USENIX Association

Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference

Monterey, California, USA
June 10-15, 2002

X Meets Z: Verifying Correctness In The Presence Of POSIX Threads

Bart Massey
Computer Science Department
Portland State University
Portland, Oregon USA 97207–0751
bart@cs.pdx.edu

Robert Bauer
Rational Software Corporation
1920 Amberglen Pkwy, Suite 200
Beaverton, Oregon USA 97006
rbauer@rational.com

Abstract

The engineering of freely-available UNIX software normally utilizes an informal analysis and design process coupled with extensive user testing. While this approach is often appropriate, there are situations for which it produces less-than-stellar results.

A case study is given of such a situation that arose during the design and implementation of a thread-safe library for interaction with the X Window System. XCB is a C binding library for the X protocol designed to work transparently with either single-threaded or multi-threaded clients. Managing XCB client thread access to the X server while honoring the constraints of both XCB and the X server is thus a delicate matter.

The problem of ensuring that this complex system is coded correctly can be attacked through the use of lightweight formal methods. A model is constructed of the troublesome portion of the system using the Z formal specification notation. This model is used to establish important system properties and produce C code with a high likelihood of correctness.

1 Introduction

The design and implementation of multi-process and multi-threaded systems has historically been fraught with peril. This sort of code is commonly defective in subtle ways, leading to errors that are difficult to reproduce and troublesome to diagnose.

Each of the past several years, one of the authors has taught a course on formal methods for modeling and analysis of software systems using the Z (pronounced "Zehd") specification notation. This course has reviewed numerous successful case studies of "lightweight" formal methods using Z and related notations to analyze and formalize requirements and design and guide the development of resulting code. While this approach is not a panacea, it typically does dramatically reduce the likelihood of defects and increase confidence in the developed system. A more surprising finding of most of these case studies, confirmed by the authors' experience in teaching formal methods to industrial practitioners, is that Z-like lightweight formal methods are quite accessible to reasonably experienced software engineers. In fact, in many studies, the savings in time and effort due to reduced defect rates has exceeded the extra cost of judiciously-applied formal analysis.

Thus, using the Z notation to construct a lightweight problem model can enable the construction of an algorithm for controlling client thread access to the server that has a high likelihood of correctness in a situation where user testing is likely to be ineffective. The methods employed, while challenging, can be learned with a reasonable amount of effort by experienced programmers. This approach is thus a useful adjunct to more traditional methods of freely-available software construction.

2 Background

The X Window System [SGN88] has for some 15 years been the underlying basis common to most graphical user interfaces for UNIX systems. During that time, the server and wire protocol have been generally praised as well-designed. Indeed, the sample server implementation provided originally by the MIT X Consortium and now maintained and enhanced by the XFree86 team is widely regarded as the prime example of a policy-free networked graphics platform.

The client side software, however, has frequently been criticized for its somewhat excessive resource consumption, mediocre performance and reliability, and inflexibility in the face of changing application demands. Such criticisms seem especially well-justified in the case of

the new generation of handheld UNIX platforms that often come with a modest CPU and memory, are mission critical, and feature GUIs constrained by limited screen size and other factors. A number of recent X "toolkits", for example GTK+ [Pen99] and Qt [Dal01], have eschewed the original X GUI approach involving Xt [AS90] and the Athena or Motif [You90] widget set. This has led to a marked improvement along the critical client-side axes.

Underneath almost all these modernized toolkits, however, communication with the X server is still typically handled by Xlib [SGN88]. The Xlib C binding to the X protocol is in some ways a superior piece of code: its quality is attested to by its longstanding sole ownership of this critical task. However, Xlib suffers, to a lesser degree, to many of the problems that plagued Xt.

The XCB "X C Binding" [MS01] is an attempt to provide a radically shrunken and simplified X protocol translation library for C programs, while simultaneously extending flexibility in a few key ways demanded by modern toolkits and applications. Primary among these extensions is first-class support for multi-threaded library clients. The XCB interface attempts to provide a convenient latency-hiding interface to multi-threaded X clients, while cleanly and efficiently handling single-threaded clients using the same interface.

A principal difficulty in the design of the XCB internals was establishing the correctness of the core routines that handle the transfer of X requests to the server and server replies and events back to the client. The complication here is that these routines must interlock properly when being invoked by several threads, and must also complete successfully in the single-threaded case. Reasoning about multi-threaded code is always difficult. The IEEE POSIX 1003.1c-1995 Threads Specification (PTHREADS) [IEE95] provides details of the required semantics of the thread system itself (although this is difficult to verify, since the cost of the printed specification is prohibitive and it is not readily available online or through the local library). The trick is to avoid the perils of deadlock, races, non-determinism, and spinning that seem to be inherent in real multi-threaded situations.

The Z specification notation [Spi92] is a specific syntax and semantics for first-order logic with finite and integer sorts. It is carefully designed to avoid logical paradoxes, to be easy to use to describe realistic software engineering problems and solutions, and to be amenable to pencil-and-paper proofs of important properties. There are many excellent books available to introduce Z to software engineers (*e.g.*, [Jac97, Wor92]), and the existing knowledge typical of software engineers greatly assists learning Z.

During the construction of XCB, it became apparent that an important component, the XCB_Connection layer, is quite difficult to make correct while simultaneously accommodating the external design constraints placed upon it. Essentially, a "thread scheduling" algorithm with somewhat unusual requirements is needed. Four different designs for this algorithm were pseudo-coded, and two of these designs were actually implemented in C. Each of these four designs was subsequently shown, after a great deal of thought, to be subtly incorrect.

The task of specifying the thread scheduling algorithm using Z was begun with the hope that a reasonable amount of work could produce a design that was arguably, if not provably, correct. This hope has proven out: the Z model described here specifies such an algorithm.

It is somewhat unusual in the literature to present a Z model for a real system in its entirety, especially to an audience not necessarily comfortable with the Z notation. Nonetheless, it is hoped that this full presentation will provide a gentle introduction to Z, illustrate some interesting details of formal modeling on a real-world problem, and show how formal methods can be of assistance to developers of freely-available software for UNIX systems.

3 A Z XCB_Connection Model

A precise description of the XCB_Connection layer of XCB is needed in order to prove the desired properties. This layer of XCB handles calls from client threads (through the XCB_Protocol layer) to perform several tasks: enqueueing X server requests, retrieving X server replies, and retrieving X events (generated spontaneously by the server).

The principal engineering difficulties of the XCB_Connection layer are twofold. First, it is desirable for a single API to transparently handle both the single-threaded and multi-threaded case. Second, it is desirable that both of these cases be handled with minimum possible client latency: the layer should never gratuitously block a thread that could continue.

In this section, an engineering solution for the XCB_Connection layer is described. Semi-formal arguments about correctness of this solution are also offered.

3.1 The Z Notation

The style of the description of XCB_Connection presented is that standard for Z specifications: natural-language text interleaved with Z paragraphs. No prior

knowledge of Z is assumed here. Thus, in this section and the succeeding description an attempt will be made to explain the semantics of the various Z constructs used in the description. For Z novices, it may be desirable to consult one of the Z references cited in Section 2 before attempting to read this section.

A Z specification describes a state machine. States in the Z specification correspond to machine states of the program being modeled. State transitions in the specification correspond to execution of the program. In Z, state transitions are expressed using constraints between unprimed variables x indicating the state before the transition and primed variables x' indicating the state after the transition.

A Z state machine is described using *paragraphs* consisting of two optional parts. A declaration section lists the data objects comprising the state and their types. A constraint section indicates, in the form of equational first-order logic, constraints on the data items that must hold for the paragraph. The constraint portion of a Z paragraph limits the permissible values of variables named in the declaration portion of the paragraph. The declared type of a variable also limits its legal set of values. A horizontal bar separates the declaration section from the constraint section when both are present.

Z paragraphs come in several flavors. Basic type definitions define the names associated with types. In Z, a type is synonymous with the set of values that comprise it. Any set of values of a given type can itself be used as a type: thus new types can be constructed from old.

Axiomatic definitions are written with a bar to the left: they define the names and constraints associated with globally visible constant values. Schema definitions are written with a bar to the left, above, and below, forming a "schema box": they are typically named, and define names and constraints associated with a state or state transition. Since a schema definition denotes a range of possible values for a set of state variables, the name of the schema may itself be used as a type.

Z is not a programming language. As in mathematics, variables are simply names for values: the value associated with a name does not change over the course of a Z description. There is no inherent notion of execution of a Z description: the description defines a state machine whose transitions model the execution of a computer program, but this state machine is not necessarily directly executable.

Nonetheless, a Z description is useful in several ways: it can precisely specify the behavior of a computer program, it can expose logical defects in program design, and it can enable informal or rigorous proofs of desired design properties. Like other mathematical notations used in engineering, Z is a powerful tool for producing a high-quality product, rather than a substitute for the product itself.

3.2 Modeling System State

An item of principal interest in the description of the XCB system state is the set of client threads using the API. In Z, the existence of a set with no particular defining characteristics is declared by placing the set's name in square brackets: this is known as a "base type" declaration, since such a set is typically used as the set of values constituting some type. The Z here declares such a set, *THREAD*. The second line of the paragraph declares that *SEQNUM* is another name for $\mathbb{N}$, since the "sequence numbers" used by the X protocol are natural numbers.

$$[THREAD]$$
$$SEQNUM == \mathbb{N}$$

3.2.1 X Server State

The X server may be either ready to accept requests or not at any given time. At any given time, it may have a reply or an event response available. The sequence number of a reply is important to its correct processing. The Z here declares the existence of a set *REQ_STATUS* consisting of two distinct elements, given the names *req_ready* and *req_not_ready*. This is known as a "free type" declaration, since the set of values declared is typically used as a type. The declaration of *RESP_STATUS* is similar: in this case, the set consists of distinct elements named *no_resp* and *event*, and a set of distinct elements produced by applying the implicitly-defined "constructor" function *reply* to any legal *SEQNUM*.

$$REQ\_STATUS ::= req\_ready \mid req\_not\_ready$$
$$RESP\_STATUS ::= no\_resp \mid reply\langle\langle SEQNUM \rangle\rangle \mid event$$
$$REPLY == \mathrm{ran}\, reply$$

The X server state is described by a record containing the request and response status. The Z *schema* for this describes something analogous to a two-field record, where the *req_status* and *resp_status* fields, above the line in the Z "schema box", have the given types.

ServerState

$$req\_status : REQ\_STATUS$$
$$resp\_status : RESP\_STATUS$$

3.2.2 XCB Client State

The state of the XCB client application is much easier to describe. Its only interesting property, for the purposes of this description, is the set of threads that are blocked waiting for XCB. The possible sets of blocked threads are described as elements of the *power set* of threads (*i.e.*, the set of all possible sets of threads).

ThreadState

$$blocked : \mathbb{P}\, THREAD$$

3.2.3 XCB_Connection State

Having described the state of the external entities, it is necessary to describe the state of XCB_Connection itself. This state is a function of the client calls received, as well as the external states seen.

Important properties of an XCB client call, from the point of view of XCB_Connection, are the sequence number of interest and the thread making the call.

AskInfo

$$seqnum : SEQNUM$$
$$thread : THREAD$$

An XCB client call is referred to in this description as an "ask". There are three kinds of asks: read asks, write asks, and event read asks. Both read and write asks have a particular sequence number and thread associated with them. (The sequence number is assumed to be generated by the caller; this is not how the system actually works, but it is common in Z modeling to abstract away such details to simplify the treatment.) The sets of read asks and write asks will be used extensively as types, so abbreviations are created for them. Note that these subsets of *ASK* are simply the ranges of their constructor functions.

$$ASK ::= read\_ask\langle\langle AskInfo\rangle\rangle \mid write\_ask\langle\langle AskInfo\rangle\rangle \mid event\_ask$$

$$READ\_ASK == \mathrm{ran}\, read\_ask$$

$$WRITE\_ASK == \mathrm{ran}\, write\_ask$$

At any given time, there may be one thread that XCB_Connection designates as a "worker". The worker thread is blocked trying to process (send to or receive from the X server) the request or response associated with a particular sequence number. It is easiest to model this in Z by having the worker be an ASK, capturing just the data needed in the description.

$$WORKER ::= no\_worker \mid worker\_is\langle\langle ASK\rangle\rangle$$

- Reader queue, containing threads waiting to
  - Get an event.
  - Get a reply to an X request.
- Writer queue, containing threads waiting to write an X request.
- Event queue, containing events waiting to be delivered.
- Reply queue, containing replies waiting to be claimed.
- An output FIFO buffer.
- Indication as to whether there is currently
  - A current reader (thread blocked reading).
  - A current writer (thread blocked writing).

Figure 1: Relevant XCB Data Structures

The XCB state is fairly complicated. Figure 1 shows a sketch of the key items as abstracted from pseudo-code constructed during the development process. The state contains a sequence of asks enqueued as writers (a sequence since writes must occur in the order in which the asks are received by XCB), a set of asks enqueued as readers, and a set of replies waiting for a read ask. The queue of events waiting for an event read ask is modeled by a simple counter denoting its cardinality, since the content of events is not important to the description. Finally, the current workers blocked on reading and writing are maintained by the system.

XCBStateVars

$$writers : \mathrm{seq}\, WRITE\_ASK$$
$$readers : \mathbb{P}\, READ\_ASK$$
$$replies : \mathbb{P}\, REPLY$$
$$nevents : \mathbb{N}$$
$$cur\_reader, cur\_writer : WORKER$$

3.3 Modeling System State Change

The state changes in the XCB_Connection layer are complex. The pseudo-code algorithms of Figures 2 and 3 informally describe the intended state changes of the system. A "reader" is an XCB client thread attempting to retrieve an X server event or a reply to an X server request. (These cases are handled nearly identically by the implementation.) A "writer" is an XCB client thread attempting to send an X server request.

Note that an attempt was made to present the pseudo-code of Figures 2 and 3 as similarly as possible to its appearance in the design notes. This is necessary to ensure that the Z is developed to describe the desired algorithm, rather than driving some completely different algorithm design. It also gives some insight into the algorithm design style used by the XCB authors.

The complexity here is largely due to the restrictions the X protocol specification places on clients. X clients must never assume that the server connection will eventually become writable unless it is continuously being read from. This is because the X server can spontaneously generate responses on its output (X events) that will block the server until they are read. If the client is blocked waiting for the server to read its request, deadlock will result.

This restriction on the protocol is a difficult fit to one of the XCB requirements: that single-threaded and multi-threaded applications be able to use the same interface for sending requests. The single-threaded case by itself would be easy: a client thread wishing to send a request could use the `select()` system call to wait until the socket is writable, in which case it sends a request, or readable, in which case it processes whatever response has appeared and calls `select()` again. With an interface designed for multi-threaded applications, it would be possible to require the XCB client to always have a thread blocked reading events.

With a possibly multi-threaded client, the situation is more complex. The problem arises as follows: while one thread is blocked waiting for an X server response (selecting only for reading), another thread arrives hoping to send a request to the server. The second thread must block until the server connection is writable, but cannot guarantee that the reader will continue to read from the socket: the reader may complete its work and exit. This causes a danger of deadlock: if there is no thread consuming server responses, the server may block, which will block the writer. The solution is for writers to `select()` for both reading and writing, as in the single-threaded case, and handle the resulting race separately.

```
pending:
  Can satisfy call
   from pending queue?
    Yes:
      done
    No:
      Is there a reader or writer?
        Yes:
          enqueue self
          block
          go to pending
        No:
          become reader
select:
          select on read
          is there a writer?
            Yes:
              enqueue self
              block
              goto pending
          get response
          is it ours?
            Yes:
              wake up top of queue
               (writer else reader)
              done
            No:
              Is a client waiting?
                Yes:  deliver to it
                No:  place in pending
              Is a writer queued?
                Yes:
                  enqueue self
                  switch to writer
                  block
                  go to pending
                No:
                  go to select
```

Figure 2: XCB Reader Pseudo-code

Z is nice for modeling the XCB pseudo-code, because states are modeled as changing instantaneously: this avoids any explicit mention of mutual exclusion via mutexes at the model level. (Indeed, the presented pseudo-code does not mention them.)

A useful building block in constructing the Z descriptions will be a function that returns the reply in the reply queue matching a given ask, if any. The *OPT_REPLY* type is used to indicate whether there was a reply, and if so, what it was.

$$OPT\_REPLY ::= no\_reply \mid reply\_is\langle\langle REPLY \rangle\rangle$$

```
  Is there a writer?
    Yes:
      enqueue self
      block
  become writer
select:
  select on read ∨ write
    read:
        get reply
        Is there a waiting client?
          Yes:  deliver to client
          No:  place in reply or
            event queue
        go to select
    write:
      try to write
      was full amount?
        Yes:  done
        No:  go to select
```

Figure 3: XCB Writer Pseudo-code

The *lookup_reply* function looks up the reply matching the given *ASK* in the reply queue. Note the use of Z *schema inclusion* here: in the first line of the schema, all of the names declared in *XCBStateVars* are included in *XCBState* as well.

The definition of *lookup_reply* is given by constraining the result over every possible *AskInfo*. This style of implicit definition is normal in Z. The **if** – **then** – **else** is not an imperative programming-language construct: it is more like the `?:` operator of C, an operator that returns one of two different values depending on its leftmost argument.

XCBState

$$
\begin{array}{l}
XCBStateVars \\
lookup\_reply : READ\_ASK \rightarrow OPT\_REPLY \\
\hline
\forall r : AskInfo \bullet \\
\quad lookup\_reply\ (read\_ask\ r) = \\
\qquad \textbf{if} (reply\ r.seqnum) \in replies \\
\qquad\quad \textbf{then}\ reply\_is\ (reply\ r.seqnum) \\
\qquad\quad \textbf{else}\ no\_reply
\end{array}
$$

The constraint, below the line in the Z schema box, defines the *lookup_reply* function. This constraint is important in the proofs: it is implicit in any reference to XCBState in subsequent Z in the description. It is this feature of Z that makes it a useful pencil-and-paper proof notation: by expanding schemata (replacing their names with their definition) and manipulating the resulting constraints, one can construct proofs of interesting properties using first-order logic.

This completes the model of the XCB environment to the level of detail necessary for what follows. The formal notation here has already accomplished several tasks relative to the informal model of Figure 1. First, the formal model is quite precise: it tells exactly what is and is not of interest. Second, the formal model is fully strongly typed. This contributes to the precision (since, for example, sequences are typed differently than sets). It also reduces the likelihood of error: this Z, like most Z published these days, has been checked for correct syntax and type safety by an automated tool. As with computer programs, semantic defects in formal notation are often accompanied by syntactic or type errors.

3.3.1 Read Ask Model

Having described the environment and the state, the next step in Z modeling is to describe the possible state changes during execution. A Z model defines a (not necessarily deterministic) state machine. The initial state of the model is given explicitly. The remaining reachable states are implicitly defined by giving schema denoting legal state transitions. A state transition schema consists of unprimed values (e.g. x) of the state before the transition, and primed values (e.g. x') of the state after the transition.

The first three lines of the first state transition schema denote Z schema inclusions. The Δ operator applied to a schema inclusion denotes that the schema should be included twice: once with its names primed and once with them unprimed. Thus, all the definitions and constraints in *XCBState* implicitly appear in *ReadAskQueued* both with and without primed names. This ensures that any invariant constraints defined in *XCBState* hold both before and after the state transition. The Ξ operator includes primed and unprimed schema as *Delta* does. In addition the Ξ operator, that denotes that the referenced state will not change, implicitly defines an equality constraint between each primed/unprimed pair in the schema being operated on. The identifier *ask*? has been decorated with a question mark to indicate that it is an input to the state change, not part of any state.

The *ReadAskQueued* schema starts a read request for *ask*?, by checking whether the reply queue already contains the data being asked for. If so, the resulting reply queue is constrained (by the somewhat complex logic in the unique-existential constraint) to no longer contain the data, and the request is satisfied.

ReadAskQueued

$\Delta XCBState$
$\Xi ThreadState$
$\Xi ServerState$
$ask? : READ\_ASK$

$readers' = readers$
$writers' = writers$
$\exists_1 r : REPLY \bullet$
$\quad reply\_is\ r = lookup\_reply\ ask? \land$
$\quad replies' = replies \setminus \{r\}$
$nevents' = nevents$
$cur\_reader' = cur\_reader$
$cur\_writer' = cur\_writer$

What if the data is not available? Then the existential precondition (constraint on the before state) fails to hold. In this case, the entire state transition is unavailable: unless there is some other transition schema that can apply, the state is terminal.

In this case, one transition schema that could apply is one in which the ask blocks waiting for a reply. In the model threads block, not asks, so as a convenience a partial function is defined that returns the thread of a read or write ask. This function is specified by constraining its explicit *map*: the set of input-output pairs that constitute the function.

$ask\_thread : ASK \pfun THREAD$
$ask\_seqnum : ASK \pfun SEQNUM$

$ask\_thread =$
$\quad \{a : AskInfo \bullet (read\_ask\ a \mapsto a.thread)\} \cup$
$\quad \{a : AskInfo \bullet (write\_ask\ a \mapsto a.thread)\}$
$ask\_seqnum =$
$\quad \{a : AskInfo \bullet (read\_ask\ a \mapsto a.seqnum)\} \cup$
$\quad \{a : AskInfo \bullet (write\_ask\ a \mapsto a.seqnum)\}$

The preconditions of the following schema and the previous one are incompatible: the previous schema required that *lookup_reply* return a reply, whereas this one required that it return *no_reply*. Making the preconditions of all state transitions mutually exclusive makes the overall system deterministic. Another precondition of this schema is that the ask's thread cannot itself become the reader, because some other thread is already reading or writing.

ReadAskBlocks

$\Delta XCBState$
$\Delta ThreadState$
$\Xi ServerState$
$ask? : READ\_ASK$

$readers' = readers \cup \{ask?\}$
$writers' = writers$
$replies' = replies \land$
$no\_reply = lookup\_reply\ ask?$
$nevents' = nevents$
$cur\_reader \neq no\_worker \lor$
$cur\_writer \neq no\_worker$
$cur\_reader' = cur\_reader$
$cur\_writer' = cur\_writer$
$blocked' = blocked \cup \{ask\_thread\ ask?\}$

Finally, if the reader's data is not yet available and yet no other thread is already prepared to acquire it, this read ask becomes a worker on its own behalf and on behalf of those who come after it.

ReadAskWorker

$\Delta XCBState$
$\Delta ThreadState$
$\Xi ServerState$
$ask? : READ\_ASK$

$readers' = readers$
$writers' = writers$
$replies' = replies \land$
$no\_reply = lookup\_reply\ ask?$
$nevents' = nevents$
$cur\_reader = no\_worker \land$
$cur\_reader' = worker\_is\ ask?$
$cur\_writer' = cur\_writer = no\_worker$
$blocked' = blocked \cup \{ask\_thread\ ask?\}$

Putting all of this together, if XCB receives a read ask, it should take one of the three state transitions indicated above.

$ReadAsk == ReadAskQueued \lor$
$\quad ReadAskBlocks \lor ReadAskWorker$

(This shorthand notation for the *ReadAsk* schema indicates that at least one of the three referenced schema always applies: it is possible, though more cumbersome, to write the definition out in full schema notation.)

There are several properties of the specification that can be checked at this point, either informally or using a

formal proof. It is important to verify that the specification is well-founded: that all functions are applied only to their defined domains, for example, and that division by zero never occurs. This check is usually informal, although in this instance some progress was made toward a formal proof using the Z/EVES theorem prover [CMS99].

A well-founded specification may be deterministic or non-deterministic. As discussed earlier, this specification is deliberately deterministic. While non-deterministic specifications are sometimes useful, non-determinism is also commonly an inadvertent result of specification error.

Finally, a well-founded specification will be complete. In this case, it is necessary to show that for any possible combination of XCB states, thread states, and server states, a read ask will meet the precondition of at least one of the given schemata. A brief inspection shows this to be the case here.

3.3.2 Write Ask Model

The action taken by XCB on receiving a write ask is quite similar to that taken on receiving a read ask. One minor difference is that the writes must be enqueued in order: as mentioned earlier, this is the reason that the writers are stored in a sequence. Reads deliver specifically-requested data, and are completed in the order the data becomes available.

Another more important difference from the read case has to do with the situation where the X server is busy and thus writes cannot proceed. In this case, the writing thread cannot simply block until the server is available, since the server may also block waiting for its responses to be consumed, leading to deadlock. Instead, the writing thread must process X server responses while waiting.

Most confusingly of all, if a reader thread is processing X server responses when a write ask comes in and cannot proceed, the writing thread should take over the response processing task after the reader is done processing its next request. Note that there will be a period of time during which both the reader and writer will awaken if data becomes available from the X server. It is the details of this process that caused the most confusion in initial attempts to design this algorithm, and has led to the use of Z to clarify and validate the design.

First, the simple case: if a write ask comes in and there is already a writer, the asking thread blocks until the pending write completes. (In the actual implementation, the fact that there is a write queue and requests are written in blocks complicates the situation a bit. However, this level of detail is irrelevant to the purpose of modeling.) The blocked ask is used to construct a single-element sequence (surrounded by $\langle$ and $\rangle$) that is appended (using the sequence concatenation operator $\frown$) to the queue of pending write asks.

WriteAskBlocks

$$\Delta XCBState$$
$$\Delta ThreadState$$
$$\Xi ServerState$$
$$ask? : WRITE\_ASK$$

$$readers' = readers$$
$$writers' = writers \frown \langle ask? \rangle$$
$$replies' = replies$$
$$nevents' = nevents$$
$$cur\_writer \neq no\_worker$$
$$cur\_reader' = cur\_reader$$
$$cur\_writer' = cur\_writer$$
$$blocked' = blocked \cup \{ask\_thread\ ask?\}$$

If the write ask can proceed, the thread simply becomes the distinguished writer and blocks waiting to be able to write.

WriteAskWorker

$$\Delta XCBState$$
$$\Delta ThreadState$$
$$\Xi ServerState$$
$$ask? : WRITE\_ASK$$

$$readers' = readers$$
$$writers' = writers$$
$$replies' = replies$$
$$nevents' = nevents$$
$$cur\_writer = no\_worker \land cur\_writer' = worker\_is\ ask?$$
$$cur\_reader' = cur\_reader$$
$$blocked' = blocked \cup \{ask\_thread\ ask?\}$$

These schema are then assembled into a whole.

$$WriteAsk == WriteAskBlocks \lor WriteAskWorker$$

Again, questions of validity, determinism, and completeness arise, and again the answers are relatively straightforward.

3.3.3 Server Request Model

When the X server is capable of accepting a request and a writer thread is blocked waiting for this eventuality, the request can be immediately written to the server. At that point, the thread may return from XCB, but not before waking up a pending writer or reader. It is easiest to model this last bit first.

An auxiliary function will be useful for finding the thread associated with a worker. This example shows another popular style of function definition in Z: constraining a function's outputs over all valid inputs. Consider the universally-quantified constraint on *worker_ask*. For every object of type *WORKER* other than *no_worker*, this constraint ensures a valid value for the function. Thus, *worker_ask* is a partial function (denoted by the $\pfun$ symbol in the type).

$$
\begin{array}{l}
worker\_ask : WORKER \pfun ASK \\
worker\_thread : WORKER \pfun THREAD \\
worker\_seqnum : WORKER \pfun SEQNUM \\
\hline
\forall a : ASK \bullet worker\_ask\ (worker\_is\ a) = a \\
worker\_thread = ask\_thread \circ worker\_ask \\
worker\_seqnum = ask\_seqnum \circ worker\_ask
\end{array}
$$

The case where there are available writers is then modeled.

WakePendingWriter

$$
\begin{array}{l}
\Delta XCBState \\
\Xi ThreadState \\
\Xi ServerState \\
new\_worker! : WORKER \\
\hline
new\_worker! = worker\_is\ (head\ writers) \\
readers' = readers \\
writers \neq \langle\rangle \land \\
writers' = tail\ writers \\
replies' = replies \\
nevents' = nevents \\
cur\_writer = no\_worker \land \\
cur\_writer' = new\_worker! \\
cur\_reader' = cur\_reader
\end{array}
$$

For the current purposes, it is sufficient to nondeterministically select a new reader if there are pending readers but no writers. In practice, it is probably most efficient to select the reader whose reply is expected next.

WakePendingReader

$$
\begin{array}{l}
\Delta XCBState \\
\Xi ThreadState \\
\Xi ServerState \\
new\_worker! : WORKER \\
\hline
\exists r : readers \bullet new\_worker! = worker\_is\ r \\
readers' = readers \setminus \\
\quad \{worker\_ask\ new\_worker!\} \\
writers' = writers = \langle\rangle \\
replies' = replies \\
nevents' = nevents \\
cur\_writer' = cur\_writer \\
cur\_reader = no\_worker \land \\
cur\_reader' = new\_worker!
\end{array}
$$

Finally, it may be that there is nothing left to do. Z makes stating this case easy.

WakePendingNone

$$
\begin{array}{l}
\Xi XCBState \\
\Xi ThreadState \\
\Xi ServerState \\
new\_worker! : WORKER \\
\hline
readers = \varnothing \\
writers = \langle\rangle \\
new\_worker! = no\_worker
\end{array}
$$

Putting these together yields

$$WakePending == WakePendingWriter \lor WakePendingReader \lor WakePendingNone$$

Finally, when the X server is writable and there is a worker waiting to write to it, the worker is awakened, completes the write, replaces itself as necessary, and is done. The server may become blocked as a result of the write: this is left undetermined. The whole series of these "ServerDo" transitions has some common structure that can be conveniently captured with a subschema.

ServerWriteStuff

$$
\begin{array}{l}
\Delta XCBState \\
\Delta ThreadState \\
\Delta ServerState \\
\hline
replies' = replies \\
nevents' = nevents \\
writers' = writers \\
readers' = readers
\end{array}
$$

The subschema together with the manipulations on the state comprise the write process itself.

ServerDoWrite

$$
\begin{array}{|l}
\hline
ServerWriteStuff \\
\hline
cur\_writer \neq no\_worker \land \\
cur\_writer' = no\_worker \\
cur\_reader' = cur\_reader \\
req\_status = req\_ready \\
resp\_status' = resp\_status \\
blocked' = blocked \setminus \\
\{worker\_thread\ cur\_writer\} \\
\hline
\end{array}
$$

In the model, two state transitions happen sequentially but atomically: first the write, then the wakeups.

$$ServerRequest == ServerDoWrite \fatsemi WakePending$$

As before, checking validity and completeness is straightforward. With the exception of the deliberate nondeterminism in server state, the model is also straightforwardly deterministic.

3.3.4 Server Response Model

When the X server has response data to deliver to XCB, at least one of two conditions should hold. The response may be an event, in which case it may be queued at leisure for future examination. The case in which any sort of response is delivered while no thread is blocked waiting to read data is modeled by merely deferring the delivery until a thread is available: this approach requires no mechanism here. If an event is delivered while a thread is ready to read it, this is handled simply by delivering the event. Note that the server read is handled by leaving its new response indeterminate: while there are obviously constraints on what the server could return next, they are outside the scope of the model.

ServerReadEvent

$$
\begin{array}{|l}
\hline
ServerWriteStuff \\
\Xi ThreadState \\
\hline
cur\_writer \neq no\_worker \lor \\
cur\_reader \neq no\_worker \\
cur\_writer' = cur\_writer \\
cur\_reader' = cur\_reader \\
replies' = replies \\
nevents' = nevents + 1 \\
req\_status' = req\_status \\
resp\_status = event \\
\hline
\end{array}
$$

If the response is not an event, then there should be some thread blocked in XCB waiting to read the response and dispatch it appropriately. This thread will be either the reader or writer worker.

There is some complexity associated with handling the server response that has to do with the interaction between simultaneous reader and writer workers. It is assumed that all threads wake up from `select()` when a response is available. (This assumption seems natural, nothing in the PTHREADS specification seems to contradict it, and it seems to be the behavior under versions of Linux and Solaris tested by the authors.) No assumptions, however, are made about the order in which threads are awakened.

Thus, the read must be handled carefully: whichever worker wakes up first could perform the read, but there would be a danger of it being erroneously re-performed by the second worker. This race is handled by making any writer worker always perform the read. If there is just a reader, it performs the read: if the reply is destined for the blocked thread, it is returned.

ServerReaderThis

$$
\begin{array}{|l}
\hline
\Delta XCBState \\
\Delta ServerState \\
\Delta ThreadState \\
\hline
writers' = writers \land \\
readers' = readers \land \\
replies' = replies \land \\
nevents' = nevents \\
cur\_writer' = cur\_writer = no\_worker \\
\exists_1\ s : SEQNUM \bullet \\
\quad worker\_seqnum\ cur\_reader = s \land \\
\quad resp\_status = reply\ s \\
cur\_reader' = no\_worker \\
req\_status' = req\_status \\
blocked' = blocked \setminus \\
\{worker\_thread\ cur\_reader\} \\
\hline
\end{array}
$$

Otherwise, the reply may be dispatched if there is a reader blocked waiting for it, and any current reader worker continues to wait.

ServerReaderOther

$\Delta XCBState$
$\Delta ServerState$
$\Delta ThreadState$

$writers' = writers \land$
$replies' = replies \land$
$nevents' = nevents$

$cur\_writer' = cur\_writer = no\_worker$

$\forall s : SEQNUM \bullet$
  $worker\_seqnum\ cur\_reader \neq s \lor$
  $resp\_status \neq reply\ s$

$cur\_reader' = cur\_reader$

$req\_status' = req\_status$

$\exists a : READ\_ASK \bullet$
  $a \in readers \land$
  $resp\_status = reply\ (ask\_seqnum\ a) \land$
  $readers' = readers \setminus \{a\} \land$
  $blocked' = blocked \setminus$
    $\{ask\_thread\ a\}$

Finally, if there is no reader yet waiting for it, the reply is simply enqueued.

ServerReaderNone

$\Delta XCBState$
$\Delta ServerState$
$\Xi ThreadState$

$readers' = readers \land$
$writers' = writers \land$
$nevents' = nevents$

$cur\_writer' = cur\_writer = no\_worker$

$\forall s : SEQNUM \bullet$
  $worker\_seqnum\ cur\_reader \neq s \lor$
  $resp\_status \neq reply\ s$

$cur\_reader' = cur\_reader$

$req\_status' = req\_status$

$\forall a : READ\_ASK \bullet$
  $a \notin readers \lor$
  $resp\_status \neq reply\ (ask\_seqnum\ a)$

$resp\_status \in REPLY \land$
$replies' = replies \cup \{resp\_status\}$

If the writer worker is present and wakes up on a read, it can perform the read, process the result, and continue to select. (Actually, there is the potential for a race condition with both a reader and a writer present: if the writer exits before the reader awakens, the writer must first notify the reader that the read has occurred. This portion of the model is omitted here for simplicity: instead the not-unreasonable assumption is made that the reader will awaken before the writer exits, and these events are handled atomically by the schemata.)

It could be that the X server reply is destined for the current reader worker. In this case, the reply is simply marked as an *ask*! for future handling.

ServerWriterReadWorker

$\Delta XCBState$
$\Delta ServerState$
$\Xi ThreadState$
$ask! : READ\_ASK$

$writers' = writers \land$
$nevents' = nevents$

$cur\_writer' = cur\_writer \neq no\_worker$

$cur\_reader' = cur\_reader$

$cur\_reader = worker\_is\ ask!$

$readers' = readers \land$
$replies' = replies$

$\exists_1 s : SEQNUM \bullet$
  $ask\_seqnum\ ask! = s \land$
  $resp\_status = reply\ s$

$req\_status' = req\_status$

Alternatively, the reply may be destined for a blocked thread in the reader set. In this case, the blocked reader should be removed from the set and treated as a new ask.

ServerWriterReadBlocked

$\Delta XCBState$
$\Delta ServerState$
$\Delta ThreadState$
$ask! : READ\_ASK$

$writers' = writers \land$
$nevents' = nevents$

$cur\_writer' = cur\_writer \neq no\_worker$

$cur\_reader' = cur\_reader$

$ask! \in readers \land$
$readers' = readers \setminus \{ask!\} \land$

$replies' = replies \cup \{resp\_status\}$

$\exists_1 s : SEQNUM \bullet$
  $ask\_seqnum\ ask! = s \land$
  $resp\_status = reply\ s$

$req\_status' = req\_status$

$blocked' = blocked \setminus \{ask\_thread\ ask!\}$

In either of these cases, the writer worker will need to awaken a blocked reader worker. The generated output

ask! is used as an input to the *ReadAsk* schema previously defined (via the $\gg$ operator), effectively restarting the state machine at this point.

$$
\begin{array}{l}
ServerWriterDoRead == \\
\quad (ServerWriterReadWorker \lor \\
\quad ServerWriterReadBlocked) \gg \\
\quad ReadAsk
\end{array}
$$

Finally, the reply may just need to be enqueued.

ServerWriterQueueRead

$$
\begin{array}{l}
\Delta XCBState \\
\Delta ServerState \\
\Xi ThreadState \\
\hline
readers' = readers \land \\
writers' = writers \land \\
nevents' = nevents \\
cur\_writer' = cur\_writer \neq no\_worker \\
cur\_reader' = cur\_reader \\
\forall a : READ\_ASK \bullet \\
\quad resp\_status \neq reply(ask\_seqnum\ a) \lor \\
\quad cur\_reader \neq worker\_is\ a \lor \\
\quad a \notin readers \\
replies' = replies \cup \{resp\_status\} \\
req\_status' = req\_status
\end{array}
$$

When the server makes a read available with a writer present, it can be processed using one of the schemata just given.

$$
\begin{array}{l}
ServerWriterRead == \\
\quad ServerWriterDoRead \lor \\
\quad ServerWriterQueueRead
\end{array}
$$

Putting it all together yields a model for X server state changes that deliver a response to XCB.

$$
\begin{array}{l}
ServerResponse == \\
\quad ServerReadEvent \lor \\
\quad ServerReaderThis \lor \\
\quad ServerReaderOther \lor \\
\quad ServerReaderNone \lor \\
\quad ServerWriterRead
\end{array}
$$

3.4 The Full Model

The final model simply notes that the system state can change by any of the three transitions noted above: *ReadAsk* (Section 3.3.1), *WriteAsk* (Section 3.3.2), or server state change (Sections 3.3.3 and 3.3.4).

$$
\begin{array}{l}
ServerStateChange == \\
\quad ServerRequest \lor ServerResponse \\
XCBModel == \\
\quad ReadAsk \lor \\
\quad WriteAsk \lor \\
\quad ServerStateChange
\end{array}
$$

Of course, any state machine needs an initial state. This is specified with a specially-named state schema. Note that the initial server state is unspecified: this is in keeping with the (lack of) server model.

InitXCBModel

$$
\begin{array}{l}
XCBState \\
ServerState \\
ThreadState \\
\hline
cur\_reader = cur\_writer = no\_worker \\
readers = \varnothing \\
writers = \langle\rangle \\
replies = \varnothing \\
nevents = 0 \\
blocked = \varnothing
\end{array}
$$

The state machine defined by this Z model describes the desired behavior of XCB_Connection. In the process of describing it, some important progress has been made toward ensuring that the model is well-defined. The model also gives some important hints about how the implementation might be structured. The data structures it suggests are largely straightforward to implement in a conventional programming language. The few nondeterministic transitions of the specification can all be implemented deterministically without sacrificing correctness or performance. In short, the Z model accurately describes an algorithm that is likely to be both correct and reasonable to implement.

4 Analysis

There are a few important theorems that should be stated and proved about the modeled algorithm.

1. It should be shown that the algorithm is deadlock-free: that any thread that enters the system will eventually return from it. To show this, it is necessary to show that XCB will never try to write from the X server in a situation where it cannot read from it: this prevents a particularly pernicious form of deadlock that was actually present for a time in the

initial Xlib implementation [Get01] in which the X server is blocked trying to write to XCB and XCB is blocked trying to write to the X server.

2. It should be shown that that the algorithm meets the guarantee of XCB that threads will exit XCB "as soon as possible", that is, as soon as their exit conditions are satisfied. This condition is more difficult to state formally: since the statement involves asserting the existence of a sequence of state transitions, the condition cannot be stated directly in first-order logic.

The authors have used the formal model presented here to argue semi-formally that these properties hold: while space considerations preclude presentation of this proof sketch here, the formal model has been extremely helpful in this regard.

Of course, the formal model should correspond to the C code that implements it. One of the weaknesses of the Z notation is that this correspondence is not generally a mechanical one: that is, the C code is not generated by iterative refinement of the formal model as in, for example, the B Method [Abr96]. (The tradeoff here is that the B Method is substantially more difficult to learn and use.)

The C code in XCB that implements the Z model described here is the result of modifying earlier, flawed code to conform to the model. Less than 100 lines of C code are directly involved in implementing the model: this greatly simplifies the task of keeping the code straight. The XCB code implementing the model is freely available: the authors plan to comment the code with correspondences to the model, but the code as it stands is far from opaque in this regard.

To illustrate the sort of problem that the Z model helps to detect, it is useful to look at Figure 4, a defective section of pseudo-code developed prior to the Z modeling process. In this earlier design, there is only a single worker: writers block until reads complete, and themselves select only for writing. In this situation, the informal proof of property (1) above does not hold: when the worker is waiting to write to the X server, no thread is available to read from the X server, and there is a potential for deadlock. While this defect was discovered through informal reasoning about the system, initial attempts to repair it led to other designs with subtle flaws. This was a principal motivator for Z modeling effort: the resulting proof sketches give confidence that defects are being eliminated rather than just pushed around.

```
  ...
  become worker
select:
  select on write
    write:
      try to write
      was full amount?
        Yes:  done
        No:  go to select
```

Figure 4: Defective XCB Writer Pseudo-code

5 Evaluation

From the authors' point of view, the effort described in the previous section has been a successful one. The model, while challenging, has not been an unreasonable amount of work to develop. The transition from model to code is an easy one.

The confidence in the algorithm and implementation gained through the formal modeling process is important. The sorts of defects inherent in earlier incorrect versions of this algorithm would be difficult to isolate through even extensive user testing: these defects tend to be infrequent, hard to reproduce, and hard to understand through examination of the implementation. As a consequence, these defects are difficult to debug and repair: often, large pieces of software infrastructure must be torn down and rebuilt.

A number of industrial software engineers participating in the Oregon Master of Software Engineering (OMSE) Program (`http://www.omse.org`) have been taught the Z formal notation during a ten-week course in modeling and analysis of software systems. While these students are quite bright and hard-working, their success suggests that a working knowledge of formal methods is not impossible for those outside the ivory tower to acquire and appreciate. The sort of lightweight modeling described in the previous section, that isolates a troublesome portion of the system for more detailed study, is especially appropriate in this regard. OMSE students have often observed that the success of these methods for them is largely in the modeling stage: detailed formal proofs may be impossible for them, but they are also rarely necessary.

The world of software development, and especially freely-available software development, is changing. Demand for software continues to grow, and minimum acceptable quality levels are increasing: traditional freely-available software development methods may not be efficient enough to meet these countervailing challenges.

In vast commercial organizations, large amounts of structured unit, integration, and system testing effort can help to meet the quality demands. For individual project developers with limited resources, a more "back-of-the-envelope" approach may be suitable. In most engineering disciplines, use of often relatively unsophisticated mathematical methods has been a key to higher quality without substantial loss of productivity. Perhaps a similar result can be attained in the engineering of software.

Availability

The XCB implementation is freely available under an MIT-style license at `http://xcb.cs.pdx.edu`. The LaTeX source for this document, including the Z model, is also available at this location for interested researchers.

Acknowledgements

Thanks to Clem Cole for continuing to shepherd chronically-late authors through a tough paper. Thanks to Keith Packard for inspiring, assisting, evaluating, and suggesting solutions throughout the XCB development process. Thanks to Jamey Sharp, who implemented XCB and provided an incisive critique of the broken bits. Thanks to Jonathan Wistar for insightful comments on short notice.

References

[Abr96] Jean-Raymond Abrial. *The B Book: Assigning Programs To Meanings*. Cambridge University Press, 1996.

[AS90] Paul J. Asente and Ralph R. Swick. *X Window System Toolkit—The Complete Programmer's Guide and Specification*. Digital Press, Bedford, MA, 1990.

[CMS99] Dan Craigen, Irwin Meisels, and Mark Saaltink. Analysing Z specifications with Z/EVES. In J.P. Bowen and M.G. Hinchey, editors, *Industrial-Strength Formal Methods in Practice*. Springer-Verlag, 1999.

[Dal01] Matthias Kalle Dalheimer. *Programming with Qt*. O'Reilly & Associates, Inc., second edition, 2001.

[Get01] Jim Gettys, 2001. Personal communication.

[IEE95] IEEE 1003.1c-1995: Information technology—interface (POSIX) – system application program interface (API) amendment 2: Threads extension (C language), 1995.

[Jac97] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[MS01] Bart Massey and Jamey Sharp. XCB: An X protocol C binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[Pen99] Havoc Pennington. *GTK+/Gnome Application Development*. New Riders Publishing, 1999.

[SGN88] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System: C Library and Protocol Reference*. Digital Press, Bedford, MA, 1988.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Sciences. Prentice-Hall, London, second edition, 1992. Freely available online at `http://spivey.oriel.ox.ac.uk/~mike/zrm/zrm.ps.gz`.

[Wor92] J. B. Wordsworth. *Software Development with Z*. Addison-Wesley, 1992.

[You90] Douglas A. Young. *The X Window System - Programming and Applications with X (OSF-Motif Edition)*. Prentice Hall, Englewood Cliffs, NJ, 1990.