Checking a Multithreaded Algorithm with $^+$CAL

Leslie Lamport
Microsoft Research

11 Jul 2006

To appear in DISC 2006

Abstract

A colleague told me about a multithreaded algorithm that was later reported to have a bug. I rewrote the algorithm in the $^+$CAL algorithm language, ran the TLC model checker on it, and found the error. Programs are not released without being tested; why should algorithms be published without being model checked?

Contents

1 Introduction

On a Wednesday afternoon in March of this year, my colleague Yuan Yu told me that Detlefs et al. [3] had described a multithreaded algorithm to implement a shared two-sided queue using a double-compare-and-swap (DCAS) operation, and that Doherty et al. [4] later reported a bug in it. I decided to rewrite the algorithm in $^+$CAL [7] and check it with the TLC model checker, largely as a test of the $^+$CAL language. After working on it that afternoon, part of Sunday, and a couple of hours on Monday, I found the bug. This is the story of what I did. A $^+$CAL specification of the algorithm and an error trace that it produced are available on the Web [6]. I hope my experience will inspire computer scientists to model check their own algorithms before publishing them.

$^+$CAL is an algorithm language, not a programming language. It is expressive enough to provide a practical alternative to informal pseudo-code for writing high-level descriptions of algorithms. It cannot be compiled into efficient executable code, but an algorithm written in $^+$CAL can be translated into a TLA$^+$ specification that can be model checked or reasoned about with any desired degree of formality. Space does not permit me to describe the language and its enormous expressive power here. The two-sided queue algorithm is a low-level one, so its $^+$CAL version looks much like its description in an ordinary programming language. The features of $^+$CAL relevant to this example are explained here. A detailed description of the language along with the translator and model-checker software are on the Web [7].

1.1 The Story

I began by converting the algorithm from the C code of the paper into $^+$CAL as a collection of procedures, the way Detlefs et al. described it. (They actually extended C with an `atomically` statement to represent the DCAS operation, and they explained in the text what operations were considered atomic.) Other than minor syntax errors, the only bug in my first try was an incorrect modeling of the DCAS operation caused by my confusion about C's "`&`" and "`*`" operators. I found my errors by running TLC on small instances of the algorithm, and I quickly fixed them.

I next wrote a small test harness consisting of a collection of processes that nondeterministically called the procedures. It kept an upper-bound approximation to the multi-set of queued elements and checked to make sure that the element returned by a *pop* was in that multi-set. It also kept a lower bound on the number of queued elements and checked for a *pop* returning

"empty" when it shouldn't have. However, my test for an incorrect "empty" was wrong, and there was no simple way to fix it. So, I eliminated that test.

Running TLC on an instance of the algorithm with 2 enqueable values, 2 processes, and a heap of size 3 completed in 17 minutes, finding no bug. (Except where noted, execution times are for a 2.4 or 3 GHz personal computer.) The algorithm uses a fixed "dummy" node, so the maximum queue length is one less than the heap size. My next step was to check it on a larger model. I figured that checking with only a single enqueable value should suffice, because a *pop* that correctly removed an element was unlikely to return anything other than that element's correct value. I started running TLC on a model with 3 processes and 4 heap locations just before leaving for a three-day vacation. I returned to find that my computer had crashed after running TLC for two or three hours. I rebooted and restarted TLC from a checkpoint. A day later I saw that TLC had not yet found an error and its queue of unexamined states was still growing, so I stopped it.

I next decided to write a higher-level specification and let TLC check that the algorithm implemented this specification under a suitable refinement mapping [1] (often called an abstraction function). I also wrote a new version of the algorithm, without procedure calls, to reduce the size of the state space. This turned out to be unnecessary; TLC would easily have found the bug without that optimization.

I made a first guess at the refinement mapping based on the pictures in the Detlefs et al. paper showing how the implementation worked, but it was wrong. Correcting it would have required understanding the algorithm, and I didn't want to take the time to do that. Instead, I decided that an atomic change to the queue in the abstract specification was probably implemented by a successful DCAS operation. So, I added a dummy variable *queue* to the algorithm that is modified in the obvious way when the DCAS operation succeeds, and I wrote a simple refinement mapping in which the abstract specification's queue equaled *queue*. However, this refinement mapping didn't work right, producing spurious error reports on *pop* operations that return "empty".

A *pop* should be allowed to return "empty" if the abstract queue was empty at any time between the call to and return from the operation. I had to add another dummy variable to the algorithm to record if the queue had been empty between the call and return, and to modify the specification. Having added these dummy variables, I realized that I could check correctness by simply adding assertions to the algorithm; there was no need for a high-level specification and refinement mapping. I added the assertions, and TLC found the bug in about 20 seconds for an instance with 2 processes and

4 heap locations. (TLC would have found the bug with only 3 heap locations.) The bug was manifest by a *pop* returning "empty" with a non-empty queue—a type of error my first attempt couldn't detect.

After finding the error, I looked at the paper by Doherty et al. to check that I had found the same error they did. I discovered that I had found one of two errors they reported. I then removed the test that caught the first error and tried to find the second one. TLC quickly discovered the error on a model with 3 processes and 4 heap locations. As explained below, getting it to do this in a short time required a bit of cleverness. Using a naive, straightforward approach, it took TLC $1\frac{2}{3}$ days to find the error. Explicit-state model checking is well suited for parallel execution, and TLC can make use of shared-memory multiprocessing and networked computing. Run on a 384 processor Azul Systems computer [2], TLC found the error in less than an hour.

1.2 The Moral

I started knowing only that there was a bug in the algorithm. I knew nothing about the algorithm, and I had no idea what the bug was—except that Yuan Yu told me that he thought a safety property rather than a liveness property was violated. (I would have begun by looking for a safety violation anyway, since that is the most common form of error.) I still know essentially nothing about how the algorithm was supposed to work. I did not keep a record of exactly how much time I spent finding the error, but it probably totaled less than 10 hours. Had Detlefs et al. used $^+$CAL as they were developing their algorithm, model checking it would have taken very little extra time. They would certainly have found the first error and would probably have found the second.

There are two reasons I was able to find the first bug as quickly as I did, despite not understanding the algorithm. The obvious reason is that I was familiar with $^+$CAL and TLC. However, because this is a very low-level algorithm, originally written in simple C code, very little experience using $^+$CAL was needed. The most difficult part of $^+$CAL for most people is its very expressive mathematical expression language, which is needed only for describing more abstract, higher-level algorithms. The second reason is that the algorithm was expressed in precise code. Published concurrent algorithms are usually written in very informal pseudo-code, and it is often necessary to understand the algorithm from its description in the text in order to know what the pseudo-code is supposed to mean. In this case, the authors clearly stated what the algorithm did.

Section 2 describes the algorithm's translation from C to +CAL, and Section 3 describes how I checked it. The translation is quite straightforward. Had +CAL been available at the time, I expect Detlefs et al. would have had no trouble doing it themselves. However, they would have gotten more benefit by using +CAL from the start instead of C (or, more precisely, pseudo-C). Before devising the published algorithm, they most likely came up with other versions that they later found to be wrong. They probably would have discovered those errors much more quickly by running TLC on the +CAL code. Algorithms are often developed by trial and error, devising a plausible algorithm and checking if it works in various scenarios. TLC can do the checking for small instances much faster and more thoroughly than a person.

2 Translating from the C Version

A major criterion for the +CAL language was simplicity. The measure of a language's simplicity is how simple its formal semantics are. A +CAL algorithm is translated to a TLA$^+$ specification [8], which can then be checked by TLC. The TLA$^+$ translation defines the meaning of a +CAL algorithm. (Because TLA$^+$ is based on ordinary mathematics, its formal semantics are quite simple.) The simplicity of +CAL was achieved by making it easy to understand the correspondence between a +CAL algorithm and its translation. The translator itself, which is implemented in Java, is specified in TLA$^+$.

Simplicity dictated that +CAL eschew many common programming language concepts, like pointers, objects, and types. (Despite its lack of such constructs, and in part because it is untyped, +CAL is much more expressive than any programming language.) C's pointer operations are represented in the +CAL version using an explicit array (function) variable *Heap* indexed by (with domain) a set of addresses. A pointer-valued variable like *lh* in the C version becomes an address-valued variable, and the C expression `lh->L` is represented by the +CAL expression $Heap[lh].L$.

The only tricky part of translating from pointers to heap addresses came in the DCAS operation. Figure 1 contains the pseudo-C version. Such an operation is naturally defined in +CAL as a macro. A +CAL macro consists of a sequence of statements, not an expression. I therefore defined a DCAS macro with an additional first argument, so the +CAL statement

```
DCAS(result, a1, ...  )
```

represents the C statement

```
result = DCAS(a1, ...  )
```

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1 == old1) &&
        (*addr2 == old2)) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
    } else return false; } }
```

Figure 1: The DCAS operation in pseudo-C.

```
macro DCAS(result, addr1, addr2, old1, old2, new1, new2) {
  if ( (addr1 = old1) ∧
       (addr2 = old2)) {
    addr1 := new1 ||
    addr2 := new2 ;
    result := TRUE;
  } else result := FALSE; }
```

Figure 2: The DCAS operation in $^+$CAL.

The difficulty in writing the DCAS macro came from the pointer arguments *addr*1 and *addr*2. A direct translation of the original DCAS operation would have required an extra layer of complexity. A pointer-valued variable like *lh* would have had to be represented by a variable whose value was not a heap address, but rather the address of a memory location containing a heap address. However, this complication was unnecessary because, in all the algorithm's uses of the DCAS operation, the first two arguments are `&`-expressions. In the translation, I essentially defined the DCAS macro as if the "`*`"s were removed from the `*addr1` and `*addr2` parameters in Figure 1, and the "`&`"s were removed from the uses of the macro. This led me to the macro definition of Figure 2. (The "`||`" multiple-assignment construct is explained in Section 2.3 below; for now, consider it to be a semicolon.) The statement

```
if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy)) ...
```

is then represented in $^+$CAL as

```
DCAS(temp, LeftHat, RightHat, lh, rh, Dummy, Dummy) ;
if (temp) ...
```

The +CAL translator replaces the `DCAS` statement by the syntactic expansion of the `DCAS` macro. (As explained below, the `atomically` is implicit in the +CAL version.)

Most of my colleagues cannot immediately see that the result of the substitution is a correct translation of the C version. Since the expanded +CAL macro is quite simple, any difficulty must lie in understanding C's "`*`" and "`&`" operators. A language for describing algorithms should be simple, and its operators should be easy to understand.

As an illustration of the translation, Figure 3 shows the original C version of the *popLeft* procedure, exactly as presented by Detlefs et al., and my +CAL version. This was my original translation, before I added dummy variables for checking. (While it is essentially the same as my first version, I have made a number of cosmetic changes—mainly reformatting the code and changing label names to correspond to the line numbers of the corresponding control points in the C version.) The non-obvious aspects of the +CAL language that appear in this example are explained in Sections 2.1–2.3 below.

A cursory examination shows how similar the two versions are. I of course formatted the +CAL version to look as much as possible like the C version. (To this end, I used +CAL's more compact c-syntax rather the alternative p-syntax that is a bit easier to understand.) The +CAL version is three lines longer because of the extra line added in translating each DCAS operation and because of the local variable declarations that are missing from the pseudo-C code.

2.1 Labels and the Grain of Atomicity

Labels are used to specify the grain of atomicity in a +CAL algorithm. Execution of a single atomic step begins at a label and continues until the next label that is encountered. For example, execution of a step starting with control at label `O6`:

- ends at `O8` if $lh = rh$ evaluates to TRUE and the DCAS operation sets *temp* to TRUE.
- ends at `O2` if $lh = rh$ evaluates to TRUE and the DCAS operation sets *temp* to FALSE.
- ends at `O10` if $lh = rh$ evaluates to FALSE.

C Version

```
 1 val popLeft() {
 2   while (true) {
 3     lh = LeftHat;
 4     rh = RightHat;
 5     if (lh->L == lh) return "empty";
 6     if (lh == rh) {
 7       if (DCAS(&LeftHat, &RightHat, lh, rh, Dummy, Dummy))
 8         return lh->V;
 9     } else {
10       lhR = lh>R;
11       if (DCAS(&LeftHat, &lh>R, lh, lhR, lhR, lh)) {
12         result = lh->V;
13         lh->L = Dummy;
14         lh->V = null;
15         return result;
16 } } } }
```

+CAL Version

```
 procedure popLeft()
 variables rh, lh, lhR, temp, result; {
O2:  while (TRUE) {
       lh := LeftHat;
O4:    rh := RightHat;
O5:    if (Heap[lh].L = lh) {rVal[self] := "empty";  return};
O6:    if (lh = rh) {
         DCAS(temp, LeftHat, RightHat, lh, rh, Dummy, Dummy);
         if (temp) {
O8:        rVal[self] := Heap[lh].V; return}
       } else {
O10:     lhR := Heap[lh].R;
O11:     DCAS(temp, LeftHat, Heap[lh].R, lh, lhR, lhR, lh);
         if (temp) {
O12:       result := Heap[lh].V;
O13:       Heap[lh].L := Dummy ||
           Heap[lh].V := null;
           rVal[self] := result;  O15: return ;
} } } }
```

Figure 3: The C and +CAL versions of the *popLeft* procedure.

Because the DCAS macro contains no labels, its execution is atomic. (+CAL does not permit labels in a macro definition.)

To simplify model checking and reasoning about an algorithm, one wants to write it with the coarsest possible grain of atomicity that permits all relevant interleavings of actions from different processes. Detlefs et al. assume that a read or write of a single memory value is atomic. (Presumably, a heap address and an enqueued value each constitute a single memory value.)

I have adopted the standard method of using the coarsest grain of atomicity in which each atomic action contains only a single access to a shared data item. The shared variables relevant for this procedure are *Heap*, *LeftHat*, and *RightHat*. However, the labeling rules of +CAL required some additional labels. In particular, the label `O2` is required, even though the call of the procedure affects only the process's local state and could be made part of the same action as the evaluation of *LeftHat*.

2.2 Procedures

To maintain the simplicity of its translation to TLA$^+$, a +CAL procedure cannot return a value. Values are passed through global variables. In this algorithm, I have used the variable *rVal* to pass the value returned by a procedure. When executed by a process p, a procedure returns the value v by setting $rVal[p]$ to v. In +CAL code, `self` equals the name of the current process.

2.3 Multiple Assignment

One of +CAL's restrictions on labeling/atomicity, made to simplify the TLA$^+$ translation, is that a variable can be assigned a value by at most one statement during the execution of a single atomic step. A single multiple assignment statement can be used to set the values of several components of a single variable. A multiple assignment like

```
Heap[lh].L := Dummy || Heap[lh].V := null
```

is executed by first evaluating all the right-hand expressions, and then performing all the indicated assignments.

3 Checking the Algorithm

To check the correctness of the algorithm, I added two global history variables:

```
 procedure popLeft()
 variables rh, lh, lhR, temp, result; {
O2:  while (TRUE) {
       lh := LeftHat;
       sVal[self] := (queue = << >>);
O4:    rh := RightHat;
O5:    if (Heap[lh].L = lh) {
         assert sVal[self] ∨ (queue = ⟨⟩) ;
         rVal[self] := "empty";  return ;} ;
O6:    if (lh = rh) {
         DCAS(temp, LeftHat, RightHat, lh, rh, Dummy, Dummy);
         if (temp) {
           sVal[self] := Head(queue);
           queue := Tail(queue);
O8:        rVal[self] := Heap[lh].V;
           assert rVal[self] = sVal[self];
           return}
       } else {
O10:     lhR := Heap[lh].R;
O11:     DCAS(temp, LeftHat, Heap[lh].R, lh, lhR, lhR, lh);
         if (temp) {
           sVal[self] := Head(queue) ;
           queue := Tail(queue) ;
O12:       result := Heap[lh].V;
           assert result = sVal[self] ;
O13:       Heap[lh].L := Dummy ||
           Heap[lh].V := null;
           rVal[self] := result;  O15: return ;
} } } }
```

Figure 4: The *popLeft* procedure with checking.

- *queue*, whose value is the state of the abstract queue.
- $sVal$, where $sVal[p]$ is set by process p to remember certain information about the state for later use.

Adding dummy variables means rewriting the algorithm by adding statements that set the new variables but do not change the behavior if the values of those variables are ignored [1]. The $^+$CAL code for the *popLeft* procedure with the dummy variables appears in Figure 4. (The original code is in gray.)

The *queue* variable is modified in the obvious way by an atomic step that contains a successful DCAS operation (one that sets *temp* to TRUE). The

`assert` statements in steps `O8` and `O12` check that the value the procedure is about to return is the correct one.

The `assert` statement in step `O5` attempts to check that, when the procedure is about to return the value `"empty"`, it is permitted to do so. An `"empty"` return value is legal if the abstract queue was empty at some point between when the procedure was called and when it returns. The assertion actually checks that the queue was empty when operation `O2` was executed or is empty when the procedure is about to execute the return in operation `O5`. This test is pessimistic. The assertion would fail if operations of other processes made the queue empty and then non-empty again some time between the execution of those two operations, causing TLC incorrectly to report an error. For a correct test, each process would have to maintain a variable that is set by other processes when they remove the last item from the queue. However, the shared variables whose values determine if the procedure returns `"empty"` are read only by these two operations. Such a false alarm therefore seemed unlikely to me, and I decided to try this simpler test. (Knowing for sure would have required understanding the algorithm.) TLC returns a shortest-length path that contains an error. I therefore knew that, if the assertion could reveal an error, then TLC would produce a trace that showed the error rather than a longer trace in which another process happened to empty the queue at just the right time to make the execution correct. This assertion did find the bug—namely, a possible execution containing the following sequence of relevant events:

- Process 1 begins a *pushRight* operation.
- Process 1's *pushRight* operation completes successfully.
- Process 1 begins a *pushLeft* operation.
- Process 2 begins a *popLeft* operation
- Process 1's *pushLeft* operation completes successfully.
- Process 1 begins a *popRight* operation.
- Process 2's *popLeft* operation returns the value `"empty"`.

The actual execution trace, and the complete $^{+}$CAL algorithm that produced it, are available on the Web [6].

After finding the bug, I read the Doherty et al. paper and found that there was another error in the algorithm that caused it to pop the same element twice from the queue. I decided to see if TLC could find it, using

the version without procedures. The paper's description of the bug indicated that it could occur in a much coarser-grained version of the algorithm than I had been checking. (Since an execution of a coarse-grained algorithm represents a possible execution of a finer-grained version, an error in the coarse-grained version is an error in the original algorithm. Of course, the converse is not true.) To save model-checking time, I removed as many labels as I could without significantly changing the code, which was about 1/3 of the them. I then ran TLC on an increasing sequence of models, and in a few hours it found the error on a model with 3 processes and 4 heap locations, reporting an execution that described the following sequence of events:

- Process 1 begins a *pushRight* operation.
- Process 2 begins a *popRight* operation.
- Process 1's *pushRight* operation completes successfully.
- Process 1 begins a *popRight* operation.
- Process 3 begins and then successfully completes a *pushLeft* operation.
- Process 3 begins a *popLeft* operation.
- Process 1's *popRight* operation completes successfully.
- Process 1 begins and then successfully completes a *pushLeft* operation.
- Process 1 begins and then successfully completes a *popLeft* operation.
- Process 2's *popRight* operation completes successfully.
- Process 3's *popLeft* operation tries to remove an item from an empty queue.

I found the second bug quickly because I knew how to look for it. However, checking models on a coarser-grained version when the fine-grained version takes a long time is an obvious way of speeding up the search for bugs. It is not often done because, when written in most model-checking languages, changing an algorithm's grain of atomicity is not as easy as commenting out some labels. Someone who understands the algorithm will have a sense of how coarse a version is likely to reveal errors.

I decided to see how long it would take TLC to find the bug by checking the fine-grained version. I started it running shortly before leaving on a long

trip. When I returned, I found that it had indeed found the error—after running for a little more than a month. Examining the $^{+}$CAL code, I realized that it had two unnecessary labels. They caused some operations local to a process to be separate steps, increasing the number of reachable states. I removed those labels. I estimate that TLC would have found the error in the new version in about two weeks. However, by observing processor utilization, it was easy to see that TLC was spending most of its time doing disk I/O and was therefore memory-bound. I had been running it on a 2.4 GHz personal computer with 1 GByte of memory. I switched to a 3 GHz machine with 4 GBytes of memory, and TLC found the error in 40 hours. Running the model checker in the background for a couple of days is not a problem.

TLC can be instructed to use multiple processors. We have found that it can obtain a factor of n speedup by using n processors, for n up to at least 8. TLC can therefore take full advantage of the coming generation of multicore computers. (Inefficiencies of the Java runtimes currently available for personal computers significantly reduce the speedup obtained with those machines.) Using a version of TLC modified for execution with a large number of processors, the 40-hour uniprocessor execution was reduced to less than an hour on a first-generation 384-processor Azul Systems computer [2]. Since each of that computer's processors is much slower than 3 GHz, this probably represents a speedup by close to a factor of 328. (Azul Systems does not reveal the actual speed of their processors.) It is likely that, within a few years, computers will be widely available on which TLC runs 10 times faster than it does today.

There is an amusing footnote to this story. After doing the checking, I noticed that I had inadvertently omitted a label from the *pushRight* operation, letting one atomic action access two shared variables. I added the missing label and ran TLC on the slightly finer-grained algorithm, using the same personal computer as before. Because TLC does a breadth-first exploration of the state space, I knew that it would find a counterexample with one additional step. Indeed, it found exactly the same error trace, except with one of the original 46 steps split into two. However, instead of 40 hours, TLC took only 37.2 hours! It found the error after examining 148 million distinct states rather than 157 million. Figuring out how this could happen is a nice puzzle.

4 Conclusion

+CAL was not needed for this example. The algorithm could have been written in other languages with model checkers. Promela, the language of the Spin model checker, would probably have been a fine choice [5]. In fact, Doherty did use Spin to demonstrate the bug, although he wrote the Promela version expressly to find the bug he had already discovered [9]. However, most concurrent algorithms are not written as low-level pseudo-C programs. They are often written as higher-level algorithms, which are naturally described using mathematical concepts like quantification, sets, and sequences rather than the primitive operators provided by languages like C and Java. For such algorithms, +CAL is clearly superior to Promela and similar model-checking languages.

One can model check not only the algorithm, but also its proof. Because TLA$^+$ is based on mathematics, TLC is well suited to check an algorithm's proof. Rigorous proofs require invariance reasoning and may also involve showing that the algorithm implements a higher-level specification under a refinement mapping. TLC can check both invariance and implementation under a refinement mapping. Since they understood the algorithm, Detlefs et al. would have been able to define *queue* as a refinement mapping instead of adding it as a dummy variable the way I did. An error in an invariant or refinement mapping usually manifests itself before the algorithm does something wrong, allowing a model checker to find the problem sooner. Checking a refinement mapping might have revealed the two-sided queue algorithm's second error quickly, even on the fine-grained version.

Model checking is no substitute for proof. Most algorithms can be checked only on instances of an algorithm that are too small to give us complete confidence in their correctness. Moreover, a model checker does not explain why the algorithm works.

Conversely, a hand proof is no substitute for model checking. As the two-sided queue example shows, it is easy to make a mistake in an informal proof. Model checking can increase our confidence in an algorithm—even one that has been proved correct.

How much confidence model checking provides depends upon the algorithm. A simple, easy-to-use model checker like TLC can verify only particular instances—for example, 3 processes and 4 heap locations. The number of reachable states, and hence the time required for complete model checking, increases exponentially with the size of the model. Only fairly small instances can be checked. However, almost every error manifests itself on a very small instance—one that may or may not be too large to model check.

TLC can also check randomly generated executions on quite large instances. However, such checking can usually catch only simple errors.

My own experience indicates that model checking is unlikely to catch subtle errors in fault-tolerant algorithms that rely on replication and unbounded counters. It does much better on traditional synchronization algorithms like the two-sided queue implementation. However, even when it cannot rule out subtle errors, model checking is remarkably effective at catching simpler errors quickly. One can waste a lot of time trying to prove the correctness of an algorithm with a bug that, in retrospect, is obvious.

Model checking can be viewed as a sophisticated form of testing. Testing is not a substitute for good programming practice, but we don't release programs for others to use without testing them. For years, model checking has been a standard tool of hardware designers. Why is it seldom used by algorithm designers? With $^+$CAL, there is no longer the excuse that the language of model checkers is too low-level for describing algorithms. Model checking algorithms prior to submitting them for publication should become the norm.

Acknowledgment

I want to thank Homayoon Akhiani for model checking the two-sided queue algorithm with TLC on an Azul Systems multiprocessor.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Azul Systems. Web page. `http://www.azulsystems.com`.

[3] David L. Detlefs, Christine H. Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. Even better dcas-based concurrent deques. In Maurice Herlihy, editor, *Distributed Algorithms*, volume 1914 of *Lecture Notes in Computer Science*, pages 59–73, Toledo, Spain, October 2000. ACM.

[4] Simon Doherty, David L. Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul A. Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. Dcas is not a silver bullet for nonblocking algorithm design. In Phillip B. Gibbons and Micah Adler, editors, *SPAA 2004: Proceedings*

of the Sixteenth Annual ACM Symposium on Parallel Algorithms, pages 216–224, Barcelona, June 2004. ACM.

[5] Gerard J. Holzmann. *The Spin Model Checker*. Addison-Wesley, Boston, 2004.

[6] Leslie Lamport. An example of using +CAL to find a bug. URL `http://research.microsoft.com/users/lamport/tla/dcas-example.html`. The page can also be found by searching the Web for the 28-letter string formed by concatenating `uid` and `lamportdcaspluscalexample`.

[7] Leslie Lamport. The +CAL algorithm language. URL `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "-" from `uid-lamportpluscalhomepage`.

[8] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003.

[9] Mark Moir. Private communication. April 2006.