array of locks L[1..n], where lock L[i] protects the contents of A[i].

The implementation of the multiset uses a subroutine FindSlot (Figure 2), which looks for an available slot in the array for a single element x. If FindSlot finds an available slot, it reserves it by setting its content to x and returns its index. If no slot is available, it returns 0.

The implementation of InsertPair (Figure 3) uses two calls to FindSlot to allocate slots in the array A for x and y. If either of the calls fail, InsertPair returns *failure* and frees up any slots it may have reserved. Otherwise, it acquires the locks protecting the allocated slots, sets the valid fields of those slots to *true*, and returns *success*.

```
FindSlot(x)
1     for i ← 1 to n
2       Acquire(L[i])
3       if (A[i].content = null)
4         A[i].content ← x
5         Release(L[i])
6         return i
7       else
8         Release(L[i])
9     return 0
```

Fig. 2. The implementation of the FindSlot subroutine

```
InsertPair(x,y)
1     i ← FindSlot(x)
2     if (i = 0)
3       return failure
4     j ← FindSlot(y)
5     if (j = 0)
6       A[i].content ← null
7       return failure
8     Acquire(L[i])
9     Acquire(L[j])
10    A[i].valid ← true
11    A[j].valid ← true
12    Release(L[i])
13    Release(L[j])
14    return success
```

```
LookUp(x)
1     for i ← 1 to n
2       Acquire(L[i])
3       if (A[i].valid and
4           A[i].content = x)
5         Release(L[i])
6         return true
7       else
8         Release(L[i])
9     return false
```

Fig. 3. The implementation of the multiset operations

The implementation of LookUp (Figure 3) is straightforward. The fact that LookUp acquires the lock for each array entry before it accesses the data makes sure that LookUp does not access data only partially modified by an InsertPair operation. The multiset implementation I/O refines its specification. An argument for this fact is provided in Section 4.1 and made precise in the Appendix.

4 Runtime refinement checking

In this section, we present a method for checking at runtime that a non-atomic implementation $\mathcal{I}$ refines a deterministic, atomic specification $\mathcal{S}$. Suppose we have a well-formed run r of $\mathcal{I}$. Let τ be the trace corresponding to the run r

and $\tau|_t$ denote the projection of τ onto the actions of an application thread t. For each such thread identifier t, the sequence $\tau|_t$ is a sequence of pairs of actions, where each pair consists of a call action $(t, \text{Call}, name, args)$ followed by a return action $(t, \text{Return}, name, rets)$.

The most straightforward method for checking refinement is to construct all possible atomic interleavings of the sequences in $\{\tau|_t \mid t \in Tid\}$, and for each interleaving check whether it is a trace of $\mathcal{S}$. If one of these interleavings is found to be a trace of $\mathcal{S}$, then we have found a run of $\mathcal{S}$ matching r. Such an interleaving is called a *witness* to the refinement. It is straightforward to check whether an atomic interleaving is a trace of $\mathcal{S}$. Since $\mathcal{S}$ is atomic and deterministic, each of its traces corresponds to a unique run. We simply execute the specification one method call at a time in the order given by the interleaving, looking for the unique run whose trace matches the interleaving. If, as in Figure 1, there is a nondeterministic choice in the return value for a method, the choice is made to conform with the return action specified in the interleaving. If at any point, it is not possible to execute the specification while conforming to the return action, the refinement check is said to fail.

To check refinement for a run r, the number of possible interleavings that might need to be evaluated increases very rapidly with the number of application threads and the number of method calls (and returns) performed by each thread. Thus, it is impractical to check all possible interleavings. To overcome this problem, we infer a witness interleaving from the run itself. We require that the programmer introduce a small number of extra annotations in the program. How these annotations are used to infer the witness interleaving is explained in Section 4.1.

We stress that this notion of a witness interleaving is essential for testing concurrent data structures effectively. In the absence of a witness interleaving, to decide whether a method's return value or the final state of a multi-threaded test program is consistent with the specification, one is forced to either consider all possible interleavings or be overly permissive.

4.1 Commit actions

A thread performs a number of actions during its execution of a method, beginning with a call action and ending with a return action. To derive a witness interleaving from a run of the implementation, we require the programmer to designate a single action within each such execution of a method as the *commit action*. The ordering of these commit actions, one for each method call by a thread, gives us the witness interleaving. In practice, commit actions are specified by designating certain lines in the implementation code to be *commit points*. When one of these lines is executed, the corresponding ac-

tion is marked as the commit action. Any line in the implementation can be a commit point. However, the programmer must make sure that for each method, exactly one of these lines is executed for every execution path through the method. Intuitively, the order of the commit actions in time is meant to coincide with the application's view of how the state of the data structure transforms over time. In other words, the process of selecting the commit actions can be seen as a simplified way of constructing an abstraction map that relates the implementation to the specification.

Intuitively, the commit point of an execution of a method by a thread is the first line whose execution changes the view of the data structure afforded to the other threads. In the multiset implementation of Section 3, the commit point in an execution of the InsertPair method that returns *success* is line 12. Suppose an application thread t is executing the InsertPair method and that this execution is going to succeed. Let p and q be the locations where thread t is going to insert `x` and `y` respectively. Another thread t' can access `A[`p`]` and `A[`q`]` either (1) before thread t executes line 12, or (2) after thread t executes line 12. Consider the first case. Thread t holds the locks `L[`p`]` and `L[`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 12, it must see them as *false*. Consequently, thread t' cannot see either `x` or `y` as having been inserted into the multiset. Consider the second case. Since `A[`p`].content` $=$ `x`, `A[`q`].content` $=$ `y`, and `A[`p`].valid` and `A[`q`].valid` are *true*, thread t' sees both `x` and `y` as having been inserted into the multiset.

Observe that, for the InsertPair method, the method call action, the commit action and the return action may have an arbitrary number of interleaved actions by other application threads separating them. Thus, a witness interleaving based on the ordering of method call or return transactions would be in error. A complete proof, based on commit actions, of the fact that the multiset implementation I/O refines its specification is provided in the Appendix.

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.

Annotating the implementation with commit points is extra effort for the programmer. The reward for this effort is the capability to perform runtime checking of refinement efficiently. In addition, the process of analyzing the implementation using these terms may itself expose design flaws and result in a better design, even before runtime verification is used. Our experience with

two practical designs, the Scan filesystem [8] and Boxwood distributed B-tree implementation [1] confirms this observation.

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 the verification method introduces a large amount of instrumentation code and computation overhead or application and data structure threads need to contend with each other for access to instrumentation data structures. At the very least, during runtime refinement checking, the implementation's progress should not be blocked while waiting for the verification code. To interfere minimally with the implementation, we run the verification as a separate thread which is informed about the implementation's actions through a log. The implementation threads write entries to the log as they run; the verification thread reads these entries and performs refinement checking. In its simplest form, the entries of the log used in refinement checking are the following actions of the state transition system.

- The call action for each method invoked by a thread.
- The return action for each method completed by a thread.
- The commit action for each method invoked and completed by a thread.

The actions appear in the log in the order in which they are executed by the implementation. One way to achieve this is to require that each logged action be performed atomically with the corresponding log update. A relaxation of this requirement is described in Section 5.1. The threads use a lock to synchronize access to the log. Each read or write to the log takes a small amount of time. Consequently, the lock is held by each thread only for a short duration and the impact of lock acquire operation on the concurrency characteristics of the implementation is minimal.

Many concurrent data structures used in distributed systems, such as the Boxwood project[1], implement similar logs to restore system state reliably in case of a crash. With some modifications, the logging mechanisms in such systems can be reused for the purpose of verification. Further, the fact that such systems tolerate the interference caused by a logging mechanism for recovery is evidence that the impact of logging for the purpose of verification may also be tolerable.

5 Improving I/O refinement

The coverage accomplished by runtime verification based on I/O refinement is particularly sensitive to the sequence of method calls performed by the threads in the test program. As an extreme example, consider a test program for the multiset of Section 3 that only calls the method InsertPair and never calls LookUp. Since the specification of InsertPair allows both *success* and *failure* as return values, the runtime checks for I/O refinement would pass trivially. For useful checking, the test program must perform a number of calls to LookUp. But introducing a large number of calls to LookUp might not be desirable, as the concurrency characteristics of the program under this workload may be significantly different from regular use. Even if the test program did perform calls to LookUp, these calls may not get scheduled at the most interesting points in the execution. I/O refinement as a correctness criterion is thorough enough for static checking but needs to be strenghtened for runtime checking.

In this section, we augment the correctness criterion of I/O refinement to enable more thorough runtime verification of concurrent data structures. The fundamental idea is to introduce an auxiliary variable `view` and specify how the implementation and the specification of the data structure update it. The implementation itself is not actually modified. Instead, the verification thread constructs what would have been the value of `view` in the implementation using information from the log as explained in Section 5.1. Intuitively, the value of `view` captures a partial view of the data structure – a view that is updated atomically and on which the value returned by the method depends. `view` is supposed to abstract away information that is not relevant to an application's view of the data structure state. For example, for a hash table, `view` might be the set of key-value pairs while the hash function itself is abstracted away.

The variable `view` is initialized to the same value in both the implementation and the specification. In the implementation, it is updated once atomically by each method at its commit point. In the specification, it is updated once atomically anytime between the call and return of each method. During runtime verification, we now also check that both the implementation and the specification perform the same sequence of updates to `view`. Formally, we make the commit action a new visible action, and annotate it with the information about how `view` is updated.

We now illustrate this method on the multiset example. We select the variable `view` to be the specification variable `M` itself. Since no new variable is introduced into the specification, the specification of InsertPair and

LookUp remain unchanged. The following code indicates how `M` is to be updated at each commit point in the implementation.

```
do_atomically {
  M ← ∅
  for i←1 to n
    lockOK = (L[i] not held by any thread) or
             (L[i] held by thread currently committing) or
             (L[i] held by a thread executing a LookUp)
    if (A[i].valid and lockOK)
      M ← M ∪ {A[i].content}
}
```

Fig. 4. The computation of `view` for the multiset implementation

Observe that lock acquisitions and releases must also be logged to compute `view` in this case. With the addition of the auxiliary variable `M` to the implementation, we get useful checking even with a test program that has no calls to LookUp. Now the refinement checking ensures that the implementation updates `M` in the same fashion as the specification. Note that the return value of any LookUp operation is uniquely determined by the value of `M` at the commit point of that particular invocation of LookUp. Therefore, after a successful refinement check, we are guaranteed that had the test program had calls to LookUp, those calls would have returned the correct result.

This stronger correctness criterion is more likely to expose errors and provide early warnings as the following example demonstrates. Consider a version of the multiset data structure that also supports a Remove operation. Suppose that a thread in the test program inserts an element a into the multiset twice, but, because of an error in the implementation, only the first a gets inserted into the array `A`. To expose the error through testing or I/O refinement checking alone, we need an execution that inserts a twice, followed by a removal of a, followed by a lookup of a. The probability of generating such an execution would be low. Even if such a test scenario were exercised, if the insert operations, the remove operation, and the lookup operation were separated from each other by large number of other method calls, then it would be difficult to locate the soucrce of the discrepancy. The use of `M` in the multiset will detect this error immediately after the attempt to insert the second copy of a.

To see how a definition of `view` other than the entire specification might be useful, consider a specification for the multiset written as a binary search tree with atomic operations. The computation of `view` for this specification would traverse the tree and insert the elements at the nodes into the multiset

represented by `view`. In this scenarion, `view` is used as a device to extract a canonical representation of the data structure state from the specification as well as the implementation. Such a device is useful if the specification itself contains detail in addition to the abstract view of the data structure state. Additional detail of this kind may make writing a specification easier. For instance, in the B-link data structure implemented by Boxwood, an indexing structure is needed to be able to update the data stored in the specification state. However, the indexing structure is not part of the abstract view of the data structure state.

5.1 Logging information to compute `view`

Computing the value of `M` in the multiset implementation requires an atomic snapshot of the contents of the entire array `A`. Achieving an atomic snapshot of the array in the presence of concurrency is difficult. The naive approach of acquiring all locks `L[1]` to `L[n]` would be very costly and in addition would radically change the concurency characteristics of the implementation. We use the log to solve the problem of taking this atomic snapshot as well. Suppose the set of program variables that influence the computation of `view` is *supp*(`view`). In addition to recording method call and return actions and commit actions, we also insert an entry into the log recording each update to a variable in *supp*(`view`). The set *supp*(`view`) can be computed by a simple static analysis of the code for updating `view`.

Intrumentation of the updates to implementation variables introduces more computational overhead and possibly effects the concurrency characteristics of the implementation more than checking I/O refinement only. For this reason, it may be necessary to perform performance optimizations in the logging process. For instance, if it can be proven that inside the body of a method a code block has exclusive modify access to a set of variables V, then the entire update to the set can be written as a single entry to the log. In the actual execution, other updates to other implementation variables may have been interleaved with updates to V. However, the exclusive modify access guarantees that the interleaving does not interfere with the modifications entered into the log. In particular, at each commit point, the values of the variables obtained using this logging method are the same as they would have been without this optimization.

6 Related work

Checking refinement as a verification approach is well-studied (See [2,9] among many others). Runtime checking of conformance to a state invariant derived

from an object model has been investigated [4]. Runtime checking of property annotations inserted into implementation code has been studied [3]. Runtime analysis methods for correctness criteria such as atomicity have been developed [5,11]. Refinement checking has recently been integrated with simulation-based validation of hardware designs [10]. Our work is the first attempt to check refinement of an executable, algorithm-level specification of a data structure at runtime.

In the following section, we contrast our notion of refinement with two well-known correctness criteria for concurrent systems.

6.1 Comparison with atomicity and linearizability

Many correctness criteria for concurrent systems (e.g., atomicity in the work of Flanagan and Qadeer [6] and Wang and Stoller [11], linearizability in the work of Herlihy and Wing [7]) 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 is different from these criteria in that it does not require the existence of such an atomic, sequential run. Therefore, we believe that it rules out fewer potentially useful implementations. Evidence for this belief is provided by the following claim.

Claim 6.1 *The implementation of the multiset data structure in Section 3*

(i) *is not atomic according to the definition in [6], and*

(ii) *is not linearizable according to the definition in [7].*

Proof.

(i) We provide an execution of the implementation that is not atomic, i.e., cannot be transformed to an atomic run by swapping left and right movers with other actions when appropriate. Consider a multiset implementation and two threads, t_1 and t_2. Thread t_1 performs the single method call InsertPair(1,2) and thread t_2 performs the single method call InsertPair(3,4). For the purpose of illustration, let us divide the code for the implementation of the InsertPair method into three parts $\mathbf{I}_1$ (lines 1-3), $\mathbf{I}_2$ (lines 4-7), and $\mathbf{I}_3$ (lines 8-14). Consider the interleaving of threads in Figure i.

Invocation of $\mathbf{I}_1$ by threads t_1 and t_2 acquires and releases the same set of locks. Hence it is not possible to reorder the execution of $\mathbf{I}_1$ by thread t_1 with the execution of $\mathbf{I}_1$ by t_2. Therefore, it is not possible to obtain an atomic run from this sequence by applying reordrering operations.

Time	1	2	3	4	5	6
t_1	$\mathbf{I}_1$				$\mathbf{I}_2$	$\mathbf{I}_3$
t_2		$\mathbf{I}_1$	$\mathbf{I}_2$	$\mathbf{I}_3$		

Fig. 5. Interleaving example for Claim 6.1 part (i)

(ii) We provide an execution of the multiset implementation that is not linearizable. Consider an implementation with an array `A` of size two, and two application threads t_1 and t_2, concurrently invoking the InsertPair method. For the following interleaving of threads, both calls to InsertPair fail.

Time	1	2	3	4
t_1	$\mathbf{I}_1$		$\mathbf{I}_2$	
t_2		$\mathbf{I}_1$		$\mathbf{I}_2$

Fig. 6. Interleaving example for Claim 6.1 part (ii)

The first call to FindSlot by each thread succeeds. At this point, both slots in the array `A[1..2]` are taken and subsequent calls to FindSlot by each thread return 0. As a result, both calls to InsertPair fail.

In a linearized execution of the implementation, the first invocation of InsertPair always succeeds. As a result, the execution of the implementation depicted above is not equivalent to any sequential execution of the implementation, which proves that this multiset implementation is not linearizable.

□

If an implementation $\mathcal{I}$ is linearizable and is a correct implementation of a specification $\mathcal{S}$ as defined in [7], then $\mathcal{I}$ I/O refines $\mathcal{S}$. For non-linearizable executions, the verification condition in [7] becomes vacuous.

7 Discussion

Run-time checking of refinement promises to be a powerful verification approach with reasonable computational cost. In this paper, we investigated two notions of refinement and techniques for checking them. We are in the process of applying these techniques to an industrial software design.

The effort required for writing formal specifications has been a barrier in the way of widespread application of refinement-based verification methods,

and the capacity limits of formal verification tools have not provided enough of an incentive for overcoming this barrier. To be able to apply run-time refinement checking to systems without formal specifications, we observe that a non-concurrent version of the implementation can serve as a deterministic specification, with minor modifications. A non-concurrent version of an implementation can, for instance, be obtained by making all data structure operations synchronized. A specification obtained in this manner may need to be made more permissive, i.e., the preconditions that enable a method to return a particular value may need to be relaxed. For instance, to obtain a specification from the implementation of the multi-set in Example 3, the specification for InsertPair must allow the method to return "failure" even when there is space in the array for the method to succeed. Also, whenever more than one return value is possible for a method, the synchronized version of the method must be written in such a way that, given the return value, it updates the data structure state exactly as the original implementation would have for that particular return value. We believe that the possibility of using a modified version of the implementation itself as a first specification may make refinement checking more easily applicable to data structures.

Acknowledgments

We thank Chandu Thekkath and Lidong Zhou for support with and discussions about Boxwood, Jim Larus and Roy Levin for making our collaboration possible, and Martin Abadi for enlightening discussions. We also thank Minwen Ji and Andrej Bogdanov for collaboration on the verification of the Scan file system.

References

[1] http://research.microsoft.com/research/sv/Boxwood.

[2] M. Abadi and L. Lamport. The existence of refinement mappings. In *Proceedings of the Third Annual Symposium on Logic in Computer Science*, pages 165–175. IEEE Computer Society Press, 1988.

[3] F. Chen and G. Rosu. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[4] M. L. Crane and J. Dingel. Runtime conformance checking of objects using Alloy. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[5] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[6] C. Flanagan and S. Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 1–12. ACM Press, 2003.

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

[8] 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.

[9] 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.

[10] 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.

[11] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

Appendix

Claim 7.1 *The implementation of the multiset in Section 3 I/O refines its specification.*

Proof. Given an execution of the implementation, let us define a commit action for each method call. For calls to InsertPair that succeed, the commit action corresponds to line 12. For calls to InsertPair that fail, we define the commit action to be the call action to InsertPair. In this latter case, there are many other possible choices since failing calls to InsertPair do not modify the data structure state.

For calls to LookUp, we define the commit action as follows:

- For calls that return *true* the commit action corresponds to line 5 of LookUp gets executed.
- For calls that return *false*, the commit action is the call action.

Let $\mu_1, \mu_2, ..., \mu_n$ be the sequence of method calls ordered according to their commit points in time. Let ν_i be the return value of μ_i in the actual execution of the implementation. We must prove that the return values of methods in the multiset implementation are consistent with what the specification dictates given this particular witness interleaving of method calls.

The argument for calls to InsertPair is easy to make, since each call to InsertPair is allowed to return *success* or *failure* independently from the specification state. Return values for LookUp are more interesting. Let us suppose that μ_k is LookUp(`x`). There are two possible cases.

- LookUp(`x`) returns *true*. We need to prove that there was a call to InsertPair that committed earlier, had a return value of *success* and had `x` as one of its arguments.

LookUp(x) returns *true* only when it finds i such that A[i].valid is *true* and A[i].content = x. It also acquires lock L[i] before it checks for these conditions. Only InsertPair contains lines that can have set A[i].valid to *true*. Let us denote by μ_* the last (and, in fact, only) invocation of InsertPair that set A[i].valid to *true*. It follows from the code for InsertPair and LookUp that A[i].valid is never set to *false* and A[i].content never gets modified afterwards. By the definition of μ_*, μ_k is able to acquire the lock L[i] only after μ_* executes line 12 and releases L[i]. This proves that μ_*'s commit action comes before that of μ_k and had μ_* had x as one of its arguments.

- LookUp(x) returns *false*. We must prove that there exists no call to InsertPair that committed earlier than μ_k, had a return value of *success* and had x as one of its arguments.

  Let us assume otherwise, i.e., that there was such a call $\mu_\#$ that set A[i].content to x, had a return value of *success*, i.e., set A[i].valid to *true*, and released lock L[i] before the commit point of μ_k, i.e., before μ_k starts execution. By the argument in the first case above, the fields of A[i] have not been modified after they are set by $\mu_\#$. Thus, when μ_k acquires L[i] in iteration i, LookUp(x) would find that all the necessary conditions to return *true* are satisfied. This contradicts the fact that LookUp(x) returned false. Therefore, it must be the case that no such $\mu_\#$ exists.

□