2010-12-09

Drop-in Concurrent API Replacement for Exploration, Test, and Debug

Everett Allen Morse
Brigham Young University - Provo

Drop-in Concurrent API Replacement for Exploration, Test, and Debug

Everett A. Morse

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric Mercer, Chair
Jay McCarthy
Scott Woodfield

Department of Computer Science

Brigham Young University

April 2011

ABSTRACT

Drop-in Concurrent API Replacement for Exploration, Test, and Debug

Everett A. Morse

Department of Computer Science

Master of Science

Complex concurrent APIs are difficult to reason about manually due to the exponential growth in the number of feasible schedules. Testing against reference solutions of these APIs is equally difficult as reference solutions implement an unknown set of allowed behaviors, and programmers have no way to directly control schedules or API internals to expose or reproduce errors. The work in this paper mechanically generates a drop-in replacement for a concurrent API from a formal specification. The specification is a guarded command system with first-order logic that is compiled into a core calculus. The term rewriting system is connected to actual C programs written against the API through lightweight wrappers in a role-based relationship with the rewriting system. The drop-in replacement supports putative what-if queries over API scenarios for behavior exploration, reproducibility for test and debug, full exhaustive search and other advanced model checking analysis methods for C programs using the API. We provide a Racket instantiation of the rewriting system with a C/Racket implementation of the role-based architecture and validate the process with an API from the Multicore Association.

Keywords: API, Concurrency, PLT Redex, Exhaustive Search, Continuations

ACKNOWLEDGMENTS

Thanks to Eric Mercer and Jay McCarthy as co-authors on the paper, as well as Nick Vrvilo for his work on our implementation. Furthermore, thanks go to Dr. Mercer for his hours of work and advice, and to Dr. McCarthy for excellent advice on and help with PLT Redex and the term rewriting definition of 4M.

Contents

List of Figures

List of Listings

Chapter 1

Preamble

Concurrent APIs are hard to reason about, due to an exponential explosion of possible linearizations for calls made in each thread. Short program examples can confuse experts (Palmer et al. [2007]). Tool support is necessary to tackle this problem.

Current approaches include reference implementations, such as the C implementation of the Multicore Association Communications API (MCAPI), and formal modeling techniques. Models have been created for subsets of several APIs, including Message-Passing Interface (MPI) and MCAPI (Palmer et al. [2007], Sharma et al. [2009]). These models are often limited in the behavior that they explore, especially those dependent on a reference implementation. Other models using formal logic are difficult to use for expressing API semantics and difficult to use for any user beyond the initial specifier. We propose a solution that provides a higher-level formal specification and that has wider use.

We allow API designers to experiment with design decisions, implementers to explore intended behavior of the API in order to test and debug their own implementations, and programmers to experiment with the API to better understand – and thus better use – the API. Our formal API specifications are closer to the natural language specification than some logic system and certainly closer than an implementation language like C, yet the are explicit in declaring internal state and all effects of operations, qualities that are purposely missing from the natural language specification. Furthermore, the specification can be used as a drop-in replacement for the API. It is thereby much easier to use than other formal methods and can move beyond the initial designers to allow exploration, debug, and testing by all users of the API.

This work presents a thesis in the form of a paper ready for journal publication. Chapters 2 through 8 are this paper, which has been submitted for publication. The appendices add further discussion of details that pertain to our thesis or that are too lengthy to include in the journal paper.

1.1 Relation to Thesis

The focus of the academic paper presented below is to introduce our architecture for utilizing a formal specification of an API as a drop-in replacement for that API for the purposes of exploration, test, and debug. In order to accomplish this, the paper presents 4M, a specification language for concurrent APIs that uses first-order logic. It presents the operational semantics of this specification language, then it presents the architecture for the drop-in. Finally, it concludes with our instantiation of this architecture as validation. The pieces of this paper are closely aligned with our original thesis.

Our thesis is that **a specification in first-order logic of an API can be used to mechanically generate an implementation that allows sound experimentation with API behavior**. The included paper presents this first-order logic API specification language. The drop-in architecture presented is the mechanically generated implementation. The purpose of the drop-in is to allow exploration, test, and debug – a purpose which includes experimentation with API behavior.

The process is mechanical in generating this implementation. It first uses a 4M compiler to produce the 4M core form which runs on the operational semantics presented in the paper. The instantiation of the architecture plugs in a PLT Redex (Felleisen et al. [2009]) model of the operational semantics along with other components in C and Racket to complete the drop-in architecture. It requires only a small amount of hand-tuning for the thin C wrappers in addition to the original 4M specification of the API; everything else is mechanically generated.

The generated implementation allows sound experimentation inasmuch as the user-made portions are correct. The drop-in architecture uses the 4M model of the API directly to generate all possible API behaviors. If the specification is correct, the model is also correct. The drop-in architecture deals with communication between the C program and the API model and with choosing behaviors to explore. This exploration method is API-agnostic and is either exhaustive, thus leaving no room for unexplored behaviors and thus unsound exploration, or it is a random walk of one path, which also soundly represents API experimentation. Each transformation in the compiler from full 4M to 4M core is simple and easily shown to preserve the meaning of the specified API. Furthermore, we have tested each portion of the drop-in instantiation to our satisfaction. If there are no errors in our instantiation, of which we are reasonably confident, then the exploration of API behavior is as sound as the 4M specification provided by the user.

The paper presents the operational semantics of 4M core in Chapter 4. The full 4M language is further explained in Appendix A, and the full rewrite rules for the operational semantics of 4M are presented all together in Appendix B. The compilation process from full 4M to 4M core is described in Appendix C. Next, the architecture of the drop-in is described in Chapter 5 along with the our implementation in Chapter 6. Usage examples for the 4M language and the drop-in are presented in Chapter 3. A much better validation, however, is the specification and exploration of a real API. This we have done with our 4M specification of MCAPI, presented in full detail in Appendix D.

Each aspect of the thesis is described herein, whether in the paper or in the appendix material. We thereby show that a specification in first-order logic of an API can be used to mechanically generate an implementation that allows sound experimentation with API behavior.

Chapter 2

Introduction

Natural language—"English"—specifications are the standard in concurrent API specifications. English is familiar and easy to produce for API designers, and they can choose the level of detail necessary. Debate about the meaning of the specification follows the time-honored standards of lawyers, literary critics, and philologists.

The vagueness often present in such specifications is valuable for implementors because implementation details are not micro-managed by designers. Instead, high-level properties such as "atomic", "blocking", or "non-overtaking" are aluded to, without requiring a detailed explanation of the API's internal state and how it provides these properties.

Implementations, of course, are only tangentially related to the specification. They are typically implemented in hardware or low-level languages, like C, where the high-level properties of the API are smeared across the low-level details of the circuit board. Implementers and users cannot easily determine the allowable behaviors of an API in such an implementation. Indeed, for many APIs there simply are no implementations because the API is in development.

There are clear benefits of formal logic-based specifications of APIs: designers can understand their creations; implementers can verify their work; and programmers can explore, test, and debug the behavior of their programs. For these benefits, designers and implementers can be viewed as special-cases of programmers: programs are particular API scenarios that designers may wish to understand and implementers may wish to use in order to validate their implementations.

Yet, few APIs are specified formally. We contend that this dearth is partly due to the tedious nature of formal specification as many languages are too low-level from a logical perspective (Palmer et al. [2007]), but more critically, because formal specifications do not serve a purpose beyond one time analysis in the constrained runtime environment of the specification framework—the formal specification rarely moves beyond the writer. When such specifications are available, it is difficult (or impossible) for a programmer, implementer, or designer to directly use these specifications for exploration, testing, and debugging of API scenarios in all but the most trivial instances, and certainly, using the actual specifications as instances of the API linked against application code is extraordinary.

The remainder of this paper presents our solution:

- a language, 4M, for concurrent API modeling and specification that combines the structure of English specifications with the detail of formal specifications (Chapter 4); and
- a novel technique for exploring, testing, and debugging actual C programs using a drop-in replacement of the API that employs a direct implementation of 4M as a term-rewriting system (Chapter 5) as an instance of the API.

Chapter 3 presents the structure of our solution using a concrete message-passing API. Chapter 6 validates our solution by applying it to MCAPI, a real communications API from the Multicore Association (MCA).

Chapter 3

Motivation and Example

Fig. 3.1(a) is the English description of a connectionless message passing API for multi-threaded applications. The specification defines four API functions to create mailboxes, get mailboxes, and then send and receive messages between mailboxes. The structure of the written English specification defines transitions with their input, effects, and error conditions, which are helpful properties in understanding individual API behavior in isolation.

As with any concurrent API, though, it can be a challenge to explain intended behavior in simple scenarios consisting of a handful of calls. Consider the scenario in Fig. 3.1(b) that includes three threads using the blocking send (`send`) and receive (`recv`) calls from the API to communicate with each other. Picking up the scenario just after the mailboxes are defined, thread 0 receives two messages from the mailbox `to0` in variables `a` and `b`; thread 1 receives one message from the mailbox `to1` in variable `c` and then sends the message "X" to the mailbox `to0`; and finally, thread 2 sends the messages "Y" and "Z" to the mailboxes `to0` and `to1` respectively. Given the scenario, an API designer, implementor, or programmer might ask the question, *"based on the written specification, which messages may be stored in variables `a`, `b`, and `c` at the end of program execution?"*

Intuitively, when the program terminates, variable `a` contains "Y" and variable `b` contains "X" since thread 2 must first send message "Y" to mailbox `to0` before it can send message "Z" to mailbox `to1`; consequently, thread 1 is then able to send message "X" to mailbox `to0`. Such intuition is correct in linearizing program execution, but it is not the only way to linearize as the specification allows an alternative scenario where message "Y" is delayed in transit and arrives at mailbox `to0` after message "X".

The English specification in Fig. 3.1(a) states that the send operation *"returns once the buffer can be reused by the application."* As such, the return of the send only implies a copy-out of the message buffer and not a delivery to the intended mailbox; thus, an additional way to linearize the program execution places the message "X" in variable `a` and the message "Y" in variable `b`.

The specification in Fig. 3.1(a) is a simplified subset of a real communications API from the MCA. Conversations with the MCAPI designers confirmed the intended behavior of the API to include both

`mbox_t mbox(int id, status_t *s)`
Description: creates a mailbox for `id`, returns its reference, and sets `*s` to 1. If `id` already exists, `*s` is set to -1 and the return has no meaning.

`mbox_t get_mbox(int id, status_t *s)`
Description: returns the reference for mailbox `id` and sets `*s` to 1. The call blocks if the mailbox has yet to be created.

`void send(mbox_t from, msg_t *msg, mbox_t to)`
Description: sends the message `msg` from the mailbox `from` to the mailbox `to`. It is a blocking function and returns once the buffer `msg` can be reused by another application.

`void recv(mbox_t to, msg_t *msg)`
Description: receives a message into `msg` from the mailbox `to`. It is a blocking function and returns once a message is available and the received data filled in `msg`. Messages from a common mailbox are non-overtaking.

(a)

Thread 0	**Thread 1**	**Thread 2**
`to0 = mbox(0,&s)`	`to1 = mbox(1,&s)`	`from2 = mbox(2,&s)`
	`to0 = get_mbox(0)`	`to0 = get_mbox(0)`
	`from1 = mbox(3, &s)`	`to1 = get_mbox(1)`
`recv(to0,&a)`	`recv(to1,&c)`	`send(from2,to0,"Y")`
`recv(to0,&b)`	`send(from1,to0,"X")`	`send(from2,to1,"Z")`

(b)

Figure 3.1: A simple message passing API with an example usage scenario. (a) A specification of the API. (b) A simple scenario to illustrate the difficulty in reasoning about API behavior.

linearizations of the scenario. Furthermore, lest the reader think this is a trivial issue, three published verification and analysis tools purpose-built for MCAPI omit the less intuitive program execution, and thus do not consider such allowed behavior in their results (Sharma et al. [2009], Elwakil and Yang [2010b,a]).

3.1 Formal Specification of the API

Without tool support it quickly becomes difficult to reason about behavior over concurrent API calls. What tools exist to support such putative what-if queries? Solutions can be roughly categorized into two groups: formal logic or direct implementation. A formal logic example is TLA, except that recent attempts to model MPI in TLA have shown the logic to be too low-level for practical application to APIs described in English prose (Lamport, Palmer et al. [2007]). A direct implementation example is to build the API directly in C. Not only is the gap between English prose and C extremely difficult to bridge, it is not possible to test in the presence of concurrency because a user cannot control execution schedules. Moreover, C is unusually susceptible to bugs as evidenced by the MCAPI reference implementation which non-deterministically deadlocks (The Multicore Association). To address the need for clear specification and modeling of concurrent APIs, we define 4M.

4M is a formal specification language designed to keep the best things from the informal written style and remove the worst. To be specific, 4M keeps the structure of the written English specification that defines transitions with their input, effects, and error conditions, but it replaces the thin veneer covering statements such as *"message non-overtaking"* in the written English with effects described in first-order logic over a predefined and explicitly listed vocabulary of API state. Furthermore, all internal processing implied by statements such as "*it returns once the buffer can be reused by another application*" is made explicit by defining daemon transitions that operate on internal API state that are concurrently enabled with pending API transitions.

The 4M description for the API used in the example scenario is given in Fig. 3.2. The vocabulary for the API state is defined in lines 1–4 comprising `mailboxes` to track defined end points, modeled as a set (indicated by the braces {}), and `queues`, initialized with the value 0, to track outstanding message sends in the form of a list of tuples. The API interface is defined as a series of transitions given in lines 5–42.

Consider the `mbox` transition defined on lines 5–18. It takes three parameters as input: a mailbox identifier `id` and references to result (`resultAddr`) and return status (`statusAddr`) to send information back to the caller. The transition itself is divided into two sections: `rule` (lines 7–13) and `errors` (lines 14–17). Each section contains a set of guarded transitions.

The 4M semantics first evaluate errors and then rules. Any enabled error may be selected, and its corresponding transition is taken. In the `mbox` transition, the guard on the error in line 15 uses existential quantification (`\E`) over the set `mailboxes` to determine if the request to create a mailbox duplicates an existing mailbox. The dot notation in `box.0` of the guard implies that `mailboxes` is a set of tuples, and `id` is the first member of the tuple. The effect of the error (indicated by the text following the `==>` on line 16) is to set the memory referenced by `statusAddr` in the next state to value `-1`. The '`@`' symbol is the dereference and the apostrophe indicates the next value

The rule section of `mbox` defines a single behavior on lines 8–12. This transition is always enabled in the absence of an error, and its effect is to (i) create an entry in the heap and set the reference to be `newAddr` using the `tmp` command (line 9); (ii) set the next value of memory referenced by `resultAddr` to be `newAddr` (the content of `resultAddr` is the return value from the transition); (iii) update the set `mailboxes` with the tuple `[id, newAddr]` using the union operator `\U` (line 11); and (iv) set the memory referenced by `statusAddr` in the next state to `1` to indicate the successful completion of the transition as per the API specification.

The other transitions get_mbox, send, and recv are defined similarly to mbox. Of note is the definition of recv, however, that blocks in the absence of a message. The guard on line 38 is only satisfied if the memory referenced by to is not empty, in which case it knows there is a pending message(s). Looking at the body of the rule, variable to references a list (lines 39–40) where the first member is the message with the content copied into msg and the second member is the list of remaining messages.

Internal API housekeeping is managed by daemon transitions as illustrated by the pump transition defined on lines 43–53. Daemon transitions are invoked infinitely often in the API, executed as often as the guards are enabled, and represent a concurrently enabled thread of execution to consider in any linearization. The pump daemon in the example API is active anytime queues has a non-zero value, and its role is to transfer messages from sending mailboxes to receiving mailboxes. It does this transfer by (i) defining a local variable from holding the first element of the queues tuple with the let expression (line 46); (ii) defining msg to hold the actual message from the sender (line 47); (iii) defining to to hold the address of the destination mailbox (line 48); (iv) adding the message to the receiver mailbox (line 49); (v) removing the message from the sender mailbox (line 50); and (vi) removing the pending send from queues (line 51).

3.2 Semantic Implementation of 4M

4M itself is intended for human consumption with form and semantics that are non-trivial to define directly. For example, 4M semantics give simultaneous update of all API state variables affected in a transition and allows calls to other transitions within an active transition. As such, it is possible to define a blocking send as a non-blocking send followed by a call to wait that blocks until the send completes. Such nuanced semantics are more easily realized by a core calculus upon which 4M is built.

The operational semantics for the 4M core is given by a term rewriting system employing small-step semantics through continuations. Fig. 3.3 shows the implementation of send, recv, and the daemon pump. Lines 1–2 define heap locations and initial values for the API state variables mailboxes (0) and queues (1). lines 3–6 and lines 7–11 implement send and recv. 4M core groups assignments into a simultaneous update action using the upd command. For the recv transition, the guard is on line 8, and the simultaneous update on msg and to is on lines 10–11. The core calculus is much more compact than 4M and more naturally implements the intended semantics.

Direct questions regarding API behavior over concurrent calls such as the scenario in Fig. 3.1(b) can be explored directly in 4M core by iteratively presenting to the calculus the current API call of each participating thread and asking the calculus for all possible next states of the system. In such a manner, it is possible to evolve the API state from a known initial state to one of several possible end states allowed by the specification.

The API state for the scenario at the point where threads 0 and 1 are blocking on their first calls to `recv`, and thread 2 is blocking on its second call to `send` is shown in Fig. 3.4. Lines 1–7 encode the heap with the `mailboxes` in location 0 and the `queues` in location 1. Location 7 in line 6 is the `from2` mailbox showing the message enqueued by `send(from2, to0, ‘‘Y’’)`. Following the heap are the states of the daemons and threads. Line 8 is the daemon for the `pump` transition. Lines 16–19 define the state of thread 2. Line 16–17 is the local environment mapping variable names to locations in the heap. Line 18 is the continuation in the tail position where `ret` marks the end of execution. Line 19 is the continuation showing the pending `send(from2, to1, ‘‘Z’’)` invocation.

The semantics allows several next states from Fig. 3.4 such as the `pump` transition moving the message out of the `from2` mailbox (location 7 in the heap) into the `to0` mailbox (location 5 in the heap) or adding the next send from thread 2 into the `queue` (location 1 in the heap) and `from2` mailbox. A user is able to trace any or all possible executions from the current API state by stepping each thread transition allowed by the semantics.

3.3 Drop-in Replacement for C Programs

Manually writing the state of the API for 4M core and manually stepping through the semantics definition is not feasible. Suppose instead that there exists an actual implementation of 4M core such that it is possible to reasonably argue that the implementation is a faithful reproduction of the term rewriting system. Naturally, it would be ideal to take a C program using the API, in form like the definition of thread 0 in Fig. 3.5(a), and connect it directly to the 4M core implementation to simulate the API behavior.

The connection is implemented by a role-based relationship between the C runtime and the 4M core implementation runtime. Thin wrappers bridge the API calls to the actual C drop-in code as shown in Fig. 3.5(b). The `gem_call` is the entry to the drop-in. The drop-in itself blocks waiting for all threads to invoke the API at which point it communicates with the 4M core implementation to send the state of the active threads. The 4M implementation returns a possible next state, and the drop-in releases the corresponding blocked `gem_call` for the stepped thread. The thread then continues until the next API entry occurs to repeat the process. The drop-in also stores a random seed from the execution for reproducibility in test and debug.

The drop-in can be further extended to exhaustive search when programs can rewind through some form of continuation. Fig. 3.5(c) shows the C code for thread 0 in a continuation passing style (CPS) semantics. The closure is completed by passing local variables from the t0 function as parameters to the continuations. These local variables are given as additional input to each API call. Note that the return value from mbox in line 4 must also be passed to the continuation in line 6. Each function ends with a call in the tail position. With CPS semantics, the drop-in queries the 4M implementation for all possible next states of the API and traverses the entire program behavior space.

3.4 Summary

The entire aforementioned process is implemented to mechanically create a drop-in replacement for a concurrent API described in 4M. The drop-in replacement is controllable for test and debug, and it can perform exhaustive search of the entire program behavior space with CPS semantics. The 4M core is implemented in Racket (Flatt and PLT [2010]) with additional support from the PLT Redex (Felleisen et al. [2009]) language for debugging and developing the core calculus.

The next three sections describe this process and our contributions in detail: Chapter 4 formally defines our language and provides an operational semantics for the 4M core rewriting system using small-step semantics. Chapter 5 presents the novel role-based architecture that bridges the C runtime to the implementation of the 4M core runtime, suitable for test, debug, and exhaustive search. Chapter 6 presents our 4M description of the connectionless message-passing MCAPI library with performance results.

```
state
  mailboxes = {}
  queues = 0
end
transition mbox
  input id, statusAddr, resultAddr
  rule
    true ==>
      tmp newAddr;
      @resultAddr' := newAddr;
      mailboxes' := mailboxes \U {[id, newAddr]};
      @statusAddr' := 1;
  end
  errors
    (\E box in mailboxes: box.0 = id) ==>
      @statusAddr' := -1;
  end
end
transition get_mbox
  input id, resultAddr
  rule
    (\E box in mailboxes: box.0 = id) ==>
      let mailbox = (box in mailboxes: box.0 = id);
      @resultAddr' := mailbox.1;
  end
end
transition send
  input from, msg, to
  rule
    true ==>
      queues' := [from, queues];
      @from' := [@msg, to, from];
  end
end
transition recv
  input to, msg
  rule
    @to != 0 ==>
      @msg' := (@to).0;
      @to' := (@to).1;
  end
end
daemon pump
  rule
    queues != 0 ==>
      let from = queues.0;
      let msg = (@from).0;
      let to = (@from).1;
      @to' := [msg, @to];
      @from' := (@from).2;
      queues' := queues.1;
  end
end
```

Figure 3.2: A simplified message-passing API in 4M

```
([0 0]
 [1 0])
(transition send (from val to)
 ([true
   ((upd [@ 1 (tuple from (@ 1))]
     [@ from (tuple val to (@ from))]))]))
(transition recv (to msg)
 ([(≠ 0 (@ to))
   ((upd ()
     ([@ msg (vecref (@ to) 0)]
      [@ to (vecref (@ to) 1)])))]))
(daemon pump ()
 ([(≠ 0 (@ 1))
   ((upd [@ 1 (vecref (@ 1) 1)]
     [@ (vecref (@ (vecref (@ 1) 0)) 1)
         (tuple
          (vecref (@ (vecref (@ 1) 0)) 0)
          (@ (vecref (@ (vecref (@ 1) 0)) 1)))]
      [@ (vecref (@ 1) 0)
         (vecref (@ (vecref (@ 1) 0)) 2)]))]))
```

Figure 3.3: The message passing API transitions **send**, **recv**, and **pump** as implemented in the core calculus that defines the semantics for 4M.

```
(((((((((∅ [0 ↦ ([0 (addr 5)] [1 (addr 6)]
                 [2 (addr 7)] [3 (addr 5)])])
          [1 ↦ ((addr 7) 0)]) [2 ↦ 0])
          [3 ↦ 0])            [4 ↦ 0])
          [5 ↦ 0])            [6 ↦ 0])
          [7 ↦ ("Y" (addr 5) (addr 7))]
          [8 ↦ 0]) [9 ↦ "Z"])
((∅ ret (ω pump))
 ((((∅ [a ↦ (addr 2)]) [b ↦ (addr 3)])
   [to0 ↦ (addr 5)])
  ret
  ((call recv (to0 a) ↦ ret)))
 ((((∅ [c ↦ (addr 4)]) [to1 ↦ (addr 6)]) [from1 ↦ 8])
  ret
  ((call recv (to1 c) ↦ ret)))
 (((((∅ [from2 ↦ 7])[to0 ↦ (addr 5)])
    [to1 ↦ (addr 6)]) [Z ↦ (addr 9)])
  ret
  ((call send (from2 to1 Z) ↦ ret)))))
```

Figure 3.4: The 4M core API state at the point where mailboxes are created and thread 2 completes its first send.

```
void t0() {
    msg_t a, b; status_t s; mbox_t to0;
    to0 = mbox(0, &s);
    recv(to0, &a);
    recv(to0, &b);
}
```

(a)

```
void send(mbox_t from,msg_t *msg,mbox_t to) {
    gem_call("send␣(%v␣%#b␣%v)",
             from, msg, to);
}
```

(b)

```
void t0() {
  msg_t a, b;
  status_t s;
  mbox(0, &s, &t0_1,{a:a,b:b});
}
void t0_1(msg_t a,msg_t b,mbox_t to0) {
  recv(to0, &a, &t0_2,{a:a,b:b,to0:to0});
}
void t0_2(msg_t a,msg_t b,mbox_t to0) {
  recv(to0, &b, &t0_3,{a:a,b:b,to0:to0});
}
void t0_3(msg_t a,msg_t b,mbox_t to0) {
  printf("a␣=␣%s\nb␣=␣%s\n", a, b);
}
```

(c)

Figure 3.5: An interface to connect the 4M core implementation to C programs. (a) The C implementation of thread 0 in the scenario. (b) The wrapper for the `send` API call. (c) A continuation passing style implementation of thread 0 in the scenario.

Chapter 4

4M

Many English specifications are simply a catalog of API calls, each with a description of effects on correct calls and a separate list of effects on incorrect calls—whether they be incorrect because of the state of the API internals or due to improper arguments. The assumed model of these specifications is a set of threads making API calls concurrently. The implementations of the API calls may coordinate with hidden state or with API "daemon" threads.

Model Our language, 4M, adopts this catalog specification structure and formalizes this correct/incorrect computation model, while adopting a few simplifications. Its model is a set of threads effecting a global store. User threads and daemon threads are explicitly represented. Each thread runs in atomic blocks that manipulate a single store state, thus representing a "transition" from one store state to another.

In any given 4M program, the threads cannot perform arbitrary transitions. Instead, they are restricted to a small set of transitions that are defined in a registry. This registry represents the particular API that the threads are using and is referred to as "the spec". This restriction on threads and, in turn, on state transitions, keeps the relationship between the 4M specification and the platonic specification tight.

The transition registry includes a list of well-known store locations that transitions can use to communicate and, for instance, implement queuing. It also singles out some transitions as "daemon" transitions that are assumed to always have dedicated threads executing them.

Structure 4M is specified as an operational abstract machine in the style of the CESK machine (Nielson and Nielson [1992]). For simplicity, it is a set of cooperating machines, each representing a different level of abstraction in the model.

The machines do not operate on the syntax exemplified by Figure 3.2. Instead, it is de-sugared into the core form on which the semantics of 4M is defined. Most of the transformation is obvious and intuitive, so we do not dwell on it. In the few places where the semantics of core 4M is tricky, we elaborate the translation in our discussion below.

The machines are as follows: a system machine for the set of all threads; a thread machine for a particular thread's state; an atomic block machine for the intermediate steps in the atomic regions; and an expression machine for the actual work of computation.

Each machine has slightly different capabilities. For example, the expression machine can inspect the store, but not modify it, and cannot inspect the transition registry. The atomic block machine may modify the store. The thread machine may inspect the registry.

Each machine has slightly different properties. For example, the expression machine is deterministic, while the others are not.

We present these machines from bottom to top: starting with the expression machine.

$$
\begin{aligned}
\mathit{estate} &::= (\sigma\ \eta\ e\ k) \\
\sigma &::= \emptyset \mid (\sigma\ \texttt{[}\ \mathit{address} \mapsto v\ \texttt{]}) \\
\eta &::= \emptyset \mid (\eta\ \texttt{[}\ \mathit{id} \mapsto v\ \texttt{]}) \\
\mathit{ae} &::= v \mid \mathit{id} \\
e &::= \mathit{ae} \mid (\texttt{@}\ e) \\
&\quad \mid (\mathit{setop}\ (\mathit{pattern}\ \texttt{in}\ e)\ e) \\
&\quad \mid (\texttt{if}\ e\ e\ e) \\
&\quad \mid (\texttt{let}\ (\texttt{[}\mathit{id}\ e\texttt{]})\ e) \\
&\quad \mid (\mathit{op}\ e\ e\ \ldots) \\
v &::= \textbf{number} \\
&\quad \mid \texttt{true} \mid \texttt{false} \\
&\quad \mid \texttt{error} \\
&\quad \mid \textbf{string} \\
&\quad \mid (\texttt{addr}\ \mathit{address}) \\
&\quad \mid (\texttt{const-set}\ v\ \ldots) \\
&\quad \mid (\texttt{const-tuple}\ v\ \ldots) \\
\mathit{pattern} &::= \mathit{id} \mid (\texttt{tuple}\ \mathit{id}\ \ldots) \\
\mathit{op} &::= \mathit{binop} \mid \mathit{unaop} \mid \texttt{set} \mid \texttt{tuple} \\
\mathit{setop} &::= \texttt{setFilter} \mid \texttt{setBuild} \\
k &::= \texttt{ret} \\
&\quad \mid (\texttt{@}\ \texttt{*}\ \texttt{->}\ k) \\
&\quad \mid (\mathit{setop}\ (\mathit{pattern}\ \texttt{in}\ \texttt{*})\ e\ \texttt{->}\ k) \\
&\quad \mid (\texttt{if}\ \texttt{*}\ e\ e\ \texttt{->}\ k) \\
&\quad \mid (\texttt{pop}\ \eta\ k) \\
&\quad \mid (\texttt{let}\ (\texttt{[}\mathit{id}\ \texttt{*]})\ e\ \texttt{->}\ k) \\
&\quad \mid (\mathit{op}\ (v\ \ldots)\texttt{*}\ (e\ \ldots)\texttt{->}\ k)
\end{aligned}
$$

Figure 4.1: Expression Machine syntax

Expression Machine The expression machine is a straight-forward CESK-style machine with a store (σ), environment (η), expression (e), and continuation (k).

The syntax is given in Figure 4.1. Stores map addresses to values; environments map identifiers to values; expressions are divided into atomic expressions (ae) (a category important in the other machines), store references (`@`), set operations (filtration and building), conditionals, binding, and operations. We do

Dereference cont

$$(\sigma\ \eta\ (@\ e)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e\ (@\ \texttt{*}\ \rightarrow k))$$

apply cont

$$(\sigma\ \eta\ (op\ e_0\ e_1\ ...)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_0\ (op\ ()\ (e_1\ ...) \rightarrow k))$$

argument eval

$$(\sigma\ \eta\ v_1\ (op\ (v_0\ ...)\ *\ (e_0\ e_1\ ...)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_0\ (op\ (v_0\ ...v_1)\ *\ (e_1\ ...) \rightarrow k))$$

pop eta

$$(\sigma\ \eta_1\ v\ (\texttt{pop}\ \eta_0\ k)) \rightarrow_e (\sigma\ \eta_0\ v\ k)$$

Variable lookup

$$(\sigma\ \eta\ id\ k) \rightarrow_e (\sigma\ \eta\ \eta(id)\ k)$$

Dereference

$$(\sigma\ \eta\ (\texttt{addr}\ address)\ (\texttt{@}\ \texttt{*}\ \rightarrow k)) \rightarrow_e (\sigma\ \eta\ \sigma(address)\ k)$$

Binary Op

$$\frac{v_{res} = v_{lhs}\ binop\ v_{rhs}}{(\sigma\ \eta\ v_{rhs}\ (binop\ (v_{lhs})\ \texttt{*}\ () \rightarrow k)) \rightarrow_e (\sigma\ \eta\ v_{res}\ k)}$$

If expression (true)

$$(\sigma\ \eta\ \texttt{true}\ (\texttt{if}\ ()\ \texttt{*}\ (e_{true}\ e_{false}) \rightarrow k)) \rightarrow_e (\sigma\ \eta\ e_{true}\ k)$$

Let expression

$$(\sigma\ \eta\ v\ (\texttt{let}\ ([id\ *])\ e \rightarrow k)) \rightarrow_e (\sigma\ (\eta\ [id\ \mapsto\ v])\ e\ (\texttt{pop}\ \eta\ k))$$

Figure 4.2: Expression Machine reductions ($\rightarrow_e$)

not specify the set of operations beyond set and tuple constructors. Continuations capture the structure of these expressions and include the top continuation (`ret`) and the environment restoring continuation (`pop`).

Figure 4.2 presents a few of the reductions. The structural rules are straight-forward except for application. The computational rules are equally simple, except for the set operations; though the details are tedious, so they are relegated to Appendix B. The meaning of an expression is defined as the transitive closure of $\rightarrow_e$ with the top continuation.

The expression machine is trivially lifted to lists of expressions with left-to-right evaluation:

Eval Leftmost

$$\frac{(\sigma\ \eta\ e_{target}\ \texttt{ret}) \rightarrow_e (\sigma\ \eta\ v_{target}\ \texttt{ret})}{(\sigma\ \eta\ (v\ ...)\ (e_{target}\ e\ ...)) \rightarrow_{es} (\sigma\ \eta\ (v\ ...\ v_{target})\ (e\ ...))}$$

$$\begin{aligned}
astate &::= (\sigma\ \eta\ (acmd\ ...)) \\
acmd &::= (\texttt{choose}\ id\ e) \\
&\quad\mid\ (\texttt{let}\ (\texttt{[}id\ e\texttt{]}\ ...)) \\
&\quad\mid\ (\texttt{alloc}\ id) \\
&\quad\mid\ (\texttt{upd}\ (\texttt{@}\ e\ e)...)
\end{aligned}$$

Figure 4.3: Atomic Block Machine syntax

Atomic Block Machine The atomic block machine performs all side-effecting operations of the language and is always executed atomically.

Choose

$$\frac{(\sigma\ \eta\ e\ \texttt{ret}) \rightarrow_e^* (\sigma\ \eta\ (\texttt{const-set}\ v_0\ ...\ v\ v_1\ ...)\ \texttt{ret})}{\begin{array}{c}(\sigma\ \eta\ ((\texttt{choose}\ id\ e)\ acmd\ ...)) \rightarrow_c \\ (\sigma\ (\eta[id\ \mapsto\ v]...)\ (acmd\ ...))\end{array}}$$

Let

$$\frac{(\sigma\ \eta\ ()\ (e\ ...)) \rightarrow_{es}^* (\sigma\ \eta\ (v\ ...)\ ())}{\begin{array}{c}(\sigma\ \eta\ ((\texttt{let}\ ([id\ e]\ ...))\ acmd\ ...)) \rightarrow_c \\ (\sigma\ (\eta[id\ \mapsto\ v]...)\ (acmd\ ...))\end{array}}$$

Alloc

$$\frac{address = \sigma_{malloc}(\sigma)}{\begin{array}{c}(\sigma\ \eta\ ((\texttt{alloc}\ id)\ acmd\ ...)) \rightarrow_c \\ ((\sigma\ [address\ \mapsto\ 0])\ (\eta\ [id\ \mapsto\ (\texttt{addr}\ address)])\ (acmd\ ...))\end{array}}$$

Store Update

$$\frac{\begin{array}{c}(\sigma\ \eta\ ()\ (e_{id}\ ...\ e\ ...)) \rightarrow_{es}^* (\sigma\ \eta\ ((\texttt{addr}\ address)\ ..._0\ v\ ..._0)\ ()) \\ \sigma' = \sigma[address\ \mapsto\ v]\ ...\end{array}}{(\sigma\ \eta\ ((\texttt{upd}\ (e_{id}\ e)\ ..._0)\ acmd\ ...)) \rightarrow_c (\sigma'\ \eta\ (acmd\ ...))}$$

Figure 4.4: Atomic Block Machine reductions ($\rightarrow_c$).

The syntax is given in Figure 4.3. The state consists of a store, an environment, and a sequence of commands. Commands in figure order are: non-deterministic choice from a set, simple binding, allocation, and store update. The rules are presented in Figure 4.4. Each rule is straight-forward, although our store update rule uses a non-standard annotation on "..." to indicate when lists should be the same length as another list. The atomic block machine relies on the expression machine for expression evaluation.

We could have made non-deterministic choice part of the expression machine. But, it would be dangerous for expression evaluation to be non-deterministic as we discuss below in the select command of the thread machine.

$$\begin{array}{rcl}
\mu & ::= & s\ (t\ |\ d)\ ... \\
s & ::= & \texttt{((}address\ v\texttt{)}...\texttt{)} \\
t & ::= & \texttt{(transition}\ id\ \texttt{(}id...\texttt{)(}r...\texttt{))} \\
d & ::= & \texttt{(daemon}\ id\ \texttt{()(}r...\texttt{))} \\
r & ::= & \texttt{(}e\ c\texttt{)} \\
c & ::= & \texttt{(}acmd\ ...\ tcmd\texttt{)} \\
tcmd & ::= & \texttt{(select}\ r\ ...\texttt{)} \\
 & | & \texttt{(call/k}\ id\ \texttt{(}ae\ ...\texttt{)}\ \ id\ \texttt{(}ae\ ...\texttt{))} \\
 & | & \texttt{(tail-call}\ id\ \texttt{(}ae\ ...\texttt{))} \\
 & | & \texttt{ret} \\
tstate & ::= & \texttt{(}\mu\ \sigma\ \eta\ tcmd\ ck\texttt{)} \\
ck & ::= & \texttt{ret} \\
 & | & \texttt{(}\omega\ id\texttt{)} \\
 & | & \texttt{(call}\ id\ \texttt{(}v\ ...\texttt{)}\ \texttt{->}\ ck\texttt{)}
\end{array}$$

Figure 4.5: Thread Machine syntax

Select

$$\frac{\begin{array}{l}(r_0\ ...\ r_1\ r_2\ ...) = (r\ ...) \qquad e_{pre} = \mathit{precondition}(r_1)\\ (\sigma\ \eta\ e_{pre}\ \texttt{ret}) \to_e^* (\sigma\ \eta\ \texttt{true}\ \texttt{ret}) \qquad (acmd\ ...\ tcmd) = \mathit{effects}(r_1) \qquad (\sigma\ \eta\ (acmd\ ...)) \dashrightarrow_c^* (\sigma'\ \eta'\ ())\end{array}}{(\mu\ \sigma\ \eta\ (\texttt{select}\ r\ ...)\ ck) \to_t (\mu\ \sigma'\ \eta'\ tcmd\ ck)}$$

Tail Call

$$\frac{\begin{array}{l}(id_x\ ...) = \mathit{arguments}(\mu, id)\\ (r\ ...) = \mathit{rules}(\mu, id) \qquad (\sigma\ \eta\ ()\ (ae\ ...)) \to_{es}^* (\sigma\ \eta\ (v\ ...)\ ()) \qquad \eta' = \emptyset[id_x\ \mapsto\ v]\ ...\end{array}}{(\mu\ \sigma\ \eta\ (\texttt{tail-call}\ id\ (ae...))\ ck) \to_t (\mu\ \sigma\ \eta'\ (\texttt{select}\ r\ ...)\ ck)}$$

Call with continuation

$$\frac{(\sigma\ \eta\ ()\ (ae_k\ ...)) \to_{es}^* (\sigma\ \eta\ (v_k\ ...)\ ())}{\begin{array}{l}(\mu\ \sigma\ \eta\ (\texttt{call/k}\ id_0\ (ae_0...)\ id_k\ (ae_k...))\ ck) \to_t\\ (\mu\ \sigma\ \eta\ (\texttt{tail-call}\ id_0\ (ae_0...)\ (\texttt{call}\ id_k\ (v_k...) \to ck))\end{array}}$$

Continuation call

$$\begin{array}{l}(\mu\ \sigma\ \eta\ \texttt{ret}\ (\texttt{call}\ id\ (v...) \to ck)) \to_t\\ (\mu\ \sigma\ \emptyset\ (\texttt{tail-call}\ id\ (v...))\ ck)\end{array}$$

Omega

$$(\mu\ \sigma\ \eta\ \texttt{ret}\ (\omega\ id)) \to_t (\mu\ \sigma\ \emptyset\ (\texttt{tail-call}\ id\ ())\ (\omega\ id))$$

Figure 4.6: Thread Machine reductions ($\to_t$)

Thread Machine The thread machine runs a sequence of API calls by looking up the appropriate transition in the transition registry and then using the atomic block machine to execute the transition block. Since it makes use of the transition registry, we specify it at this time. The syntax of both is given in Figure 4.5.

The transition registry (μ) consists of a specification of the initial store (s) followed by a sequence of normal (t) and daemon (d) transitions. Transitions have labels, input arguments, and a sequence of rules. Daemons are only distinguished by their lack of input arguments.

Rules (r) consist of an enabling condition expression and a command sequence. There is no distinction in transitions for error rules. Instead, the compiler from full 4M includes a negated-or of all error conditions in each non-error rule to ensure that they are never enabled when an error rule is enabled.

Command sequences are a list of commands that will execute atomically before a single thread command; they are effectively basic blocks with explicit control transitions at the end. This style is the outcome of the compiler from the full 4M language and is *not* present in that language.

Thread commands (*tcmd*) are either non-deterministic rule selection, branching to a transition with an explicit continuation, branching to a transition with no further action, or return.

The state of the thread machine is the read-only transition registry, the store, an environment, a single thread command, and a continuation stack. Continuation are either the top continuation, an infinitely called daemon transition (ω *id*), or call frame.

The reductions of the machine are shown in Figure 4.6, are quite subtle, and merit further discussion below.

Select non-deterministically chooses a rule, ensures that its precondition is enabled, then atomically applies its basic block and replaces the select command with the thread command of the chosen rule. (We use $\dashrightarrow_c^*$ to clarify that the outcome of this atomic block is also non-deterministic.) The evaluation of the atomic block may effect the store and can populate an environment with identifiers for use in the thread command.

The select rule shows why it is important that the expression machine is totally deterministic and non-deterministic choice is part of the atomic block machine: the effects of a rule depend on the pre-condition evaluating to true, but if the pre-condition *could have* evaluated to either true or false, then it would not be an invariant the effect could rely on. If the expression machine were non-deterministic, but we enforced that *every* execution of the pre-condition evaluated to true, then many common non-deterministic pre-conditions (such as there is some non-empty mailbox) would never be enabled, and thus not useful.

A tail-call looks up the parameters of the transition, evaluates the arguments with the expression machine, constructs a new environment binding them, and replaces the thread command with a non-deterministic selection of the transition's rules. Arguments are restricted to atomic expressions, which are only values or identifiers. This restriction ensures that they cannot depend on the store and thus are automatically schedule-independent.

A non-tail call evaluates the arguments of the *second* call (remember that arguments are atomic and pure) and pushes the frame onto the stack, then transforms the thread command into a tail-call.

A return to a call frame introduces a new tail-call command and pops the stack. A return to an ω frame introduces a new call to the daemon transition, but does not pop the stack, so the transition will be called infinitely.

$$\begin{aligned} \mathit{mstate} &::= \ (\mu\ \sigma\ (\mathit{thread}\ ...)\ (\mathit{thread}\ ...)) \\ \mathit{thread} &::= \ (\eta\ \mathit{tcmd}\ \mathit{ck}) \end{aligned}$$

Figure 4.7: System Machine syntax

System Machine The entire state of the system is managed by the system machine. Its syntax is specified in Figure 4.7. The state of the system machine consists of a transition registry, a store, and two lists of thread states. Each thread state has an environment, a thread command, and a thread continuation. One list of threads represents user threads and one list represents daemon threads.

The reduction rules for the system machine are given in Figure 4.8. When the system machine transitions, it may select any thread and run one transition of the thread machine for that thread, then reintegrate the new store and new thread state. Daemon threads are only selected if there is at least one

$$\textsc{System Step (non-daemon)}\quad \frac{(\mu\ \sigma\ \eta_1\ \mathit{tcmd_1}\ \mathit{ck_1}) \dashrightarrow_t (\mu\ \sigma'\ \eta_1'\ \mathit{tcmd_1}'\ \mathit{ck_1}')}{\begin{array}{c}(\mu\ \sigma\ (\mathit{thread_0}\ ...\ (\eta_1\ \mathit{tcmd}_1\ \mathit{ck}_1)\ \mathit{thread_2}\ ...)\ \mathit{threads}_d) \rightarrow_m \\ (\mu\ \sigma'\ (\mathit{thread_0}\ ...\ (\eta_1'\ \mathit{tcmd}_1'\ \mathit{ck}_1')\ \mathit{thread_2}\ ...)\ \mathit{threads}_d)\end{array}}$$

$$\textsc{System Step (daemon)}\quad \frac{\mathit{active}(\mathit{threads}_u) \qquad (\mu\ \sigma\ \eta_1\ \mathit{tcmd_1}\ \mathit{ck_1}) \dashrightarrow_t (\mu\ \sigma'\ \eta_1'\ \mathit{tcmd_1}'\ \mathit{ck_1}')}{\begin{array}{c}(\mu\ \sigma\ \mathit{threads}_u\ (\mathit{thread_0}\ ...\ (\eta_1\ \mathit{tcmd}_1\ \mathit{ck}_1)\ \mathit{thread_2}\ ...)) \rightarrow_m \\ (\mu\ \sigma'\ \mathit{threads}_u\ (\mathit{thread_0}\ ...\ (\eta_1'\ \mathit{tcmd}_1'\ \mathit{ck}_1')\ \mathit{thread_2}\ ...))\end{array}}$$

Figure 4.8: System Machine reductions ($\rightarrow_m$)

active user thread. This constraint ensures that the machine cannot compute forever with only daemon transitions.

Evaluation A 4M program is specified as a transition registry and a list of initial thread stacks. A program can be evaluated by constructing an initial system machine state. This state includes the transition registry; the initial store specified therein; a list of thread states where each has an empty environment, `ret` for the thread command, and the appropriate initial stack; and a daemon thread state for each daemon transition with (ω *id*) for the initial stack.

Summary The 4M semantics allows us to formally reason about the behavior of concurrent API scenarios. A system machine state can represent: the initial state of the API prior to the scenario's execution; any intermediate state that is not otherwise observable in normal implementations of the API; and all possible final states.

The real power comes from our ability to control the evolution of the API state by carefully selecting the reduction of interest from those reductions allowed by the system machine. For small scenarios, this is plausible by hand, but not enjoyable. For realistic scenarios, we require a more mechanized approach. The next section provides one possible architecture for deploying 4M.

Chapter 5

Drop-In Architecture

Our architecture for drop-in replacement of APIs in C programs is divided into four different components: an implementation of 4M, a mechanism for capturing and rewinding the state of the C program, a strategy for exploring the possible system states, and a technique for efficiently storing intermediate system states in a state database. These components are connected as follows.

As the C program runs, it makes API calls. Those calls transfer control to the drop-in. The drop-in waits for each thread to enter its control. At that point, it may capture the state of each thread for potential rewinding during exhaustive exploration. In any case, after each thread is waiting for the drop-in to return the result of each API call, the drop-in is free to resolve those calls in accord with the API specification.

The meaning of the API specification is brokered by an implementation of 4M. At any given point in the execution, a 4M system machine could be constructed to resolve allowable next states of the entire system. The drop-in does just this, by transforming the set of pending API calls and the states of the C threads into the form required by the 4M implementation. The implementation component computes these allowable next states.

The search strategy selects some next state to explore from the set of all available next states, including those generated by this transition and any previously unexplored states recorded in the state database.

After the search strategy has selected a next state, two types of adjustment must be made to the C program. First, the C program's stack and heap may need to be rewound to an earlier point if the search strategy is backtracking. Second, the changes in state computed by the 4M system machine are reified into the running C program. For example, if the global store is modified by API communication, that modification must be reflected in the actual C heap. Some state changes are not reflected, because they are internal to the API; however, they are preserved until the next state transition.

At this point, some C threads are resumed if the new state represents a completion of their API call. In practice, most new states only reflect the completion of a single API call, so only a single thread is resumed. Other threads remain in a quiescent state in the drop-in. The process begins again once the resumed threads return control to the drop-in.

In the following subsection we discuss the details for transferring control to the drop-in and reifying the 4M system machine state into the C runtime. Subsequently, we discuss instantiation options for the four components.

Transferring Control The C program calls into the drop-in through thin wrappers that replicate the interface of the API. The responsibility of the wrappers is to convert data types and parameters as needed, register memory shared by the C program and API, then communicate the call to the drop-in where the state capture component takes over. Once the next state has been computed and reified into the C program, the drop-in returns control to the wrappers. The wrappers then interpret data as necessary before returning to the caller.

Data conversion may be necessary where C datatypes do not match 4M core datatypes. For example, C distinguishes between integer and floating-point numbers while 4M does not. C also allows arrays of bytes, while 4M has only strings. The details are important, but trivial and tedious.

Reification of 4M State Some parameters to an API function may be pointers into C memory. These need to be registered with the drop-in to include them in the heap state copied between the C runtime and the 4M runtime. The wrappers also must translate between a C function with a return value and 4M transitions without return values. The wrapper function registers a pointer to a new local variable to use as a parameter to the 4M transition, from which the return value will be extracted and returned once the wrapper function regains control.

Implementing 4M An implementation of 4M must define an encoding of system machine states and a function from one state to a set of next states after one step of the system machine. In all likelihood, C—the language we assume the API is used from—is not appropriate for an obviously correct implementation of 4M. Our architecture allows this possibility by compartmentalizing the interaction between the C program and 4M into a single location in the drop-in. The drop-in may therefore call-out to an external implementation of 4M, through, e.g., pipe or network communication. The details of the encoding and the communication are opaque to the rest of the architecture.

Reification of C State The drop-in architecture relies on a reification of a C program's state to allow search strategies to backtrack. In this case, both the control state (stack) and the data state (heap) must be reified for correct resumption.

Reification of control state is a well-studied area where it is commonly called continuation capture. As such, a drop-in can apply any continuation capture method. In particular, conversion to continuation-passing style and direct duplication of the stack. This technology is implemented and readily available in many programming language implementations and user-level thread runtimes.

Reification of data state is more subtle. For programs that rely entirely on the API for data, the reification of their state is already present in the encoding of the 4M system machine state. For other data, using single-static assignment and lambda-lifting can turn data capture into continuation capture. Additionally, techniques for memory monitoring used in software transactional memory systems could be applied.

Finally, dynamic model checkers often address both these issues by re-executing the program from the beginning with a record of the branching decisions made in a previous run to restore an earlier state. These techniques are equally applicable in our architecture.

Search Strategy The simplest search strategy is random walk down a single concurrent execution with no memory of prior states. More involved search strategies are depth-first, breadth-first, or guided search. The more involved searches rely on the state database to memoize previously seen states and the mechanism to capture and rewind the C program state. Advanced search strategies may implement full linear temporal logic verification, partial order reduction, or other model checking analysis methods to verify properties or combat state explosion.

State Database The most direct state database is forgetful and only remembers the current state. A naïve state database stores everything in every state. More refined state databases employs full heap symmetry reduction, thread symmetry reduction, or other novel encoding techniques to maximize sharing. The state database may also be instantiated with hash compaction, bit-state hashing (i.e., super-trace or bloom filters), or other probabilistic techniques to track visited states. It is also possible to use an iterative process such as successive over/under approximation through predicate abstraction over the state vocabulary.

Summary This generic, role-based drop-in API architecture is flexible enough to support a variety of applications, including single-step execution, random walk single execution, replay for testing and debugging, full exhaustive search, as well as a myriad of sophisticated dynamic model-checking techniques, through the role specialization and instantiation.

As we discuss in the next section, we instantiated this architecture, applied it with a real API and real running C programs, and validated that it works not only in theory, but in practice as well.

Chapter 6

Instantiation and Validation

We instantiate our architecture (Fig. 6.1) with the components discussed below. Usage of our drop-in requires compilation of the API specification written in 4M (Fig. 6.1(a)) into 4M core and thin API wrappers, and linking the user program (Fig. 6.1(b)) with the drop-in.

Implementing 4M Our implementation of the 4M machine (Fig. 6.1(g)) in Racket (Flatt and PLT [2010]) uses pipes to communicate with the C drop-in runtime.

Our 4M implementation is written in PLT Redex (Felleisen et al. [2009]), a domain-specific language that ships with Racket for encoding operational semantics as rewriting systems. Since 4M is defined in exactly this way, the encoding is obvious. In fact, the figures we used to present 4M (e.g. Fig. 4.2) are automatically generated from the Redex sources. This correspondence increases our confidence that we have implemented 4M correctly.

This Redex model of 4M is used purely syntactically by the drop-in runtime: on each transition, the drop-in constructs the syntax-tree for the system state and the Redex rewriting engine returns the possible evolutions of the state after a single step of the system machine.

Our initial Redex implementation of 4M was too slow to be usable. For example, a single message send and receive with the MCAPI spec took about 12 minutes to find a single execution path. However, we were able to construct a new compiler for Redex and use it on our 4M implementation. This new compiler ran the same test in less than half a second. In fact, the time to start the process and load the bytecode was longer than the computation of the execution path.

Control Transfer and Data Reification Since we use a pipe to communicate between C and Racket runtimes, the data sent along this pipe must be encoded. A message is formatted per transition invocation. Most data is converted into a string representation.

Fig. 6.2 is an example of a wrapper (Fig. 6.1(c)). It demonstrates data conversion and passing control from the wrappers to the drop-in by use of the function `gem_call` on line 7–8. The wrapper indicates the transition to call, what the parameters are, and what the data formats should be per parameter. Line

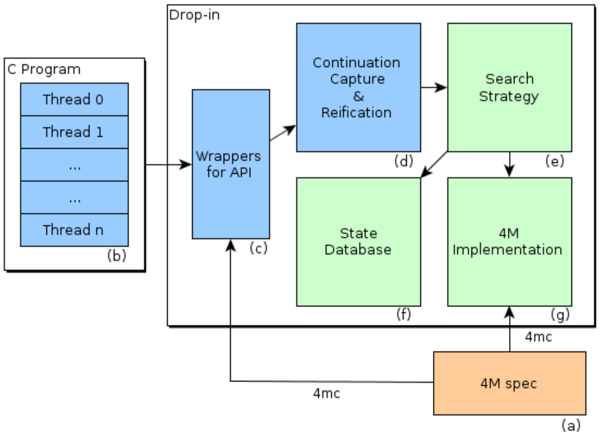

Figure 6.1: The Drop-In architecture. Blue boxes are in the C runtime while green boxes are in the Racket runtime.

```
1  mbox_t mbox(int id, status_t* status) {
2      char result[24] = {0};
3      char status_str[8] = {0};
4      int result_addr = reg_var(result, 24);
5      int status_addr = reg_var(status_str, 8);
6
7      gem_call("make_mailbox␣(%d␣%v␣%v)",
8              id, status_addr, result_addr);
9
10     *status = atoi(status_str);
11     return atoi(result);
12 }
```

Figure 6.2: An example wrapper for the `mbox` function.

10 is an example of reification to convert data types back, in this case, from a string representation of a number into a C integer. Lines 4 and 11 show the creation and registration of a local variable to contain the return result, then returning the contents of this temporary variable.

Reification of C State Our implementation uses an explicit conversion to continuation-passing style to capture the control state (Fig. 6.1(d)) and assumes that the API state fully reflects the relevant data state. When this is not true, we assume an appropriate transformation has occurred to reflect all data in the environment of the continuation.

Search Strategy The implementation has two possible search strategies (Fig. 6.1(e)): random-walk and bounded depth-first search. Random walk explores a single execution and reports the corresponding random seed that produced the execution. The depth-first search, as expected, runs to the ascribed bound (or termination) before backtracking to the most recent decision point. The search is randomized to avoid pitfalls in default-search order with the random seed reported at the end of execution. The depth-first search also stores its own backtrack points and only uses the state database to detect duplicate states rather than to retrieve full states for backtracking.

State Database The state database (Fig. 6.1(f)) employs bit-state hashing on a single bit to only detect duplicate states based on computed key values (Holzmann [1998]). The database is implemented as a Racket hash-table and uses the default key computation over the machine system state to store a single bit in the entry. A collision in the hash-table is assumed to imply a previously seen state.

Validation The process is validated on the connectionless message passing portion of the MCAPI communications library (The Multicore Association). There are 43 API calls in the library registry, and 18 of those are related to the connectionless message passing. We implement the 12 most relevant calls that cover the bulk of the functionality. The 4M comprises 488 lines of code (without comments) utilizing 3 distinct daemon transitions for internal house keeping on the library. The API state itself only contains 6 unique variables—two of which encode the library version and error status. The 4M descriptions compiles into 284 lines of 4M core.

Running times are measured with the Unix `time` command on an Intel Core 2 Quad 2.4 GHz machine with 4 GB of memory running Ubuntu 10.04. Running the scenario in Fig. 3.1(b) directly in Racket (not through the C runtime) in single execution mode takes 1.6 seconds. The same single execution through the C runtime takes 14.8 seconds which includes the overhead to setup and tear down the server. The bulk of the runtime appears to be in the pipe communication and state construction code that is currently naïvely implemented. As a reference point, the running time for the MCAPI dynamic verifier MCC on a simpler scenario with 3 threads, two of which perform parallel sends, and the third making two sequential receives is under 1 second (Sharma et al. [2009]). The tool, though, does rely critically on a reference implementation, which as discussed previously, does not include all the behavior allowed in the API.

Chapter 7

Related Work

There are several general purpose and purpose-built specification languages with complete frameworks for analysis and model checking. These include Promela, Murphi, TLA+, Z, Alloy, and B, to name a few (Holzmann [1997], Dill et al. [1992], Lamport, Spivey [1989], Jackson [2006], Abrial [1996]). There are two differentiators as related to the proposed approach in this paper: first, the connection in other solutions between the mathematical semantic definition of the language and the runtime is not clear whereas the math is the implementation in our solution; and second, the other solutions target analysis in the runtime whereas our work is intended as a drop-in API replacement.

There are several attempts to model MPI in existing specification languages including tools to convert from C programs using MPI to the chosen specification language (Siegel and Avrunin [2003], Georgelin et al. [1999], Palmer et al. [2007], Li et al. [2008]). Recent work takes CUDA and C to SMT languages (Wang et al. [2009], Li et al. [2010], Elwakil and Yang [2010b,a]). Such implementations are only suitable to scenario evaluation and not drop-in API replacement and have the extra burden of proving a correct translation to the analysis language.

Recent work in dynamic verification uses the program artifact directly as the model with the actual API implementation to perform model checking (Godefroid [1997], Mercer and Jones [2005], Păsăreanu et al. [2008], Musuvathi and Qadeer [2007], Wang et al. [2008], Vakkalanka et al. [2008]). Although the exhaustive search through continuations rather than repeated program invocation is similar to (Mercer and Jones [2005], Păsăreanu et al. [2008]), the proposed work in this paper does not critically rely on an existing runtime implementation; thus, it is able to elicit all behaviors captured in the specification and directly control internal API behavior. Without such control, verification results are dependent on the chosen implementation, even then, on just those implementation aspects that are controllable. For example, it is not possible to affect arbitrary buffering in the MPI or MCAPI runtime libraries and as a result, behaviors such as those in our example scenario are omitted in the analysis (Sharma et al. [2009]).

Chapter 8

Conclusion

English specification of concurrent APIs catalog interfaces and list effects of correct and incorrect calls to those interfaces. They also abstract the internal API state providing flexibility in implementing the intent of the API. At the same time they provide no framework with which a programmer or even a designer might experiment to further understand the API in the presence of many concurrent calls. Although it is possible to provide such framework in a formal specification, the specification is rarely created, and when it is, it is rarely used beyond the API designer to understand simple scenarios. Regardless, it does not provide an instance of the API suitable for exploration, test and debug, or exhaustive search for proof construction.

The work in this paper provides a drop-in replacement for concurrent APIs by

- creating a specification language, 4M, implemented as a term rewriting system that is suited for API specification;
- designing a novel role-based architecture to directly connect the C runtime to the 4M runtime to use the specification as an instance of the API that is explorable, testable, and capable of exhaustive search;
- providing an implementation of the rewriting system in Racket that is a direct embodiment of the mathematical description of the semantics; and
- validating the process in a portion of the MCAPI communication API.

The result is that when an API is now formally specified, it is possible to use the same specification with native programs written against the API to explore system-wide program behavior that existing solutions cannot reason about.

Future work includes (i) adapting reductions from model checking that combat the state explosion resulting from data and scheduling non-determinism; (ii) improving the code between different runtimes to reduce overhead; and (iii) case study in larger programs and APIs. For (i) we intend to implement a full partial order reduction for the 4M core that employs SMT technology to compute equivalence classes between concurrent schedules as in (Wang et al. [2009]). A particular challenge is modeling the API state in SMT and defining the partial order extent in the 4M core. We also intend to implement symmetry reduction in both data and threads. For (ii), we intend to look critically at search order and undo stacks for the heap to minimize the flow of data between the 4M implementation and runtime environment for the program itself. We also hope to leverage understanding of the partial order relationship in 4M core to implement macro-steps so scheduling only occurs at dependency points. And finally, for (iii), we hope to test the implementation against larger codes and different APIs. Of particular interest is the MCA API for resource allocation that manages shared memory regions for thread-like shared objects. It is not clear how the current approach needs to adapt to keep shared memory regions consistent in the internal API state without rewriting memory operations in the application code. We suspect that we can use some form of memory mapping to capture the effects of writes to common address spaces.

Appendix A

Syntax

This appendix presents the syntax of the full, human-readable form of 4M. The structure is similar to the 4M core in that a specification consists of state, transitions, and daemons. In addition, macros are added and a few syntactic-sugar operations are provided. Fig. A.1 presents this syntax. Fig. A.2 shows the conversion from syntactic sugar forms into other 4M language forms.

There are three kinds of macros allowed in full 4M. The first two types are `function` and `procedure` macros. These are invoked in a notation similar to function calling in most languages, where the bound macro identifier is given with parameters following in parentheses. Textual replacement is performed, substituting values of parameters into the macro body and then substituting that body into the application site. `function` macros produce expressions while `procedure` macros produce commands.

The third kind is a `let` macro, distinct from the `let` expression for local binding in the scope of the expression evaluation, and distinct from the `let` command for local binding in the scope of command evaluation. A `let` macro is similar, but applies to the scope of a transition or daemon and is substituted non-hygienically — specifically, that identifiers may capture variables bound in the scope of the application site that would not be in scope of the macro definition. These are useful to reduce the length of transitions which would otherwise repeat an expression in multiple locations (in rule guards and effects) and to give names to expressions for enhanced readability. Furthermore, 4M full syntax allows only identifiers on the left-hand-side of an assignment statement, whereas 4M core accepts any expression (so long as it evaluates to an address). Thus `let` macros allow naming an expression and using this name on the left-hand-side of an assignment statement.

```
specification ::= (state | transition | function | procedure)*
        state ::= state
                      (ID = expr)*
                  end
   transition ::= (transition | daemon) ID
                      (input ID (, ID)*)?
                      macro*
                      (rule rule* end)?
                      (errors rule* end)?
                  end
     function ::= function ID ( (ID (, ID)*)? )
                      expr
                  end
    procedure ::= procedure ID ( (ID (, ID)*)? )
                      command
                  end
         rule ::= expr ==> command command*
      command ::= ID ( (expr (, expr)*)? ) ;
                | (@)? ID' := expr ;
                | call ID ( (expr (, expr)*)? ) ;
                | tmp ID ;
                | choose ID in e ;
                | let ID = e ;
```

```
  macro ::= let ID = expr
   expr ::= value | ID | ID' | @ ID
          | unaop expr
          | expr binop expr
          | if expr then expr else expr fi
          | let ID = expr in expr
          | ( (\E | \A) pattern in expr : expr )
          | ( pattern in expr : expr )
          | { (| expr |)? pattern in expr : expr }
          | { (expr (, expr)*)? }
          | [ (expr (, expr)*)? ]
          | expr . NUMBER
          | ( expr )
  value ::= NUMBER | STRING | BOOLEAN | ERROR
pattern ::= ID
          | [ ID (, ID)* ]
```

```
binop → + - * / ^ %  /\  \/  \in \notin \U \int \ =  !=  >  <  >=  <=
truncate
unaop → - ! deset typeof
BOOLEAN → true | false
NUMBER → -? [0-9]+ (. [0-9]+)?
STRING → " ([^"] | \" )* "
ID → [a-Z_] [a-Z0-9_]*
COMMENT → #.*\n | (#.*#)
```

Figure A.1: The syntax of 4M.

Sugar Form	De-sugared Form
`(\E x in S : p(x))`	`{x in S : p(x)} != {}`
`(\A x in S : p(x))`	`{x in S : !p(x)} = {}`
`(x in S : p(x))`	`deset {x in S : p(x)}`
`a notin b`	`!(a in b)`

Figure A.2: Removal of syntactic sugar, largely dealing with quantification.

Appendix B

Full 4M Core

The description of 4M core in Chapter 4 excludes some of the rewrite rules for the expression machine. These are included here, along with the rest of the rules for convenience.

$$\begin{array}{rcl} astate & ::= & (\sigma\ \eta\ (acmd\ ...)) \\ acmd & ::= & (\texttt{choose}\ id\ e) \\ & | & (\texttt{let}\ ([id\ e]\ ...)) \\ & | & (\texttt{alloc}\ id) \\ & | & (\texttt{upd}\ (\texttt{@}\ e\ e)...) \end{array}$$

Figure B.1: Atomic Block Machine syntax

$$\textsc{Choose}\quad \frac{(\sigma\ \eta\ e\ \texttt{ret}) \rightarrow_e^* (\sigma\ \eta\ (\texttt{const-set}\ v_0\ ...\ v\ v_1\ ...)\ \texttt{ret})}{\begin{array}{c}(\sigma\ \eta\ ((\texttt{choose}\ id\ e)\ acmd\ ...)) \rightarrow_c \\ (\sigma\ (\eta[id\ \mapsto\ v]...)\ (acmd\ ...))\end{array}}$$

$$\textsc{Let}\quad \frac{(\sigma\ \eta\ ()\ (e\ ...)) \rightarrow_{es}^* (\sigma\ \eta\ (v\ ...)\ ())}{\begin{array}{c}(\sigma\ \eta\ ((\texttt{let}\ ([id\ e]\ ...))\ acmd\ ...)) \rightarrow_c \\ (\sigma\ (\eta[id\ \mapsto\ v]...)\ (acmd\ ...))\end{array}}$$

$$\textsc{Alloc}\quad \frac{address = \sigma_{malloc}(\sigma)}{\begin{array}{c}(\sigma\ \eta\ ((\texttt{alloc}\ id)\ acmd\ ...)) \rightarrow_c \\ ((\sigma\ [address\ \mapsto\ 0])\ (\eta\ [id\ \mapsto\ (\texttt{addr}\ address)])\ (acmd\ ...))\end{array}}$$

$$\textsc{Store Update}\quad \frac{\begin{array}{c}(\sigma\ \eta\ ()\ (e_{id}\ ...\ e\ ...)) \rightarrow_{es}^* (\sigma\ \eta\ ((\texttt{addr}\ address)\ ..._0\ v\ ..._0)\ ()) \\ \sigma' = \sigma[address\ \mapsto\ v]\ ...\end{array}}{(\sigma\ \eta\ ((\texttt{upd}\ (e_{id}\ e)\ ..._0)\ acmd\ ...)) \rightarrow_c (\sigma'\ \eta\ (acmd\ ...))}$$

Figure B.2: Atomic Block Machine reductions ($\rightarrow_c$).

$$
\begin{array}{rcl}
\mathit{estate} & ::= & (\sigma\ \eta\ e\ k) \\
\sigma & ::= & \emptyset \mid \texttt{(}\sigma\ \texttt{[}\ \mathit{address} \mapsto v\ \texttt{])} \\
\eta & ::= & \emptyset \mid \texttt{(}\eta\ \texttt{[}\ \mathit{id} \mapsto v\ \texttt{])} \\
\mathit{ae} & ::= & v \mid \mathit{id} \\
e & ::= & \mathit{ae} \mid \texttt{(@}\ e\texttt{)} \\
 & \mid & \texttt{(}\mathit{setop}\ \texttt{(}\mathit{pattern}\ \texttt{in}\ e\texttt{)}\ e\texttt{)} \\
 & \mid & \texttt{(if}\ e\ e\ e\texttt{)} \\
 & \mid & \texttt{(let ([}\mathit{id}\ e\texttt{])}\ e\texttt{)} \\
 & \mid & \texttt{(}\mathit{op}\ e\ e\ \ldots\texttt{)} \\
v & ::= & \textbf{number} \\
 & \mid & \texttt{true} \mid \texttt{false} \\
 & \mid & \texttt{error} \\
 & \mid & \textbf{string} \\
 & \mid & \texttt{(addr}\ \mathit{address}\texttt{)} \\
 & \mid & \texttt{(const-set}\ v\ \ldots\texttt{)} \\
 & \mid & \texttt{(const-tuple}\ v\ \ldots\texttt{)} \\
\mathit{pattern} & ::= & \mathit{id} \mid \texttt{(tuple}\ \mathit{id}\ \ldots\texttt{)} \\
\mathit{op} & ::= & \mathit{binop} \mid \mathit{unaop} \mid \texttt{set} \mid \texttt{tuple} \\
\mathit{setop} & ::= & \texttt{setFilter} \mid \texttt{setBuild} \\
k & ::= & \texttt{ret} \\
 & \mid & \texttt{(@ * ->}\ k\texttt{)} \\
 & \mid & \texttt{(}\mathit{setop}\ \texttt{(}\mathit{pattern}\ \texttt{in *)}\ e\ \texttt{->}\ k\texttt{)} \\
 & \mid & \texttt{(if *}\ e\ e\ \texttt{->}\ k\texttt{)} \\
 & \mid & \texttt{(pop}\ \eta\ k\texttt{)} \\
 & \mid & \texttt{(let ([}\mathit{id}\ \texttt{*])}\ e\ \texttt{->}\ k\texttt{)} \\
 & \mid & \texttt{(}\mathit{op}\ \texttt{(}v\ \ldots\texttt{)* (}e\ \ldots\texttt{)->}\ k\texttt{)}
\end{array}
$$

Figure B.3: Expression Machine syntax

Dereference cont

$$(\sigma\ \eta\ (@\ e)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e\ (@\ \texttt{*}\ \rightarrow k))$$

setop cont

$$(\sigma\ \eta\ (\mathit{setop}\ (\mathit{pattern}\ \texttt{in}\ e_s)\ e_b)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_s\ (\mathit{setop}\ (\mathit{pattern}\ \texttt{in}\ *)\ e_b \rightarrow k))$$

if cont

$$(\sigma\ \eta\ (\texttt{if}\ e_c\ e_t\ e_f)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_c\ (\texttt{if}\ *\ e_t\ e_f \rightarrow k))$$

let cont

$$(\sigma\ \eta\ (\texttt{let}\ ([\mathit{id}\ e_i])\ e_b)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_i\ (\texttt{let}\ ([\mathit{id}\ *])\ e_b \rightarrow k))$$

apply cont

$$(\sigma\ \eta\ (\mathit{op}\ e_0\ e_1\ \ldots)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_0\ (\mathit{op}\ ()\ (e_1\ \ldots) \rightarrow k))$$

argument eval

$$(\sigma\ \eta\ v_1\ (\mathit{op}\ (v_0\ \ldots)\ *\ (e_0\ e_1\ \ldots)\ k) \rightarrow_e$$
$$(\sigma\ \eta\ e_0\ (\mathit{op}\ (v_0\ \ldots v_1)\ *\ (e_1\ \ldots) \rightarrow k))$$

pop eta

$$(\sigma\ \eta_1\ v\ (\texttt{pop}\ \eta_0\ k)) \rightarrow_e (\sigma\ \eta_0\ v\ k)$$

Variable lookup

$$(\sigma\ \eta\ \mathit{id}\ k) \rightarrow_e (\sigma\ \eta\ \eta(\mathit{id})\ k)$$

Dereference

$$(\sigma\ \eta\ (\texttt{addr}\ \mathit{address})\ (\texttt{@}\ \texttt{*}\ \rightarrow k)) \rightarrow_e (\sigma\ \eta\ \sigma(\mathit{address})\ k)$$

Binary Op

$$\frac{v_{res} = v_{lhs}\ \mathit{binop}\ v_{rhs}}{(\sigma\ \eta\ v_{rhs}\ (\mathit{binop}\ (v_{lhs})\ \texttt{*}\ () \rightarrow k)) \rightarrow_e (\sigma\ \eta\ v_{res}\ k)}$$

Unary Op

$$\frac{v_{res} = \mathit{unaop}\ v_{rhs}}{(\sigma\ \eta\ v_{rhs}\ (\mathit{unaop}\ ()\ \texttt{*}\ () \rightarrow k)) \rightarrow_e (\sigma\ \eta\ v_{res}\ k)}$$

Set Expr

$$\frac{\begin{array}{c}(v\ \ldots) = (v_{before}\ \ldots\ v_{last}) \\ v_{set} = \mathit{removedups}((\texttt{const-set}\ v\ \ldots))\end{array}}{(\sigma\ \eta\ v_{last}\ (\texttt{set}\ (v_{before}\ \ldots)\ \texttt{*}\ () \rightarrow k)) \rightarrow_e (\sigma\ \eta\ v_{set}\ k)}$$

Tuple Expr

$$\frac{(v\ \ldots) = (v_{before}\ \ldots\ v_{last})}{\begin{array}{l}(\sigma\ \eta\ v_{last}\ (\texttt{tuple}\ (v_{before}\ \ldots)\ \texttt{*}\ () \rightarrow k)) \rightarrow_e \\ (\sigma\ \eta\ \texttt{const-tuple}\ v\ \ldots\ k)\end{array}}$$

If expression (true)

$$(\sigma\ \eta\ \texttt{true}\ (\texttt{if}\ ()\ \texttt{*}\ (e_{true}\ e_{false}) \rightarrow k)) \rightarrow_e (\sigma\ \eta\ e_{true}\ k)$$

If expression (false)

$$(\sigma\ \eta\ \texttt{false}\ (\texttt{if}\ ()\ \texttt{*}\ (e_{true}\ e_{false}) \rightarrow k)) \rightarrow_e (\sigma\ \eta\ e_{false}\ k)$$

Let expression

$$(\sigma\ \eta\ v\ (\texttt{let}\ ([\mathit{id}\ *])\ e \rightarrow k)) \rightarrow_e (\sigma\ (\eta\ [\mathit{id}\ \mapsto\ v])\ e\ (\texttt{pop}\ \eta\ k))$$

Set Filter - empty set

$$(\sigma\ \eta\ (\texttt{const-set})\ (\texttt{setFilter}\ (\mathit{pattern}\ \texttt{in}\ \texttt{*})\ e \rightarrow k)) \rightarrow_e (\sigma\ \eta\ (\texttt{const-set})\ k)$$

Set Filter - one element

$$\begin{array}{l}(\sigma\ \eta\ (\texttt{const-set}\ v_1\ v\ \ldots)\ (\texttt{setFilter}\ (\mathit{pattern}\ \texttt{in}\ \texttt{*})\ e \rightarrow k)) \rightarrow_e \\ (\sigma\ \eta\ (\texttt{union}\ \ (\texttt{if}\ (\mathit{pattern-let}\ \mathit{pattern}\ v_1\ e)\ (\texttt{const-set}\ v_1)\ (\texttt{const-set})) \\ \quad (\texttt{setFilter}\ (\mathit{pattern}\ \texttt{in}\ (\texttt{const-set}\ v\ \ldots))\ e))\ k)\end{array}$$

Set Build - empty set

$$(\sigma\ \eta\ (\texttt{const-set})\ (\texttt{setBuild}\ (\mathit{pattern}\ \texttt{in}\ \texttt{*})\ e \rightarrow k)) \rightarrow_e (\sigma\ \eta\ (\texttt{const-set})\ k)$$

Set Build - one element

$$\begin{array}{l}(\sigma\ \eta\ (\texttt{const-set}\ v_1\ v\ \ldots)\ (\texttt{setBuild}\ (\mathit{pattern}\ \texttt{in}\ \texttt{*})\ e \rightarrow k)) \rightarrow_e \\ (\sigma\ \eta\ (\texttt{union}\ \ (\texttt{set}\ (\mathit{pattern-let}\ \mathit{pattern}\ v_1\ e))\ \ (\texttt{setBuild}\ (\mathit{pattern}\ \texttt{in}\ (\texttt{const-set}\ v\ \ldots))\ e))\ k)\end{array}$$

Eval Leftmost

$$\frac{(\sigma\ \eta\ e_{target}\ \texttt{ret}) \rightarrow_e (\sigma\ \eta\ v_{target}\ \texttt{ret})}{(\sigma\ \eta\ (v\ \ldots)\ (e_{target}\ e\ \ldots)) \rightarrow_{es} (\sigma\ \eta\ (v\ \ldots\ v_{target})\ (e\ \ldots))}$$

Figure B.4: Expression Machine reductions ($\rightarrow_e$)

$$\begin{array}{rcl}
\mu & ::= & s\ (t \mid d) \ldots \\
s & ::= & \texttt{((}address\ v\texttt{)}\ldots\texttt{)} \\
t & ::= & \texttt{(transition}\ id\ \texttt{(}id\ldots\texttt{)(}r\ldots\texttt{))} \\
d & ::= & \texttt{(daemon}\ id\ \texttt{()(}r\ldots\texttt{))} \\
r & ::= & \texttt{(}e\ c\texttt{)} \\
c & ::= & \texttt{(}acmd\ \ldots\ tcmd\texttt{)} \\
tcmd & ::= & \texttt{(select}\ r\ \ldots\texttt{)} \\
 & \mid & \texttt{(call/k}\ id\ \texttt{(}ae\ \ldots\texttt{)}\ \ id\ \texttt{(}ae\ \ldots\texttt{))} \\
 & \mid & \texttt{(tail-call}\ id\ \texttt{(}ae\ \ldots\texttt{))} \\
 & \mid & \texttt{ret} \\
tstate & ::= & \texttt{(}\mu\ \sigma\ \eta\ tcmd\ ck\texttt{)} \\
ck & ::= & \texttt{ret} \\
 & \mid & \texttt{(}\omega\ id\texttt{)} \\
 & \mid & \texttt{(call}\ id\ \texttt{(}v\ \ldots\texttt{) -> }ck\texttt{)}
\end{array}$$

Figure B.5: Thread Machine syntax

Select

$$\frac{\begin{array}{l}(r_0\ \ldots\ r_1\ r_2\ \ldots) = (r\ \ldots) \qquad e_{pre} = \mathit{precondition}(r_1) \\ (\sigma\ \eta\ e_{pre}\ \texttt{ret}) \rightarrow_e^* (\sigma\ \eta\ \texttt{true}\ \texttt{ret}) \qquad (acmd\ \ldots\ tcmd) = \mathit{effects}(r_1) \qquad (\sigma\ \eta\ (acmd\ \ldots)) \dashrightarrow_c^* (\sigma'\ \eta'\ ())\end{array}}{(\mu\ \sigma\ \eta\ (\texttt{select}\ r\ \ldots)\ ck) \rightarrow_t (\mu\ \sigma'\ \eta'\ tcmd\ ck)}$$

Tail Call

$$\frac{\begin{array}{l}(id_x\ \ldots) = \mathit{arguments}(\mu, id) \\ (r\ \ldots) = \mathit{rules}(\mu, id) \qquad (\sigma\ \eta\ ()\ (ae\ \ldots)) \rightarrow_{es}^* (\sigma\ \eta\ (v\ \ldots)\ ()) \qquad \eta' = \emptyset[id_x\ \mapsto\ v]\ \ldots\end{array}}{(\mu\ \sigma\ \eta\ (\texttt{tail}-\texttt{call}\ id\ (ae\ldots))\ ck) \rightarrow_t (\mu\ \sigma\ \eta'\ (\texttt{select}\ r\ \ldots)\ ck)}$$

Call with continuation

$$\frac{(\sigma\ \eta\ ()\ (ae_k\ \ldots)) \rightarrow_{es}^* (\sigma\ \eta\ (v_k\ \ldots)\ ())}{\begin{array}{l}(\mu\ \sigma\ \eta\ (\texttt{call/k}\ id_0\ (ae_0\ldots)\ id_k\ (ae_k\ldots))\ ck) \rightarrow_t \\ (\mu\ \sigma\ \eta\ (\texttt{tail}-\texttt{call}\ id_0\ (ae_0\ldots)\ (\texttt{call}\ id_k\ (v_k\ldots) \rightarrow ck))\end{array}}$$

Continuation call

$$\begin{array}{l}(\mu\ \sigma\ \eta\ \texttt{ret}\ (\texttt{call}\ id\ (v\ldots) \rightarrow ck)) \rightarrow_t \\ (\mu\ \sigma\ \emptyset\ (\texttt{tail}-\texttt{call}\ id\ (v\ldots))\ ck)\end{array}$$

Omega

$$(\mu\ \sigma\ \eta\ \texttt{ret}\ (\omega\ id)) \rightarrow_t (\mu\ \sigma\ \emptyset\ (\texttt{tail}-\texttt{call}\ id\ ())\ (\omega\ id))$$

Figure B.6: Thread Machine reductions ($\rightarrow_t$)

$$\begin{array}{rcl}
mstate & ::= & (\mu\ \sigma\ (thread\ \ldots)\ (thread\ \ldots)) \\
thread & ::= & (\eta\ tcmd\ ck)
\end{array}$$

Figure B.7: System Machine syntax

System Step (non-daemon)

$$\frac{(\mu\ \sigma\ \eta_1\ \mathit{tcmd_1}\ \mathit{ck_1}) \dashrightarrow_t (\mu\ \sigma'\ \eta'_1\ \mathit{tcmd_1}'\ \mathit{ck_1}')}{\begin{array}{c}(\mu\ \sigma\ (\mathit{thread_0}\ ...\ (\eta_1\ \mathit{tcmd}_1\ \mathit{ck}_1)\ \mathit{thread_2}\ ...)\ \mathit{threads}_d) \rightarrow_m \\ (\mu\ \sigma'\ (\mathit{thread_0}\ ...\ (\eta'_1\ \mathit{tcmd}'_1\ \mathit{ck}'_1)\ \mathit{thread_2}\ ...)\ \mathit{threads}_d)\end{array}}$$

System Step (daemon)

$$\frac{\mathit{active}(\mathit{threads}_u) \qquad (\mu\ \sigma\ \eta_1\ \mathit{tcmd_1}\ \mathit{ck_1}) \dashrightarrow_t (\mu\ \sigma'\ \eta'_1\ \mathit{tcmd_1}'\ \mathit{ck_1}')}{\begin{array}{c}(\mu\ \sigma\ \mathit{threads}_u\ (\mathit{thread_0}\ ...\ (\eta_1\ \mathit{tcmd}_1\ \mathit{ck}_1)\ \mathit{thread_2}\ ...)) \rightarrow_m \\ (\mu\ \sigma'\ \mathit{threads}_u\ (\mathit{thread_0}\ ...\ (\eta'_1\ \mathit{tcmd}'_1\ \mathit{ck}'_1)\ \mathit{thread_2}\ ...))\end{array}}$$

Figure B.8: System Machine reductions ($\rightarrow_m$)

Appendix C

4M Compilation

Full 4M is compiled into 4M core in which the semantics of 4M are specified. This compilation uses a top-down LL(k) parser and several transformation passes. These passes are: Parsing, Macro Expansion, De-sugaring, Kernel Preparation, Parenthetical Output. Each is described below.

Parsing The parser is defined in an ANTLR top-down LL(k) grammar (Parr and Quong [1995]). Rules are appropriately factored to handle recursion, associativity, and precedence. Tokens are created to disambiguate between binary minus and unary negation, as well as tokens for other helpful markers in the standard ANTLR abstract syntax tree (AST).

Macro Expansion Macros are expanded in two passes during this phase. The first pass collects macros, the second replaces them. `let` macros are expanded during the first pass, since the grammar restricts there definitions to the beginning of the transition or daemon in which their scope operates, thus they are defined before any application site. Macros are replaced repeatedly until a fixed point is reached. Macros cannot be recursive since cycle detection prevents this as well as any infinite macro replacement loop.

Note that the grammar for each of these transformation phases is simplified since associativity and precedence are already encoded in the AST. The grammars are "tree grammars" as they operate on tokens in the AST rather than on lexer tokens.

De-sugaring Fig. A.2 gives a list of de-sugaring transformations that occur in terms of the full 4M syntax. In addition to these, set construction expressions that define an output pattern (between vertical bars) in addition to a search pattern are converted to a "set-build" operation with an inner "set-filter" operation. Set construction expressions without the output pattern are merely filters, and thus are already in the "set-filter" form. These are the two set operations shown in the 4M core.

Kernel Preparation This phase converts a 4M AST into 4M core AST. It builds a symbol table for state variables so that address locations may be assigned i the next phase. The required transformations for this phase are:

1. Separate lists of success and error rules are combined into a single list of rules.
2. The effects of a rule are converted into CPS.
3. A sequence of updates becomes a single multi-update.

The first transformation builds a negated disjunction of the rule guards for all error rules in a transition or daemon to union with the rule guard of each success rule. This ensures that success rules are enabled only if no error rule is enabled, preserving the semantics of full 4M which states that error rules have precedence.

The second transformation is affected by creating new transitions with unique names for the rest of the computation (the "continuation") of a command list following a `call` command. A `call` command may indicate guards that prevent forward progress until met. These are placed as the rule guard in the generated transition. Alternatively, the `call` command may be followed by a list of rules to handle alternative outputs of the transition call. These rules become the body of the generated transition. It is an error, enforced by this phase, to have further commands after a `call` command that gives a list of rules. Transition calls that give only progress guards (or no guards) may have following commands. If there are no following commands, a 4M core `tail-call` command is generated instead of a `call/w`.

The third transformation proceeds by iterating over the commands in a rule's effect, keeping the `tmp`, `let`, and `choose` commands while queuing up the `upd` commands rather than outputting them in the new AST. When the command list is finished or a `call` command is reached, all the queued `upd` commands are combined into a single multi-update and output.

As a help to the user, this phase also keeps a symbol table to report undefined (or misspelled) symbols, and performs minimal type checking. The type system assigns a type to constants and the special type "any" to transition parameters and state variables. Type rules for each operator accept the correct types or the "any" type. Most type proofs quickly become type "any", but any type errors that are obvious are detected. Although a full type-inference phase may prove helpful, this minimal checking was helpful in developing the MCAPI 4M specification.

Parenthetical Output The output format is an AST serialized as tuples grouped by parentheses. This phase walks the transformed AST to output each transition and daemon. As it encounters a state variable it replaces it with the address assigned in the symbol table computed in the preceding phase. The final output includes the initial store first followed by each transition and daemon.

Appendix D

MCAPI 4M Specification

Below is the full code listing (with most comments removed) of the 4M specification of MCAPI. It implements 14 of the 18 connectionless API calls, our of 43 total API calls. We wrote wrappers only for 12, ignoring two endpoint attribute functions. We chose this subset as the most interesting, since connection-oriented communication calls do not lead to deadlocks and rarely have interesting properties to observe. Since they are simpler than message passing they can be trivially added, requiring merely author time.

In the next sections we discuss the implementation of MCAPI in 4M in detail, serving as an in-depth look at implementing a real API, as well as giving insight into the MCAPI 4M specification and drop-in which may be useful for designers, implementers, and users of the drop-in.

D.1 Interesting Behaviors

We focused our specification modeling efforts on areas that contain interesting behaviors. Connection-oriented communication, like the packet and scalar channels in MCAPI, are simple and do not lead to deadlocks. Connectionless communication primitives like send and receive, however, provide many interesting behaviors. MCAPI receives will receive a message sent from any thread, thus races and deadlocks can occur when message order is not what the programmer intended. Non-determinism in the reception order of messages was shown in the example in Chapter 3 where the message is copied to internal buffers, passed, and then eventually copied into the receiving buffer, leaving opportunities for messages to be delayed in transit and other messages to be delivered first.

Since connectionless communication operations are the most interesting, we defined them in our 4M specification – both blocking and non-blocking variants – with the other calls necessary for initialization, endpoint creation, and finalization.

D.2 Data Structures

4M requires the internal state of the API to be explicitly declared. Lines 1–8 of the MCAPI 4M specification show this state. Next to each is a comment indicating what the variable should contain. 4M does not have complex types, nor does it include type annotations or contracts. The specification writer must therefore manage what type of information is kept in each variable.

Lines 10–84 show the macros used to help manage these data structures. For example, lines 20–23 give functions for accessing elements of a tuple that constitutes an endpoint record. This allows the specification writer to abstract the specific tuple indexes and instead treat the tuple as a record with named fields. Lines 24–26 show a macro for creating new endpoint records. Lines 27–29 show a macro that allows updating one member of an endpoint record (the one member that needs to be updated). Lines 11–19 show helper functions to look up data in the internal state, such as `getNodeId` which looks up a node (thread or process) in the internal `ActiveNodes` state variable and returns the corresponding ID.

Endpoints use a FIFO queue for received messages. This data structure is implemented functionally in lines 59–81. Line 60 shows the state of an empty FIFO queue: it contains a set with index-value pairs that acts as a map, a minimum index, and a maximum index. The minimum points at the first element in the queue, the maximum indicates the last index added. Thus the maximum starts as -1 when the list is empty. Adding an element to the queue inserts a pair into the map with the next largest index (maximum plus one), then increases the maximum (as seen in lines 62–65). Getting the next element searches the map for the element with the index indicated by minimum (lines 70–73). Removing this element uses set subtraction then increments the minimum value to point to the new head of the list (lines 74–81).

This encoding of a FIFO queue is used, rather than a linked list, because a linked list would require recursion for some FIFO queue operations, yet 4M macros cannot be recursive. Linked lists work for a stack, and are thus used in some 4M specifications, like the example in Fig. 3.2.

The use of macros to implement data structures abstracts the details and keeps the remainder of the 4M specification clean.

D.3 Blocking Calls

MCAPI includes both blocking and non-blocking variants of the send and receive operations. The blocking version returns after the message has been copied out of or into the node's buffer. The non-blocking version returns immediately and the node must call `wait` to determine when the buffer can be reused. A blocking call is equivalent to a non-blocking call followed by a `wait`.

In our 4M specification of MCAPI, we model blocking calls precisely in this way: a transition for a blocking call makes a sub-call to the transition for the non-blocking call followed by a sub-call to the transition for `wait`. An example of this is the `msg_send` call in lines 415–431. Since the non-blocking `msg_send_i` returns a request object, the `msg_send` rule creates a temporary variable `RequestAddr` to hold this (line 420). It then passes this request on to wait (line 426). wait has a return value and also gives the message size. msg_send does not need those, but creates temporary variables (lines 421–422) to pass in to wait and then ignores the values.

One catch to this implementation is that msg_send_i may fail. The MCAPI English specification indicates that the list of errors from msg_send is the same as those returned by msg_send_i combined with those of wait. Thus if either sub-call fails, the error status should be returned to the caller of msg_send. This is accomplished by passing StatusAddr from msg_send's parameters into each sub-call. However, msg_send must still check whether msg_send_i has errored before calling wait. This is why the branching continuation rules in lines 425–428 are necessary. Lines 427–428 represent the case where an error occurred, in which case the continuation is to do nothing (indicated by the useless let nop) and allow the error to be reported to the caller in the StatusAddr variable.

D.4 Non-blocking Calls

The non-blocking calls must return immediately, but also must remember the work to do later and have a way of completing this work. Line 4 shows the API state variable Requests which contains all the pending work from non-blocking calls. The non-blocking calls simply insert their data into the Requests list and return. A daemon transition will later complete the request non-deterministically.

msg_send_i, on lines 366–405, is an example of a non-blocking call. After checking all the possible error conditions for this call (lines 376–404), it places a "msg_send" request in the list (lines 371–372), returns the request id which will serve as the request object handle (line 373), and returns a success status (line 374).

_finish_msg_send_i is the corresponding daemon to msg_send_i. This daemon must non-deterministically select a request that can be completed, then do so. There may be zero, one, or many such requests. The transition guard must be able to determine whether there is at least one, then the effect needs to non-deterministically select one to use. Lines 433–446 are a local macro that constructs the set of all pending, valid "msg_send" requests for which the send can be completed. Lines 437–445 are the portion that indicate the request can be completed. We will further discuss these lines in the next section. The rule guard simply checks whether this set is non-empty. Line 452 shows the use of choose to bind the local variable req to a non-deterministically chosen element from that set. Finally lines 455-457 copy the message to the receiving endpoint's queue and mark the request as finished.

D.5 Message Non-Overtaking

Message non-overtaking is the property that two messages sent from the same endpoint to the same endpoint will arrive in order. Only messages sent from different sources may be reordered. This property is of vital importance to the semantics of MCAPI. We confirmed its meaning through discussions with MCAPI designers.

The 4M specification implements message non-overtaking in the daemons that process sends and receives. Lines 437–445 in the send daemon and lines 517–523 in the receive daemon are the key lines to note. In the send daemon, after a request has been identified as a pending, valid request of the right type, a term in the set builder predicate asserts that there is not another request that should be received first (lines 437–438). The next lines are the predicate determining if another message should be sent first. Line 442 says "sending from the same place", line 443 says "and to the same place", line 444 says "and this one was sent first". Line 444 asserts the order by appealing to the monotonically-increasing request ids. These three properties are those required by the definition of message non-overtaking, as indicated in the previous paragraph.

Lines 517–523 in the receive daemon operate similarly, to ensure that when multiple receives are posted for the same endpoint – for example, two non-blocking receives in a row before the call to `wait` – that messages read from the endpoint queue in order are also assigned to receive requests in order. This exposes another property of MCAPI semantics, which is that a `wait` on the second non-blocking receive (for the same endpoint) will ensure that both receives have completed.

D.6 Implementation Specific Properties

MCAPI is defined loosely on purpose to allow the greatest freedom for implementers to adapt. Due to this freedom, some properties of MCAPI will be implementation dependent. These include: internal memory limits, endpoint attributes, message priorities, API version number, and various failure errors.

The specification for `msg_send_i` includes errors for no more internal buffers (`MCAPI_ENO_BUFFER`), no more request handles (`MCAPI_ENO_REQUEST`), and no more memory (`MCAPI_ENO_MEM`). These are shown in the 4M specification on lines 385–392. Since they are implementation dependent, we instead model a system with arbitrarily large memory so the rule guards are simply `false` to disable them. They could, however, be modeled by tracking the number of buffers, number of requests, or amount of memory usage consumed by the current set of requests and comparing to limits adjusted to match some specific implementation.

Endpoint attributes allow implementation-specific properties on endpoints. These are not interesting so we ignore them. A model of a specific implementation might include some specific attributes.

Messages can be given a priority in MCAPI, but the MCAPI specification does not make any statements about the use of this property. Priority can be completely ignored by an implementation. In our 4M specification we do ignore it, since that provides the greater range of API behaviors.

The API version number is handled by setting a state variable (line 6). This value is returned in the initialization call. It should be changed to match whichever version of MCAPI the model represents. Properties for memory limits may also be represented as state variables that a user of the 4M specification can change to match a specific MCAPI implementation.

There are other error messages allowed by MCAPI calls that depend on implementation details or that can only occur under certain situations that may not arise in some implementations. For example, lines 394–396 report an invalid priority for a message send, but the list of valid priorities is implementation dependent. (In our 4M specification, any positive integer is considered valid.) Lines 381–383 indicate a limit on the maximum size of a message, which is handled with the configuration state variable `_MaxMsgSize`. Lines 201–203 indicate an error in creating an endpoint on a node that is not allowed to create endpoints, yet there is nothing in the official MCAPI specification to indicate a node cannot do so, this is implementation dependent. Finally, each call may report an invalid status parameter error. An implementation may have requirements for the type of data structure used as a status parameter. However, if the status parameter is simply a null-pointer it cannot be used to report the status! Hence on lines 113–114 and similarly in other transitions we set an error flag to indicate this error, since there is no other way to report it. A verification tool may add a monitor on this variable to report when this error occurs.

D.7 Tricky Timeout

The MCAPI `wait` call allows an application to specify timeout values. Our 4M specification ignores this. There is no measure of time in 4M, thus it cannot be correctly implemented. Adding a notion of time is difficult. Counting real time allows the program to observe the slower performance of the drop-in architecture versus an actual implementation, causing timeouts to occur more often than they should or causing the programmer to increase the timeout bounds just for use with the drop-in. Counting a false notion of time breaks the smooth transition from the C program to the drop-in as well, requiring the program to change or some form of translation from real time to model time. We chose simply to avoid the issue.

Timeout could, however, be modeled as one more non-deterministic choice available by giving the timeout rule an always-enabled guard as long as the timeout parameter is set. This would allow exploration of both behaviors. However, it would be problematic for the random-walk exploration using the drop-in, since timeouts would occur frequently whereas they should occur rarely in an running actual program. It would be difficult for a user to explore a complete execution trace when each `wait` call had a random chance of timing out. Some users may prefer this option, since it would encourage defensive programming. The 4M specification could be modified to include this behavior, however we chose not to for the time being.

D.8 Full MCAPI 4M specification

```
state
    ActiveNodes = {}        #Map of Node -> NodeId
    Endpoints = {}          #Set of [NodeId, PortId, EndpointId, Messages]
    AssignedAttributes = {} #Set of [EndpointId, AttrNum, value]
    Requests = {}           #Set of [RequestType, RequestId, Status, Valid, CreatorNode, Data]
    _LibraryVersion = 1
    ErrorStatus = false
end

## Data Structure functions ##
function getNodeId(Node)
    ([node, nodeId] in ActiveNodes : node = Node).1
end
function getNode(NodeId)
    ([node, nodeId] in ActiveNodes : nodeId = NodeId).0
end
function getEndpoint(EpId)
    (ep in Endpoints : ep_id(ep) = EpId )
end
function ep_node_id(Ep) Ep.0 end
function ep_port(Ep)    Ep.1 end
function ep_id(Ep)      Ep.2 end
function ep_msgs(Ep)    Ep.3 end
function newEndpoint(NodeId, Port)
    [NodeId, Port, getMaxEndpointId() + 1, fifo_empty()]
end
function changeEpQueue(ep, newq)
    [ep.0, ep.1, ep.2, newq]
end
function getMaxEndpointId()
    if Endpoints = {} then
        -1
    else
        ep_id( (ep in Endpoints : (\A ep2 in Endpoints : ep_id(ep) >= ep_id(ep2) )) )
    fi
end
function getRequest(ReqId)
    (req in Requests : req_id(req) = ReqId) #old FSpec had /\ valid = true
end
function req_type(Req)    Req.0 end
function req_id(Req)      Req.1 end
function req_status(Req)  Req.2 end
function req_valid(Req)   Req.3 end
function req_creator(Req) Req.4 end
function req_data(Req)    Req.5 end
function newRequest(Type, CreatorNode, Data)
    [Type, getMaxRequestId() + 1, "Pending", true, CreatorNode, Data]
end
function getMaxRequestId()
    if Requests = {} then
        -1
    else
        req_id( (req in Requests : (\A req2 in Requests : req_id(req) >= req_id(req2) )) )
    fi
end
function hasRequest(ReqId)
    (\E req in Requests : req_id(req) = ReqId) #old FSpec had /\ valid = true
end
function fifo_empty()
    [{}, 0, -1] #map of index=>value, min-index, max-index
end
function fifo_add(lst, value)
    let list = lst in
        [list.0 \U {[list.2 + 1, value]}, list.1, list.2 + 1]
end
function fifo_hasNext(lst)
    let list = lst in
```

```
        list.1 <= list.2
end
function fifo_next(lst)
    let list = lst in
        (pair in list.0 : pair.0 = list.1).1
end
function fifo_remove(lst)
    let list = lst in
        if fifo_hasNext(list) then
            [list.0 \ {pair in list.0 : pair.0 = list.1}, list.1 + 1, list.2]
        else
            list
        fi
end
function min(a, b)
    if a < b then a else b fi
end

### MCAPI CALLS ###

## INITIALIZATION
transition initialize
    input Node, NodeId, VersionAddr, StatusAddr
    rule
        true ==>
            @StatusAddr' := "MCAPI_SUCCESS";
            @VersionAddr' := _LibraryVersion;
            ActiveNodes' := ActiveNodes \U {[Node, NodeId]};
    end
    errors
        false (#could not initialize#) ==>
            @StatusAddr' := "MCAPI_ENO_INIT";

        ValidStatusParam(StatusAddr) /\
        (\E [n, nid] in ActiveNodes : n = Node ) ==>
            @StatusAddr' := "MCAPI_INITIALIZED";

        ValidStatusParam(StatusAddr) /\
        (!ValidNode(Node) \/ (\E [n, nid] in ActiveNodes : nid = NodeId )) ==>
            @StatusAddr' := "MCAPI_ENODE_NOTVALID";

        ValidStatusParam(StatusAddr) /\
        !ValidVersionParam(VersionAddr) ==>
            @StatusAddr' := "MCAPI_EPARAM";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end
function ValidStatusParam(StatusAddr)
    ValidAddr(StatusAddr)
end
function ValidVersionParam(VersionAddr)
    ValidAddr(StatusAddr)
end
function ValidAddr(Addr)
    typeof(Addr) = "address"
end
function ValidNode(Node)
    true #whatever we're using as node identifiers, just accept it
end
function ValidRequestParam(RequestAddr)
    ValidAddr(RequestAddr)
end
function ValidBufferParam(BufferAddr)
    ValidAddr(BufferAddr)
end

transition finalize
    input Node, StatusAddr
    let pending = { req in Requests: req_status(req) = "Pending" /\ req_creator(req) = Node }
    let nodeId = getNodeId(Node)
    let canceled = { | [req_type(r), req_id(r), "Canceled", req_valid(r), req_creator(r),
                       req_data(r)] | r in pending : true }
    rule
        true ==>
            ActiveNodes' := ActiveNodes \ { [node, nid] in ActiveNodes : node = Node };
            @StatusAddr' := "MCAPI_SUCCESS";
            Endpoints' := Endpoints \ { ep in Endpoints : ep_node_id(ep) = nodeId };
            Requests' := (Requests \ pending) \U canceled;
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !(\E [node, nid] in ActiveNodes: node = Node) \/ false (#could not finalize#) ==>
            @StatusAddr' := "MCAPI_ENO_FINAL";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end
transition get_node_id
    input Node, StatusAddr, ResultAddr
    rule
        true ==>
            @ResultAddr' := getNodeId(Node);
            @StatusAddr' := "MCAPI_SUCCESS";
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !(\E [node, nid] in ActiveNodes: node = Node) ==>
            @StatusAddr' := "MCAPI_ENODE_NOTINIT";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end

## Endpoints
transition create_endpoint
    input Node, PortId, StatusAddr, ResultAddr
    rule
        true ==>
            @StatusAddr' := "MCAPI_SUCCESS";
            Endpoints' := Endpoints \U {newEndpoint(getNodeId(Node), PortId)};
            @ResultAddr' := getMaxEndpointId() + 1; #previous max + 1 is new EpId
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !ValidPortId(PortId) ==>
            @StatusAddr' := "MCAPI_EPORT_NOTVALID";

        ValidStatusParam(StatusAddr) /\
        (\E ep in Endpoints: [Node,ep_node_id(ep)] \in ActiveNodes /\ ep_port(ep) = PortId) ==>
            @StatusAddr' := "MCAPI_EENDP_ISCREATED";

        ValidStatusParam(StatusAddr) /\
        !(\E [node, nodeId] in ActiveNodes: node = Node) ==>
            @StatusAddr' := "MCAPI_ENODE_NOTINIT";

        ValidStatusParam(StatusAddr) /\
        false (# Max endpoints exceeded #) ==>
            @StatusAddr' := "MCAPI_EENDP_LIMIT";

        ValidStatusParam(StatusAddr) /\
        false (# can't make endpoints on this node #) ==>
            @StatusAddr' := "MCAPI_EEP_NOT_ALLOWED";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end
function ValidPortId(PortId)
    PortId >= 0
end

transition get_endpoint_i
    input Node, NodeId, PortId, EndpointAddr, RequestAddr, StatusAddr
    rule
        true ==>
            Requests' := Requests \U {newRequest("get_endpoint", Node,
                    [StatusAddr, NodeId, PortId, EndpointAddr])};
            @RequestAddr' := getMaxRequestId() + 1;
            @StatusAddr' := "MCAPI_SUCCESS";
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !ValidNodeId(NodeId) ==>
            @StatusAddr' := "MCAPI_ENODE_NOTVALID";

        ValidStatusParam(StatusAddr) /\
        !ValidPortId(PortId) ==>
            @StatusAddr' := "MCAPI_EPORT_NOTVALID";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end
function ValidNodeId(NodeId)
    NodeId >= 0
end
transition get_endpoint
    input Node, NodeId, PortId, StatusAddr, ResultAddr
    let EndpointAddr = ResultAddr #alias name for clarity.

    rule
        true ==>
            tmp RequestAddr;
            tmp SizeAddr;
            tmp Result2Addr;
            call get_endpoint_i(Node, NodeId, PortId, EndpointAddr, RequestAddr, StatusAddr) {
                @StatusAddr = "MCAPI_SUCCESS" ==>
                    call wait(Node, RequestAddr, SizeAddr, StatusAddr, 0(#timeout#), Result2Addr);
                @StatusAddr != "MCAPI_SUCCESS" ==>
                    let nop = 0;
            };
    end
end

daemon _finish_get_endpoint_i
    let reqs = {req in Requests :
               req_valid(req) = true
            /\ req_status(req) = "Pending"
            /\ req_type(req) = "get_endpoint"
            /\ (\E ep in Endpoints : ep_node_id(ep) = req_data(req).1 /\ ep_port(ep) = req_data(req)
                .2)
            }
    let EpAddr = req_data(req).3
    let StatusAddr = req_data(req).0
    rule
        reqs != {}
        ==>
            choose req in reqs;
            @EpAddr' := ep_id( (ep in Endpoints : ep_node_id(ep) = req_data(req).1 /\ ep_port(ep) =
                req_data(req).2) );
            Requests' := (Requests \ {req}) \U { [req.0, req.1, "Finished", req.3, req.4, req.5] };
            @StatusAddr' := "MCAPI_SUCCESS";
    end
end
transition delete_endpoint
    input Node, Endpoint, StatusAddr
    let endp = (ep in Endpoints : ep_id(ep) = Endpoint)
    rule
        true ==>
            Endpoints' := Endpoints \ {endp};
            @StatusAddr' := "MCAPI_SUCCESS";
    end
    errors
        ValidStatusParam(StatusAddr) /\
        endp = ERROR ==>
            @StatusAddr' := "MCAPI_ENOT_ENDP";

        ValidStatusParam(StatusAddr) /\
        getNode(ep_node_id(endp)) != Node ==>
            @StatusAddr' := "MCAPI_ENOT_OWNER";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end

## Endpoint Attributes
transition get_endpoint_attribute
    input Node, Endpoint, AttributeNum, AttributeAddr, AttributeSize, StatusAddr
    let ExistingValue = {[eid, attrn, val] in AssignedAttributes:
            [eid, attrn, val] \in AssignedAttributes
            /\ eid = Endpoint
            /\ attrn = AttributeNum}
    rule
        true ==>
            @AttributeAddr' := if ExistingValue = {} then 0 else ExistingValue fi;
            @StatusAddr' := "MCAPI_SUCCESS";
    end
    errors
```

```
        ValidStatusParam(StatusAddr) /\
        !ValidEndpointId(Endpoint) ==>
            @StatusAddr' := "MCAPI_ENOT_ENDP";

        ValidStatusParam(StatusAddr) /\
        false (#Unknown attribute number#) ==>
            @StatusAddr' := "MCAPI_EATTR_NUM";

        ValidStatusParam(StatusAddr) /\
        false (#incorrect attribute size#) ==>
            @StatusAddr' := "MCAPI_EATTR_SIZE";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end
function ValidEndpointId(EndpointId)
    if EndpointId >= 0
       /\ (\E ep in Endpoints : ep_id(ep) = EndpointId)
    then true
    else false fi
end

transition set_endpoint_attribute
    input Node, Endpoint, AttributeNum, Attribute, AttributeSize, StatusAddr
    let ExistingValue = {[eid, attrn, val] in AssignedAttributes:
            [eid, attrn, val] \in AssignedAttributes
            /\ eid = Endpoint
            /\ attrn = AttributeNum}
    rule
        true ==>
            AssignedAttributes' := (AssignedAttributes \ ExistingValue) \U {[Endpoint, AttributeNum,
                  Attribute]};
            @StatusAddr' := "MCAPI_SUCCESS";
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !ValidEndpointId(Endpoint) ==>
            @StatusAddr' := "MCAPI_ENOT_ENDP";

        false (#Unknown attribute number#) ==>
            @StatusAddr' := "MCAPI_EATTR_NUM";

        false (#incorrect attribute size#) ==>
            @StatusAddr' := "MCAPI_EATTR_SIZE";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...

        false (#endpoint is connected#) ==>
            @StatusAddr' := "MCAPI_ECONNECTED";

        false (#read only attribute#) ==>
            @StatusAddr' := "MCAPI_EREAD_ONLY";
    end
end

## Message passing
transition msg_send_i
    input Node, SendEndpoint, ReceiveEndpoint, BufferAddr, BufferSize,
            Priority, RequestAddr, StatusAddr
    rule
        true ==>
            Requests' := Requests \U {newRequest("msg_send", Node,
                [SendEndpoint, ReceiveEndpoint, BufferAddr, BufferSize, Priority])};
            @RequestAddr' := getMaxRequestId() + 1; #return the request id, serves as handle
            @StatusAddr' := "MCAPI_SUCCESS";        #Means msg is queued to send
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !ValidEndpointId(SendEndpoint) \/ !ValidEndpointId(ReceiveEndpoint) ==>
            @StatusAddr' := "MCAPI_ENOT_ENDP";

        ValidStatusParam(StatusAddr) /\
        BufferSize > _MaxMsgSize ==>
            @StatusAddr' := "MCAPI_EMESS_LIMIT";

        false (# no more buffers #) ==>
            @StatusAddr' := "MCAPI_ENO_BUFFER";

        false (# no more request handles #) ==>
```

```
            @StatusAddr' := "MCAPI_ENO_REQUEST";

        false (# out of memory #) ==>
            @StatusAddr' := "MCAPI_ENO_MEM";

        ValidStatusParam(StatusAddr) /\
        !ValidPriority(Priority) ==>
            @StatusAddr' := "MCAPI_EPRIO";

        ValidStatusParam(StatusAddr) /\
        !ValidRequestParam(RequestAddr) ==>
            @StatusAddr' := "MCAPI_EPARAM";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
    end
end
function ValidPriority(priority)
    if priority >= 0 /\ priority < 1024 #TODO: what is a valid priority???
    then true
    else false fi
end
state
    _MaxMsgSize = 100000000000
end

transition msg_send
    input Node, SendEndpoint, ReceiveEndpoint, BufferAddr, BufferSize,
            Priority, StatusAddr
    rule
        true ==>
            tmp RequestAddr;
            tmp SizeAddr;
            tmp ResultAddr;
            call msg_send_i(Node, SendEndpoint, ReceiveEndpoint, BufferAddr, BufferSize,
                    Priority, RequestAddr, StatusAddr) {
                @StatusAddr = "MCAPI_SUCCESS" ==>
                    call wait(Node, RequestAddr, SizeAddr, StatusAddr, 0(#timeout#), ResultAddr);
                @StatusAddr != "MCAPI_SUCCESS" ==>
                    let nop = 0; #already is returning error code
            };
    end
end
daemon _finish_msg_send_i
    let reqs = {req in Requests :
                       req_valid(req) = true
                    /\ req_status(req) = "Pending"
                    /\ req_type(req) = "msg_send"
                    /\ !(
                        (\E r in Requests:
                           req_type(r) = "msg_send"
                        /\ req_status(r) = "Pending"
                        /\ req_valid(r)
                        /\ req_data(r).0 = req_data(req).0  #Sending from the same place
                        /\ req_data(r).1 = req_data(req).1  #and to the same place
                        /\ req_id(r) < req_id(req))         #and this one was sent first
                    )
                }
    let RecvEndp = (ep in Endpoints : ep_id(ep) = req_data(req).1) #Find the receiving endpoing
    let RecvEndpQueue = ep_msgs(RecvEndp) #we'll need to append to this queue (functionally)
    let sendBuf = req_data(req).2         #we'll find the message to send here
    let msgSize = req_data(req).3
    rule
        reqs != {}  #Has at least one pending request to satisfy
        ==>
            choose req in reqs;
            Endpoints' := (Endpoints \ {RecvEndp}) \U
                    {changeEpQueue(RecvEndp, fifo_add(RecvEndpQueue, [@sendBuf, msgSize]))};
            Requests' := (Requests \ {req}) \U { [req.0, req.1, "Finished", req.3, req.4, req.5] };
    end
end
transition msg_recv_i
    input Node, RecvEndp, BufferAddr, BufferSize, RequestAddr, StatusAddr
    rule
        true ==>
            Requests' := Requests \U {newRequest("msg_recv", Node,
                    [RecvEndp, BufferAddr, BufferSize])};
            @RequestAddr' := getMaxRequestId() + 1; #return the request id, serves as handle
            @StatusAddr' := "MCAPI_SUCCESS";        #Means recv_i is queued, no data yet
    end
    errors
```

```
        ValidStatusParam(StatusAddr) /\
        !ValidEndpointId(RecvEndp) ==>
            @StatusAddr' := "MCAPI_ENOT_ENDP";

        false (# message size exceeds BufferSize #) ==>
            @StatusAddr' := "MCAPI_ETRUNCATED";

        false (# no more request handles #) ==>
            @StatusAddr' := "MCAPI_ENO_REQUEST";

        ValidStatusParam(StatusAddr) /\
        !ValidRequestParam(RequestAddr) \/ !ValidBufferParam(BufferAddr)  ==>
            @StatusAddr' := "MCAPI_EPARAM";

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...

        ValidStatusParam(StatusAddr) /\
        ep_node_id(getEndpoint(RecvEndp)) != getNodeId(Node) ==>
            @StatusAddr' := "MCAPI_ENOT_ENDP";
    end
end
transition msg_recv
    input Node, ReceiveEndpoint, BufferAddr, BufferSize, RecvSizeAddr, StatusAddr
    rule
        true ==>
            tmp RequestAddr;
            tmp ResultAddr;
            call msg_recv_i(Node, ReceiveEndpoint, BufferAddr, BufferSize,
                    RequestAddr, StatusAddr) {
                @StatusAddr = "MCAPI_SUCCESS" ==>
                    call wait(Node, RequestAddr, RecvSizeAddr, StatusAddr, 0(#timeout#), ResultAddr);
                @StatusAddr != "MCAPI_SUCCESS" ==>
                    let nop = 0; #already is returning error code
            };
    end
end
daemon _finish_msg_recv
    let Endp = (ep in Endpoints : ep_id(ep) = req_data(req).0 )
    let EndpQueue = ep_msgs(Endp)
    let recvBuf = req_data(req).1
    let recvBufSize = req_data(req).2
    let reqs = {req in Requests :
                       req_valid(req) = true
                    /\ req_status(req) = "Pending"
                    /\ req_type(req) = "msg_recv"
                    /\ !(
                        (\E r in Requests:
                           req_type(r) = "msg_recv"
                        /\ req_status(r) = "Pending"
                        /\ req_valid(r)
                        /\ req_data(r).0 = req_data(req).0  #Receiving on the same endpoint
                        /\ req_id(r) < req_id(req))         #and this one was registered first
                    )
                    /\ fifo_hasNext(EndpQueue) #and has a message waiting to receive
               }
    rule
        reqs != {} #has at least one request to process
        ==>
            choose req in reqs;
            let msg = fifo_next(EndpQueue);
            @recvBuf' := truncate(msg.0, recvBufSize);  #copy msg
            Endpoints' := (Endpoints \ {Endp}) \U
                    {changeEpQueue(Endp, fifo_remove(EndpQueue))}; #remove msg from queue
            Requests' := (Requests \ {req}) \U {
                    [req.0, req.1, "Finished", req.3, req.4,
                        [req.5.0,
                         req.5.1,
                         #Change the size to the actual message size, so we can report it
                         min(recvBufSize, msg.1)]]
                };
    end
end

## Non-blocking operations
transition wait
    input Node, RequestAddr, SizeAddr, StatusAddr, Timeout, ResultAddr
    let req = getRequest(@RequestAddr)
    rule
        req_type(req) \notin {"msg_send","msg_recv"} /\ req_status(req) = "Finished" ==>
```

```
            @ResultAddr' := true;
            @StatusAddr' := "MCAPI_SUCCESS";

        # For Send / Recv, we also respond with the buffer size specified
        req_type(req) = "msg_send" /\ req_status(req) = "Finished" ==>
            @SizeAddr' := req_data(req).3; #request has the message size here
            @ResultAddr' := true;
            @StatusAddr' := "MCAPI_SUCCESS";

        req_type(req) = "msg_recv" /\ req_status(req) = "Finished" ==>
            @SizeAddr' := req_data(req).2; #the size in the req was updated to actual message size
            @ResultAddr' := true;
            @StatusAddr' := "MCAPI_SUCCESS";
    end
    errors
        ValidStatusParam(StatusAddr) /\
        !hasRequest(@RequestAddr) ==>
            @StatusAddr' := "MCAPI_ENOTREQ_HANDLE";
            @ResultAddr' := false;

        ValidStatusParam(StatusAddr) /\
        req_status(req) = "Canceled" ==>
            @StatusAddr' := "MCAPI_EREQ_CANCELED";
            @ResultAddr' := false;

        ValidStatusParam(StatusAddr) /\
        Timeout != "MCAPI_INFINITE" /\ false (# timeout #) ==>
            @StatusAddr' := "MCAPI_EREQ_TIMEOUT";
            @ResultAddr' := false;

        ValidStatusParam(StatusAddr) /\
        SizeAddr = 0 ==>
            @StatusAddr' := "MCAPI_EPARAM";
            @ResultAddr' := false;

        !ValidStatusParam(StatusAddr) ==>
            ErrorStatus' := "MCAPI_EPARAM"; #Can't actually report this...
            @ResultAddr' := false;
    end
end
```

References

J. R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-49619-5.

David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

Mohamed Elwakil and Zijiang Yang. CRI: Symbolic debugger for mcapi applications. In *The 8th International Symposium on Automated Technology for Verification and Analysis (LNCS)*, 2010a.

Mohamed Elwakil and Zijiang Yang. Debugging support tool for mcapi applications. In *PADTAD '10: Proceedings of the 8th Workshop on Parallel and Distributed Systems*, 2010b.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 2009. ISBN 0262062755, 9780262062756.

Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

Philippe Georgelin, Laurence Pierre, and Tin Nguyen. A formal specification of the MPI primitives and communication mechanisms. Technical report, LIM, 1999.

Patrice Godefroid. Model checking for programming languages using Verisoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13:289–307, November 1998. ISSN 0925-9856. doi: 10.1023/A:1008696026254. URL `http://portal.acm.org/citation.cfm?id=303322.303329`.

Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, April 2006. ISBN 0262101149.

Leslie Lamport. TLA - the temporal logic of actions. `http://research.microsoft.com/users/lamport/tla/tla.html`.

Guodong Li, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of the MPI-2.0 standard in tla+. In *PPoPP: In Proceedings of the ACM SIGPLAN Symposium onPrinciples and Practices of Parallel Programming*, pages 283–284, 2008.

Guodong Li, Ganesh Gopalakrishnan, Robert M. Kirby, and Dan Quinlan. A symbolic verifier for CUDA programs. In *PPoPP: In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–358, 2010.

E. G. Mercer and M. Jones. Model checking machine code with the GNU debugger. In *12th International SPIN Workshop*, volume 3639 of *Lecture Notes in Computer Science*, pages 251–265, San Francisco, USA, August 2005. Springer.

M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI '07: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, New York, NY, USA, 2007. ACM.

Hanne Riis Nielson and Flemming Nielson. *Semantics with applications: a formal introduction.* John Wiley & Sons, Inc., New York, NY, USA, 1992. ISBN 0-471-92980-8.

Robert Palmer, Michael Delisi, Ganesh Gopalakrishnan, and Robert M. Kirby. An approach to formalization and analysis of message passing libraries. In *FMICS: Proceedings of the 12th Workshop on Formal Methods for Industrial Critical Systems*, pages 164–181, Berlin Heidelberg, 2007. Springer.

Terence J Parr and R W Quong. Antlr: A predicated-ll(k) parser generator. *Software–Practice and Experience*, 25:789–810, July 1995.

C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *ISSTA '08: Proceedings of the International Symposium on Software Testing and Analysis 2008*, pages 15–26, New York, NY, USA, 2008. ACM.

Subodh Sharma, Ganesh Gopalakrishnan, Eric Mercer, and Jim Holt. MCC: A runtime verification tool for mcapi user applications. In *Formal Methods in Computer-Aided Design*, pages 41–44, 2009.

Stephen F. Siegel and George Avrunin. Analysis of mpi programs. Technical Report UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst, 2003.

J. M. Spivey. The Z notation: a reference manual. *Prentice-Hall International Series In Computer Science*, page 155, 1989.

The Multicore Association. `http://www.multicore-association.org`.

Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. CAV '08: Computer-Aided Verification 2008, pages 66–79, Berlin, Heidelberg, 2008. Springer-Verlag.

Chao Wang, Yu Yang, Aarti Gupta, and Ganesh Gopalakrishnan. Dynamic model checking with property driven pruning to dectect data race conditions. In *ATVA: International Symposium on Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, Seoul, Korea, 2008. Springer.

Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 23–32, 2009.