Goldilocks and the Three Specifications

John V. Guttag*

Massachusetts Institute of Technology
Cambridge, MA 02139 USA
Email: guttag@lcs.mit.edu

Abstract. A young girl enters Threads Forest. She becomes lost. She searches for enlightenment.

1 Prolog

Goldilocks' mother had often warned her not to enter Threads Forest. It was a strange and sometimes dangerous place. On the floor of the forest, where the sun never penetrated, grew toxic fungi. Dangerous beasts lurked everywhere.

There were many paths through the forest, and since they often crossed and frequently dead ended it was easy to get lost, even to starve. Furthermore from time to time old paths disappeared and new ones appeared. Wise men and women asserted that some paths would always be there, but were cryptic about which ones.

Despite her mother's repeated warnings, Goldi ventured into the forest. She soon became hopelessly lost. After giving her plight a moment's thought, she took the only rational course of action. "Help, get me out of here," she screamed.

Somewhat to her surprise, her plea was answered almost immediately. Three strangers slipped from behind three graceful deciduous conifers. "What seems to be the problem?" they asked in unison. "I can't find my way out of this stupid forest," Goldi replied. "Not to worry," said the shortest of the strangers. "You're in luck. It just so happens that we're cartographers, and we've each just completed a map of the forest."

With that, the cartographers each handed Goldi a map. Upon examining them, Goldi discovered (to her disgust) that they were all different. "Of course they are," exclaimed the cartographers. "One is too weak, one is too strong, and one is juuuuust right."

For some reason, that explanation didn't satisfy Goldi. Seeing her puzzlement, the smallest cartographer tried to explain. "All of the paths on the too

*Support for this research has been provided in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research Research under contract N00014-89-J-1988, and in part by the National Science Foundation under grant 9115797-CCR.

weak map exist at present and will continue to exist forever. However, there are many paths through the forest that don't appear on the map. The too strong map contains all of the useful paths through the forest at the present time. However, some of these paths may disappear in the future. The just right map contains all of the paths that will always be available."

Again, the explanation did not satisfy Goldi. "Why," she demanded, "do you insist on confusing me by giving me three maps?" The tallest cartographer, who was more patient than the others, made one more attempt at educating Goldi. "The too weak map is easier to follow than either of the others, and does contain some paths (albeit often longer than necessary) through the forest. The too strong map contains the shortest paths through the forest today. The problem is that you may not be able to use them tomorrow. The just right map, well that's juuuuust right."

Goldi was still puzzled. "Perhaps," she pleaded, a tear glistening in the corner of one eye, "you could give me formal specifications of all this?" "Ok," chortled the middle-sized cartographer, with a mocking glint in his eye, "you asked for it."

2 Introduction

In designing an interface, there is sometimes a tradeoff between making it easy to implement and making it easy to use. This tradeoff often centers, particularly in concurrent programs, around the amount of nondeterminism allowed. More nondeterminism leaves freedom for the implementor to choose a simpler or more efficient implementation. Less nondeterminism may support the development of simpler or more efficient client programs.

This paper presents three alternative formal specifications of part of a threads interface. The specifications presented here differ in the amount of nondeterminism allowed, and describe a hierarchy of implementations: the implementations satisfying the "too strong" specification are a strict subset of those satisfying the "just right" specification which, in turn, are a strict subset of those satisfying the "too weak" specification.

The specifications presented here are based upon work the author did in conjunction with Andrew Birrell, Jim Horning, Roy Levin and Garret Swart—all of the Digital Equipment Corporation Systems Research Center (SRC). In [1] we published what purported to be a formal specification of the threads synchronization primitives implemented as part of the Topaz operating system on the Firefly multi-processor. To the best of our knowledge, the implementation of Topaz indeed satisfied these specifications. However, the specifications were simultaneously too weak and too strong. The specifications were too weak in that they could not be used to justify some reasonable uses of the specified primitives in client programs, because the specifications permitted paths that could not actually arise in the implementation. The specifications were too strong in that they would not be satisfied by contemplated remote procedure call (RPC) implementations, because the specifications guaranteed the existence of paths that would disappear in those implementations.

The next section presents a short overview of Larch interface specifications. The section after that presents a short overview of what threads are about, lays out the issues involved in choosing the degree and types of non-determinism to be allowed, and presents formal specifications of three interesting alternatives in that space. The note concludes with a brief discussion of how the specifications were derived and the utility of the process and the specifications.

3 Larch Interface Specifications

Larch is a family of languages for writing formal specifications of interfaces in digital systems. The basic approach is described in [4].

The Larch family of languages supports a *two-tiered*, definitional style of specification. Each specification has components written in two languages: one language that is designed for a specific programming language and another language that is independent of any programming language. We call the former kind *Larch interface languages*, and the latter the *Larch Shared Language* (LSL).

Interface languages are used to specify the interfaces between program components. Each specification provides the information needed to use an interface. Each interface language deals with what can be observed by client programs written in a particular programming language. It provides a way to write assertions about program states, and it incorporates programming-language-specific notations for features such as side effects, exception handling, iterators, and concurrency.

Larch interface languages encourage a style of programming that emphasizes the use of abstractions, and each provides a mechanism for specifying abstract types. If its programming language provides direct support for abstract types, the interface language facility is modeled on that of the programming language; if its programming language does not, the facility is designed to be compatible with other aspects of the programming language.

An interface specification can describe exported types, constants, variables, and procedures. The specification of each procedure in an interface can be studied, understood, and used without reference to the specifications of other procedures. A specification consists of a procedure header (declaring the types of its arguments, results, and any global variables it may access) followed by a body of the form:

```
requires Predicate
modifies Target list
ensures  Predicate
```

A specification places constraints on both clients and implementations of the procedure. The *requires clause* is used to state restrictions on the state, including the values of any parameters, at the time of any call. The *modifies* and *ensures clauses* place constraints on the procedure's behavior when it is called properly. When specifying sequential programs, they relate two states, the state when

```
uses TaskQueue;
mutable type queue;
immutable type task;

task *getTask(queue q) {
  modifies q;
  ensures
    if isEmpty(q^)
      then result = NIL ∧ unchanged(q)
      else (*result)' = first(q^) ∧ q' = tail(q^);
}
```

Figure 1: An LCL interface specification

the procedure is called, the *pre-state*, and the state when it terminates, the *post-state*.

A modifies clause says what objects a procedure is allowed to change. It says that the procedure must not change the value of any objects visible to the client except for those in the target list. Any other object must have the same value in the pre and post-states. If there is no modifies clause, then no externally visible object can be changed.

Figure 1 contains a sample interface specification. The specification is written in LCL (a Larch interface language for C). This fragment introduces two abstract types and a procedure (function in C parlance) for selecting a task from a task queue. Briefly, * means pointer to (as in C), `result` refers to the value returned by the function, the symbol ^ is used to refer to the value of an object in the pre-state, and the symbol ' to refer to its value in the post-state.

The specification of `getTask` is not self-contained. For example, looking only at this specification there is no way to know which task `getTask` selects, because the meaning of the operators `first` and `tail` are not given. Is `first(q^)` the task that has been in q the longest? Is it is the one in q with the highest priority?

Interface specifications rely on definitions from *auxiliary specifications*, written in LSL, to provide semantics for the primitive terms they use. Specifiers are not limited to a fixed set of notations, but can use LSL to define specialized vocabularies suitable for particular interface specifications or classes of specifications. The `uses` clause in Figure 1 incorporates the LSL specification `TaskQueue` (not shown), which defines the operators `first` and `tail`.

The logical basis for Larch's treatment of concurrency is described in [1] and is similar to the one discussed in [7]. Our specifications deal only with safety, not with liveness.

Specifications of procedures for concurrent programs are similar to specifications of procedures for sequential programs. In both cases, the specifications prescribe the observable effects of procedures, without saying how they are to be achieved. In a sequential program, the states between a procedure call and its return cannot be observed in the calling environment. Thus one can specify a procedure by giving a predicate relating just the state when the procedure is called and the state when it returns [5]. Similarly, an *atomic action* in a concurrent program has no visible internal structure; its observable effects can also be specified by a predicate on just two states.

Any behavior of a concurrent system can be described as the execution of a sequence of atomic actions. In specifying atomic actions, we don't specify how atomicity is to be achieved, only that it must be. In an implementation, atomic actions may proceed concurrently as long as the concurrency isn't observable.

Atomic procedures execute just one atomic action per call. Each can be specified in terms of just two states: the state immediately preceding and the state immediately following the action. Note that, when dealing with concurrent programs, the pre-state of the action corresponding to a procedure body is not necessarily the state in which the procedure call was initiated; actions by other threads may have intervened. Similarly, the post-state of the action is not necessarily the state in which the caller resumes execution.

The observable effects of a *non-atomic procedure* cannot be described in terms of just two states. Its effects may span more states, and actions of other threads may be interleaved with its atomic actions. However, each execution of a non-atomic procedure can be viewed as a sequence of atomic actions. We specify a non-atomic procedure by giving a predicate that defines the allowable sequences of atomic actions (i.e., sequences of pre-post state pairs). Each execution of the procedure must be equivalent to such a sequence. Although it is sometimes necessary to specify constraints on the sequence as a whole, it often suffices to specify the atomic actions separately.

In addition to the constructs used to specify sequential interaces, we use the following constructs in specifying the procedures in the threads' interface:

- **when** clauses stating conditions that must be satisfied for atomic actions to take place. They place constraints not on the client but on the called procedure, which is obligated to make sure that its condition holds before taking any externally visible action. A **when** clause may thus impose a delay until actions of other threads make its predicate true. An omitted **when** clause is equivalent to **when** true, that is, no delay is required.

- **action** clauses specifying named actions. They are within the scope of the procedure header, and may refer to the procedure's formal parameters, results, and specification variables.

- **composition of** clauses indicating that any execution of a non-atomic procedure must be equivalent to execution of the named actions in the given order, possibly interleaved with actions of other threads.

- The reserved word **self**, standing for the identity of the thread executing the specified action.

```
mutable type mutex;

uses Mutex;

mutex mutex_create(void) {
  ensures fresh(result) ∧ result' = free;
}

void acquire(mutex m) {
  modifies m;
  when isFree(m^) ensures m' = grantMutex(self);
}

void release(mutex m) {
  requires holder(m^) = self;
  modifies m;
  ensures m' = free;
}
```

Figure 2: Specification of mutexes

4 The threads interface

The threads interface provides facilities for creating and controlling threads, which may or may not share memory. This note is concerned only with the facilities used for synchronizing threads.[1] Some of these are rather simple, and are derived from the concepts of *monitors* and *condition variables* described by Hoare[6]. Their semantics is similar to that provided by Mesa [8].

As far as clients of threads are concerned, all threads can execute concurrently. The threads implementation is responsible for assigning threads to real processors. The way in which this assignment is made affects performance, but does not affect the semantics of the synchronization primitives. The programmer can reason as if there were as many processors as threads.

Mutexes

A *mutex* [3] is the basic tool enabling threads to cooperate on access to shared variables. A mutex is normally used to achieve an effect similar to monitors, ensuring that a set of actions on a group of variables can be made *atomic* relative to any other thread's actions on these variables. A mutex is associated with the set of variables, and each critical section is bracketed by calls of the procedures `acquire` and `release`. The semantics of `acquire` and `release` ensure that these critical sections are indeed mutually exclusive.

There seems to be little controversy about the appropriate semantics for mutexes, so we give only one specification, Figure 2. The interface specification

[1]For more extensive descriptions of how threads are intended to be used and how they can be implemented see [1] and [2].

begins by asserting that `mutex` is a mutable abstract type. It then incorporates an LSL specification, `Mutex` that provides the operators `isFree`, `holder`, `grantMutex`, and `free` used in specifying the procedures exported by the interface. This LSL specification contains the axioms

```
holder(grantMutex(t)) = t
¬isFree(grantMutex(t))
isFree(free)
```

Condition Variables

Condition variables make it possible for a thread to suspend its execution while awaiting an action by some other thread. The normal paradigm for using condition variables is as follows. A condition variable `c` is associated with some shared variables protected by a mutex `m` and a predicate based on those shared variables. A thread acquires `m` (i.e., enters a critical section) and evaluates the predicate to see if it should call `wait(m, c)` to suspend its execution. This call atomically releases the mutex (i.e., ends the critical section) and suspends execution of that thread.

After any thread changes the shared variables so that `c`'s predicate might be satisfied, it calls `signal(c)` to awaken one thread or `broadcast(c)` to awaken all of them. `Signal` and `broadcast` allow blocked threads to resume execution and re-acquire the mutex. When a thread returns from `wait` it is in a new critical section. It re-evaluates the predicate and determines whether to proceed or to call `wait` again.

There are several subtleties in the semantics of these procedures, e.g., even if threads take care to call `signal` only when the predicate is true, the predicate may become false before a waiting thread resumes execution. These subtleties, and the utility of formal specifications in clarifying them, are discussed at some length in [1].

Figure 3 contains a specification of condition variables. It begins by importing the specification of the `mutex` interface. Next, type `condition` is specified to be a mutable abstract type. The `uses` clause incorporates an LSL specification of finite sets, and asserts that objects of type `condition` will be modeled in the specification as sets of objects of type `thread`. When a thread executes `wait(m, c)` it notionally inserts itself in the set `c` and waits for some other thread to remove it by calling `signal` or `broadcast`. We sat "notionally" because while it is convenient in the specification to model a condition variable as a set of threads, it need not be implemented that way.

Note that the specification of `broadcast` implies the specification of `signal`. `Broadcast` must unblock all threads waiting on the condition variable, whereas `signal` unblocks one or more threads (assuming that there are any waiting on the condition). This means that any implementation that satisfies `broadcast`'s specification will also satisfy `signal`'s. Of course, a good implementation of `signal` should strive to unblock exactly one thread. The difficulty in guaranteeing this stronger specification arises from the possibility of a race between calls

```
imports mutex;

mutable type condition;

uses Set(thread, condition);

condition condition_create(void) {
   ensures fresh(result) ∧ result' = { };
}

void signal(condition c) {
   modifies c;
   ensures if c^ = { } then c' = { } else c' ⊂ c^;
}

void broadcast(condition c) {
   modifies c;
   ensures c' = { };
}

void wait(mutex m, condition c) {
   = composition of Enqueue; Resume
   requires holder(m^) = self;
   modifies m, c;
   action Enqueue
     ensures m' = free ∧ c' = insert(self, c^);
   action Resume
     when isFree(m^) ∧ self ∉ c^
     ensures m' = grantMutex(self) ∧ c' = c^;
}
```

Figure 3: Specification of condition variables

to **signal** and **wait**. It is a limitation of the specification technique that we specify only what is guaranteed, and not what should happen most of the time.

Alerts

Alerts provide a polite form of interrupt. They are most often used to request termination of a long-running computation or a long-term wait.

The procedure call `alert(t)` is used to request that the thread `t` respond to an alert. A thread can check whether it has been alerted by calling `testAlert`. It is considered good practice for threads executing long-running computations to occasionally call `testAlert`. Similarly, threads executing waits that may last for a long time should block themselves by calling `alertWait` rather than `wait`. The function `alertWait` is similar to `wait`, except that

1. The wait can be terminated by an alert even if the condition on which the

```
imports condition;

typedef enum {alerted, signaled} alertStatus;

spec immutable type pendingAlerts;

uses Set(thread, pendingAlerts);

spec pendingAlerts pa = { };

void alert(thread t) pendingAlerts pa; {
   modifies pa;
   ensures pa' = insert(t, pa^);
}
```

Figure 4: Common part of specification of alerts

thread is waiting has not been signaled, and

2. `alertWait` returns a value indicating whether it is responding to a signal or to an alert.

A key issue in the design of a threads package with alerts is the amount of nondeterminism allowed in `testAlert`. If a thread executes `alert(t)` before the thread `t` executes `testAlert`, will `testAlert` necessarily return true? Clients would prefer an unqualified "yes." However, in a distributed system with remote procedure calls, `t` may migrate from node to node and `alert` may have to chase `t`. Guaranteeing that `testAlert` will always return true may be unacceptably inefficient, and clients may have to settle for "sometimes."

A similar question arises with `alertWait`. Things are straightforward when a blocked thread receives only a signal or an alert, but if a thread receives both, what is guaranteed about the outcome? Must it always return `signaled` if possible? Always return `alerted` if possible? Respond to whichever comes first? Or can the choice be nondeterministic?

The three specifications Goldilocks was given by the cartographers deal with these issues. Figure 4 contains material common to all three.

The specification uses a specification variable, `pa`, to keep track of pending alerts. Specification variables are declared solely to facilitate writing specifications. Like other global variables, they appear in the headers and bodies of specification. However, they are not exported by the interface, therefore client code cannot refer to them.

The weakest, and simplest, specification considered here, Figure 5, allows considerable nondeterminism. It allows `testAlert` to return true only if the thread has been alerted, but to return false any time. The specification allows `alertWait` to return `signaled` only if the thread has been removed from `c`, and to return `alerted` only if the thread has a pending alert. However, since the two `when` clauses of the `AlertResume` action are not mutually exclusive, the

```
bool testAlert(void) pendingAlerts pa; {
   modifies pa;
   ensures
     if result
       then self ∈ pa^  ∧ pa' = delete(self, pa^)
       else pa' = pa^;
}

alertStatus alertWait(mutex m, condition c) pendingAlerts pa; {
     = composition of Enqueue; AlertResume
   requires holder(m^) = self;
   modifies m, c, pa;
   action Enqueue ensures
     m' = free ∧ c' = insert(self, c^) ∧ pa' = pa^;
   action AlertResume
     when isFree(m^) ∧ self ∉ c^
       ensures result = signaled ∧ m' = grantMutex(self)
         ∧ c' = c^  ∧ pa' = pa^;
     when isFree(m^) ∧ self ∈ pa^
       ensures result = alerted ∧ m' = grantMutex(self)
         ∧ c' = delete(self, c^) ∧ pa' = delete(self, pa^);
```

Figure 5: Too weak specification

```
alertStatus alertWait(mutex m, condition c) pendingAlerts pa; {
   = composition of Enqueue; ChooseOutcome; GetMutex

  requires holder(m^) = self;
  modifies m, c, pa;

  spec bool signalChosen;

  action Enqueue
    ensures m' = free ∧ c' = insert(self, c^) ∧ pa' = pa^;
  action ChooseOutcome
    when self ∉ c^ ∨ self ∈ pa^
      ensures signalChosen' = self ∉ c^
        ∧ c' = delete(self, c^) ∧ m' = m^
        ∧ pa' = (if signalChosen' then pa^
                  else delete(self, pa^));
   action GetMutex
     when isFree(m^)
       ensures m' = grantMutex(self) ∧ c' = c^ ∧ pa' = pa^
         ∧ result = (if signalChosen then signaled else alerted);
}
```

Figure 6: Just right specification of `alertWait`

procedure can return either `signaled` or `alerted` when the thread has been alerted and the condition variable signaled. Therefore, this specification does not rule out the possibility that a signal will be lost by being delivered only to a thread that subsequently returns `alerted` rather than `signaled`. Since it fails to provide this guarantee, which is useful to clients without causing implementation problems, we label this specification "too weak."

The "just right" specification, Figure 6, provides a stronger guarantee for `alertWait`. It introduces a local specification variable, `signalChosen` and an extra action, `ChooseOutcome`. Here `alertWait` must return `signaled` when a signal comes first, but either outcome is allowed when an alert comes before a signal. This specification does not allow signals to be lost; clients of `signal` can rely on a waiting thread being unblocked, if there are any.

The "too strong" specification, Figure 7, strengthens `testAlert`'s specification by requiring that it return true if the thread has been alerted. Although this guarantee is appropriate for uniprocessors, it is unlikely to be valid in an RPC implementation in a distributed environment. The specification strengthens the "too weak" specification of `alertWait` by adding the conjunct `self ∈ c` to the second `when` clause of the action `AlertResume`. This ensures that signals are not lost. As it happens, however, this specification is "way too strong" in that we know of no efficient implementation that satisfies it.

```
bool testAlert(void) pendingAlerts pa; {
  modifies pa;
  ensures result = (self ∈ pa^) ∧ pa' = delete(self, pa^);
}

alertStatus alertWait(mutex m, condition c) pendingAlerts pa; {
     = composition of Enqueue; AlertResume
   requires holder(m^) = self;
   modifies m, c, pa;
   action Enqueue ensures
     m' = free ∧ c' = insert(self, c^) ∧ pa' = pa^;
   action AlertResume
     when isFree(m^) ∧ self ∉ c^
       ensures result = signaled ∧ m' = grantMutex(self)
         ∧ c' = c^ ∧ pa' = pa^;
     when isFree(m^) ∧ self ∈ pa^ ∧ self ∈ c^
       ensures result = alerted ∧ m' = grantMutex(self)
         ∧ c' = delete(self, c^) ∧ pa' = delete(self, pa^);
```

Figure 7: Too strong specification

5 Discussion

An important aspect of the work discussed here was the interplay among clients, implementors, and specifiers. We wrote a great many more specifications than the three discussed here. Some were rejected because they didn't ensure properties needed by clients, others because they promised more than the implementors knew how to provide efficiently.

Formalizability was never the bottleneck. Although the process of formalizing something we thought we understood often revealed a lack of precision in that understanding, we never found that we couldn't formalize something that we truly understood. Many times when we had seemingly reached agreement in oral discussion, the attempt to record that agreement in a specification revealed ambiguity or incompleteness. On those occasions the specifiers would come back to the clients or implementors with some precisely formulated alternatives and ask "Which do you want?"

We now have specifications that both clients and implementors have accepted. Does this mean that they are necessarily perfect, or even perfect for some purpose? No, but that isn't the issue.

In this exercise, we wanted to use formal specification as a tool to better understand a set of possible threads interfaces. The formal specifications helped us formulate precise questions about which design decisions were sensible, which were reflected in an existing implementation, and which were likely to persist in future implementations. Comparing alternative specifications helped us to explore a portion of the design space. Others may not agree with our definition of "just right" but at least we have clarified some of the choices.

6 Epilog

Shortly after parting from the cartographers, Goldi came upon a lone man, frantically chopping away at the forest with a dull machete. Upon spotting Goldi, he ceased his work and asked her what it was she held in her hand. Upon being told that they were specifications, he immediately seized and then ate them.

"What have you done?" protested Goldi, "Now I'll never find my way out of here." "Real hackers don't use specifications," thundered the man. With that, he handed Goldi a machete and a piece of paper. "You should be able to beat your way out on your own. However, if you decide to wimp out, you can always call the number on this paper, and ask to speak to the wizard who created this place."

Goldi, by now wise to the ways of the forest, wasted no time in dialing the number. "The number you have reached is no longer in service," the computer-generated voice informed her.

Acknowledgements

Most of the material in this paper is drawn from an earlier note co-authored with Jim Horning. I appreciate his graciousness in allowing me to use it here.

Steve Garland played a major role in designing the specification language used in this paper, and in adapting the original Modula-2 specifications to C.

Barbara Liskov, Sharon Perl, and Bill Weihl all provided useful and timely comments on an earlier draft of this paper. Jim Horning provided many useful comments on several versions of this paper.

References

[1] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levin. "Synchronization primitives for a multiprocessor: a formal specification." *Operating Systems Review* 21(5), Nov. 1987. Revised version in [9].

[2] Andrew Birrell, "An Introduction to Programming with Threads," Report 35, Digital Equipment Corporation Systems Research Center, Palo Alto, January 1989. Revised version in [9].

[3] Edsger W. Dijkstra, "The Structure of the 'THE'—Multiprogramming System," *Comm. ACM*, vol. 11, no. 5, 341–346, 1968.

[4] J.V. Guttag and J.J Horning, with S.J. Garland, K.D. Jones, A. Modet and J.M. Wing, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, New York, 1993.

[5] C. A. R. Hoare, "Procedures and Parameters: An Axiomatic Approach," Symposium on Semantics of Algorithmic Languages, Springer-Verlag, 1971.

[6] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Comm. ACM*, vol. 17, no. 10, 549–557, 1974.

[7] Leslie Lamport, "A Simple Approach To Specifying Concurrent Systems," Report 15, Digital Equipment Corporation, Systems Research Center, Palo Alto, 1986.

[8] B. W. Lampson and D. D. Redell, "Experiences with Processes and Monitors in Mesa," *Comm. ACM*, vol. 23, no. 2, 105–117.

[9] Greg Nelson (ed.). *Systems Programming with Modula-3*, Prentice Hall, 1991.