Modeling Asynchronous Message Passing for C Programs

Everett Morse, Nick Vrvilo, Eric Mercer, and Jay McCarthy

Brigham Young University, Provo UT 84601, USA,
{eamorse,nick.vrvilo}@byu.net,
{egm,jay}@cs.byu.edu,
WWW home page: http://vv.cs.byu.edu

Abstract. This paper presents a formal modeling paradigm that is callable from C, the dominant language for embedded systems programming, for message passing APIs that provides reasonable assurance that the model correctly captures intended behavior. The model is a suitable reference solution for the API, and it 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 that use the API. This paper illustrates the modeling paradigm on the MCAPI interface, a growing industry standard message passing library, showing how the model exposes errors hidden by the C reference solution provided by the Multicore Association.

Keywords: Model Checking, Concurrency, Test, Debug, Validation

1 Introduction

Asynchronous message passing for C is important in writing applications for embedded heterogeneous multicore systems. The Multicore Association (MCA), an industry consortium promoting multicore technology, is working to standardize message passing into a single API, MCAPI, for bare metal implementation and portability across platforms [26]. The MCAPI specification is a 169 page document in English. The inherent vagueness of such a description is valuable because implementation details are not micro-managed by API designers. In other words, high-level properties of the API such as “atomic”, “blocking”, or “non-overtaking” are specified without detailed explanation of internal API state nor how they should be provided. Correctness in implementing and using such an API, however, is difficult to reason about manually.

It is not unusual to provide an initial API implementation (production or otherwise) with a natural language description of the interface and MCAPI is no different, providing a C implementation of the interface. Unfortunately, there are two issues with it: (1) it is implemented in a production language that is semantically distant from the natural language description so it is not clear which behaviors of the description it implements nor if it is correct; and (2) the reference is non-deterministic due to concurrency in the reference itself making test and

debug activities difficult. A reference implementation needs to be semantically near the natural language description, while still being formal, and it needs to be deterministic for test, debug, and exploration. Programmers must have have a way to directly control API internals to expose or reproduce errors.

There are several formal modeling languages with mathematically defined operational semantics. A few languages such as TLA also provide a general runtime implementation of the operational semantics [14]; though most only provide tools to verify properties of models expressed in the formal language [11, 17, 13, 22]. Regardless, the implementation of the operational semantics for these general languages is in a low-level language such as C introducing a significant gap between the mathematical expression of the semantics and the actual rendered implementation that is difficult to reason about. Additionally, for those that do implement a runtime for the formal language, there is no obvious way to connect that runtime to C programs written against the API. As such, these formal models are not suitable reference implementations for test and debug.

This paper presents a modeling language, 4M, for message passing APIs defined by natural language descriptions. The modeling language is sufficiently abstract to provide a reasonable assurance that the model correctly captures the intent of the natural language description. Furthermore, since 4M formally defines operational semantics, standard model checking or theorem proving techniques can be applied to prove the model implements the API specification (assuming such a specification exists). Novel to the 4M modeling language is a deterministic runtime, callable from C, derived from its operational semantics that is suitable for test, debug, scenario exploration, model checking, or other verification techniques. We demonstrate the methodology through a case study on the MCAPI communication library. The contributions of this work are

- a modeling language, 4M, implemented as a term rewriting system that is suited to natural language descriptions;
- a novel architecture to directly connect the C runtime to the 4M runtime to use the model as an instance of the API that is explorable, testable, and capable of model checking;
- an implementation of the rewriting system in Racket which a programming language based on PLT Scheme; and
- an MCAPI 4M model with running time results to measure overhead and bugs discovered from several C programs written against the MCAPI.

The result is that when an API is formally modeled in 4M, it is possible to use that model to explore system-wide program behavior that existing models and implementations cannot reason about.

Fig. 1 illustrates the methodology. 4M intuitively expresses the intent of the natural language API description (Fig. 1(a)). The core calculus describing the operational semantics of 4M is directly implemented by PLT Redex (Fig. 1(b)).[1] Programs using the API are developed in the C language as intended by the

[1] PLT Redex is a tool for creating and debugging language semantics defined as term rewriting systems, and it is part of the Racket runtime [7, 8].

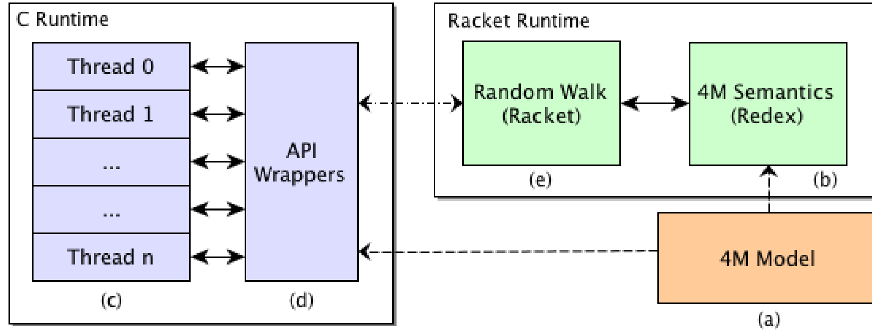

Fig. 1. Architecture for an API model callable from a native runtime.

API (Fig. 1(c)). Such a program calls C function stubs that define the API interface and communicate with the runtime implementation of the 4M semantics through a pipe (Fig. 1(d)). Program execution proceeds in a normal fashion. As each thread enters the API, the corresponding thread is blocked. When all of the threads are blocked in the API, the API then communicates with a search strategy to choose a next state, and returns to the C runtime. In the case study on MCAPI presented in this paper, the search strategy is a random walk or an exhaustive search. The user can make both of these deterministic for debug or replay by setting the random seed to a known value (Fig. 1(e)).[2]

The following sections describe this process and our contributions in detail: Sec. 2 informally presents 4M on a toy message passing library;[3] Sec. 3 presents the novel client-server architecture that bridges the C runtime to the 4M core runtime; Sec. 4 presents our study on MCAPI; Sec. 5 addresses specific related work to this research; And Sec. 6 concludes and presents future work.

2 Modeling with 4M

Fig. 2(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 natural language specification defines transitions with their input, effects, and error conditions, which are helpful properties in understanding individual API behavior in isolation.

It is a challenge to explain intended behavior in simple scenarios consisting of a handful of calls when dealing with APIs for concurrent programs. Such scenarios are often created by adopters or implementers of the API to reason

[2] The random seed is provided as output from the tool and can be specified as part of the run configuration for a test.

[3] More details on the 4M language with its core operational semantics and programming framework are in [1].

```
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 frm, msg_t *msg, mbox_t to)
```
Description: Sends the message `msg` from the mailbox `frm` to the mailbox `to`. It is a blocking function and returns once the buffer `msg` can be reused by the 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, &s)`	`to0 = get_mbox(0, &s)`
	`from1 = mbox(3, &s)`	`to1 = get_mbox(1, &s)`
`recv(to0,&a)`	`recv(to1,&c)`	`send(from2,"Y",to0)`
`recv(to0,&b)`	`send(from1,"X",to0)`	`send(from2,"Z",to1)`

(b)

Fig. 2. A simple message passing API. (a) The natural language description of the API. (b) A scenario written over the API.

about the expected API behavior relative to its documentation. Most often these scenarios assist in understanding the API behavior and are used to convey that understanding to the broader community. Consider the scenario in Fig. 2(b) that includes three threads using the blocking send (`send`) and receive (`recv`) calls from the API to communicate with each other. The declarations of the local variables (e.g., *to0*) are omitted for space. Picking up 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. After the scenario, we may ask: *"Which messages may be in which variables?"*

Intuitively, variable a should contain "Y" and variable b should contain "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 a correct program execution, but it is not

the only execution, since the specification allows an alternative scenario where message "Y" is delayed in transit and arrives at mailbox *to0* after message "X".

The natural language description in Fig. 2(a) states that the send operation *"returns once the buffer* `msg` *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 program execution places the message "X" in variable a and the message "Y" in variable b.

Nuances like this are discovered in the process of concretizing internal API details, leading the modeler to engage in an iterative process with API designers which brings value to both by clarifying the semantics of the API. The specification in Fig. 2(a) is a simplified subset of a real communications API from the MCA (MCAPI). Conversations with the MCAPI designers confirmed the intended behavior of the API to include both program executions of the scenario. To date, there have been three published verification and analysis tools purpose-built for MCAPI and all of them omit the less intuitive program execution [23, 6, 5]. Naturally, our model does not omit it.

2.1 Formal Model of the API

There are several languages one might consider in modeling an API (see Sec. 5). Recent attempts to model MPI in the formal logic of TLA have shown the logic to be too low-level for practical application [14, 20]. Alternatively, when considering a direct implementation such as one in C, not only is the gap between the natural language description and C extremely difficult to bridge, for example, the MCAPI reference solution includes 11776 lines of code to consider, it is not easy to test in the presence of concurrency because a user cannot readily control execution schedules. Moreover, C is unusually susceptible to bugs as evidenced by our experience with the MCAPI reference implementation which non-deterministically deadlocks.

We propose 4M, which matches the natural language description, as opposed to most existing modeling languages that do not. Furthermore, the intent of 4M is not to embed verification assertions such as guaranteed message delivery into the model. Rather, those types of correctness properties should be expressed in an appropriate logic and used to verify the correctness of the model. The goal is to capture the natural language description in an operational model. While 4M is yet another modeling language, it is domain specific, rather than general purpose, making it more amenable to the task at hand.

4M is a formal modeling language designed to keep the best things from the natural written style and remove the worst. To be specific, 4M keeps the structure of the natural language specification that defines transitions with their input, effects, and error conditions, but it replaces the statements such as *"message non-overtaking"* 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 our toy API is given in Fig. 3. This model is the input in Fig. 1(a) of our proposed solution. 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 with always enabled daemon transitions in lines 43–53 to manage internal state.

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

The 4M language has a first order treatment of errors in any given transition. The language is designed for natural language description that presents a transition's normal behavior followed by a set of possible errors. An error or rule is enabled when its guard is true. The semantics of 4M block a transition until a guard becomes true (rule or error), and give preference to error 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 duplicates an existing mailbox. The dot notation in `box.0` of the guard implies that `mailboxes` is a set of tuples, and the notation is comparing the first member of each tuple to `id`. 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 the value `-1`. 4M does not support unbounded non-determinism so integers range over a bounded set using modular arithmetic. The '`@`' symbol is the dereference and the apostrophe indicates the next value. Evaluation of guards and application of the effect is one atomic step.

The rules 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 store 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. All of this occurs as one atomic step.

The other transitions `get_mbox`, `send`, and `recv` are defined like `mbox`. The transition `recv`, which blocks in the absence of a message, is protected by the guard on line 38 that is only satisfied if the memory referenced by variable `to` is not empty (i.e., when a message is pending). In the rule body, the 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

```
1  state
2   mailboxes = {}
3   queues = 0
4  end
5  transition mbox
6   input id, statusAddr, resultAddr
7   rules
8    true ==>
9     tmp newAddr;
10    @resultAddr' := newAddr;
11    mailboxes' := mailboxes \U {[id, newAddr]};
12    @statusAddr' := 1;
13   end
14  errors
15   (\E box in mailboxes: box.0 = id) ==>
16    @statusAddr' := -1;
17  end
18 end
19 transition get_mbox
20  input id, resultAddr
21  rules
22   (\E box in mailboxes: box.0 = id) ==>
23    let mailbox = (box in mailboxes: box.0 = id);
24    @resultAddr' := mailbox.1;
25  end
26 end
27 transition send
28  input from, msg, to
29  rules
30   true ==>
31    queues' := [from, queues];
32    @from' := [@msg, to, @from];
33  end
34 end
35 transition recv
36  input to, msg
37  rules
38   @to != 0 ==>
39    @msg' := (@to).0;
40    @to' := (@to).1;
41  end
42 end
43 daemon pump
44  rules
45   queues != 0 ==>
46    let from = queues.0;
47    let msg = (@from).0;
48    let to = (@from).1;
49    @to' := [msg, @to];
50    @from' := (@from).2;
51    queues' := queues.1;
52  end
53 end
```

Fig. 3. A simplified message-passing API in 4M

infinitely often in the API, executed as often as the guards are enabled, and represent a concurrent thread of execution. 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). All of this occurs as one atomic step.

2.2 Semantic Implementation of 4M

4M is intended for human consumption with a form and semantics that are non-trivial to define. For example, 4M gives 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. The nuances of this semantics are more easily realized by a simpler core calculus.

The operational semantics for the 4M core is given by a term rewriting system employing small-step semantics through continuations. The 4M core is mathematically defined in [1]. The novelty in the 4M semantics is the layering of machines to isolate non-determinism in a single machine. The 4M language itself is not terribly unique and rather its contribution lies more in the technique in creating a domain specific language to model a system.

The implementation of the semantics corresponds to Fig. 1(b) in our architecture for API modeling. Questions regarding API behavior over concurrent calls such as the scenario in Fig. 2(b) can be explored directly in the 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.

For example, consider the API state and the thread states shown in Fig. 4 for our 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`. The top portion of the figure shows the state of each thread with its local variables and the current program location indicated by the •-mark. The local variables hold references into the API state for each of the mailboxes created in the scenario.

The API state in the bottom portion of the figure is: `mailboxes`, that associates an ID with a memory reference that is the actual mailbox; `queues`, a list tracking undelivered messages; and the mailboxes themselves with their contents. From Fig. 4, the scenario has created four mailboxes, and mailbox 2, located at (addr 7), has the pending message "`Y`" to be delivered to (addr 5). The zero entry in the tuple indicates the end of the list (i.e., there is only one pending message). The `queues` variable indicates that the message from (addr 7) needs to be delivered (by the `pump` daemon in the 4M model of Fig. 3).

The semantics allows several next states from the state in Fig. 4 such as the `pump` transition moving the message out of the `from2` mailbox–(addr 7)–into the `to0` mailbox–(addr 5)–or adding the next send from thread 2 into the `queue` and the `from2` mailbox–(addr 7). A test is able to trace any possible execution from the current API state by randomly picking a transition allowed by the semantics.

Thread 0	Thread 1	Thread 2
s: 0 to0: (addr 5) a: b:	s: 0 to0: (addr 5) to1: (addr 6) from1: (addr 8) c:	s: 0 to0: (addr 5) to1: (addr 6) from2: (addr 7)
•recv(to0,&a) recv(to0,&b)	•recv(to1,&c) send(from1,"X",to0)	•send(from2,"Z",to1)
API Global State		
mailBoxes–*[id,ref]*	[0, (addr 5)] [2, (addr 7)]	[1, (addr 6)] [3, (addr 8)]
queues–*[ref,queues]*	[(addr 7), 0]	
(addr 5)–*mailbox 0*		
(addr 6)–*mailbox 1*		
(addr 7)–*mailbox 2*	("Y", (addr 5), 0)	
(addr 8)–*mailbox 3*		

Fig. 4. The state of the threads and API for Fig. 3 and Fig. 2(b) where the threads have run until thread 0 and thread 1 are blocking (indicated by the •-mark) at the receives and thread 2 is attempting its last send.

3 4M Implementation

3.1 Reference Solution for Test, Debug, and Behavior Exploration

Manually writing the state of the API for the 4M core and manually stepping through the semantics definition is not feasible. Suppose instead that there exists an actual implementation of the 4M core that captures precisely the operational semantics. Naturally, it would be ideal to take a C program using the API, similar to the definition of thread 0 in Fig. 5(a), and connect it directly to the 4M core implementation to simulate the API behavior.

We provide such a connection. It is implemented by a role-based relationship between the C runtime and the 4M core implementation runtime, which we call the GEM (Golden Executable Model). Thin wrappers bridge the API calls to the actual C code as shown in Fig. 5(b). These correspond to Fig. 1(d) of our solution. The `gem_call` is the entry to the model of the API. The GEM implementation itself blocks waiting for all threads to invoke the API at which point it communicates with the 4M core implementation to send the states of the active threads. The component representing the search strategy corresponds to Fig. 1(e) of our solution. The search strategy determines priority in the search order of possible next states and can be customized by the user. The 4M implementation, corresponding to Fig. 1(b) of our solution, returns a possible next state, and the model releases the corresponding blocked `gem_call` for the stopped thread. The thread then continues until the next API entry occurs to repeat the process. The model stores a random seed from the execution for reproducibility.

Our architecture for a model replacement of APIs in C programs is divided into three different components: an implementation of 4M, a mechanism for capturing C API calls, and a strategy for exploring the possible system states (see Fig. 1). These components are connected as follows:

1. A driver process spawns the GEM server and GEM client processes and creates their inter-process communication pipes.

```
1  void t0() {                          void send(mbox_t f,msg_t* b,mbox_t t) {
2    msg_t a, b;    status_t s;           gem_var bv;
3    mbox_t to0 = mbox(0, &s);            bv=init_var(b,buf_len(b),GEM_STRING);
4    recv(to0, &a);                       gem_call("send␣(%v␣%v␣%v)",f, bv,t);
5    recv(to0, &b);                       del_var(bv);
6  }                                    }
```

(a) (b)

Fig. 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.

2. As long as there are threads that are not terminated or blocked on API calls, the GEM client runs the user program using a cooperative threading model, executing threads one at a time.
3. As the GEM client makes API requests, the GEM server responds to each one and synchronizes its API state with that of the GEM client.
4. The 4M API model generates a list of possible next states given the information it has received about the threads and the blocked API call for each thread. A next state determines which blocking API call will finish. The GEM server randomizes the list of possible states and designates the first state as the one to be explored. To ensure deterministic behavior, the random seed used for the random walk can be set by the user.
5. Once a next state has been selected, the state change is synchronized with the GEM client and the corresponding threads are unblocked.
6. Steps 2 - 5 are repeated until program termination.

The underlying assumptions for correctness is that i) a thread eventually enters the API, even if it enters through an explicit call to exit the thread, at which point we can ignore it forever; and ii) there is no other non-determinism in the system (i.e., no I/O etc.). The first point (i) is needed to return control to the API model and indicate when it is time to compute a next state (i.e., all of the threads have arrived); otherwise, the model does not know if it should continue to wait for a thread to enter the API or compute a next state. The second point (ii) is important for replay.

The steps described represent the execution of a single pathway of the user program. In order to perform an exhaustive exploration, our tool implements a rewind and replay mechanism. When API calls are made by the GEM client on behalf of the user program, the responses returned by the GEM server and API are recorded in a file. This logging enables a zero-calculation replay of the user program up to the point where a new next state is to be explored. Again, we currently restrict out all other sources of non-determinism in the program in order for the replay to work correctly (i.e., I/O etc. must be deterministic for replay). The GEM client merely reads logged responses from the pipe rather than reading live responses. During the replay phase, the GEM server ignores any requests sent by the GEM client. The user program is not rerun in its entirety. It is instead run to a point decided by the GEM server (i.e., the point at which is new or different next state is to be considered).

```
1  mbox_t mbox(int id, status_t* s) {
2    mbox_t res;
3    gem_var sv, rv;
4    sv = reg_var(s, sizeof(int));
5    rv = reg_var(&res, sizeof(mbox_t));
6    gem_call("mbox␣(%d␣%v␣%v)",id,sv,rv);
7    del_var(sv); del_var(rv);
8    return res;
9  }
```

Fig. 6. An example wrapper for the `mbox` function.

The immutable and recursive characteristics of functional programs afford the GEM server some abilities not easily mirrored in the GEM client. In particular, they enable the server to "rewind" itself to an earlier program state by simply returning up the execution stack. We utilize this feature to exhaustively explore execution paths. Picking up just after step 6 above:

1. The GEM server checks if all possible paths have been explored.
2. If unexplored paths exist, the GEM server rewinds to the point where it last selected a next state from the list of next states given by the API model. If all states in the list have already been explored, the server is instead rewound to the latest point where there still exists unexplored next states.
3. The GEM client is told to replay the user program.
4. The GEM server waits for the GEM client to replay. The responses recorded from the last execution are sent to the client so it may replay to the execution point where the GEM server is waiting.
5. Normal execution continues, but this time the GEM server selects the first *unexplored* next state from the list.
6. When all paths have been explored, the tool terminates.

Following are some of the finer details of the C Wrappers. Fig. 6 shows an example wrapper that demonstrates the issues each wrapper must manage:

- The wrapper must match the API interface to be a suitable model. (Line 1)
- As the C runtime and 4M runtime communicate through a pipe, we must use an external representation of values. (Lines 2 and 3)
- Some parameters may be pointers to C memory (such as `s`), so the wrapper must allocate a 4M location (implemented by `reg_var`) for it. (Line 4)
- Similarly, 4M transitions do not have "return values"—instead a return is accomplished by passing a reference that gets updated. This encoding is managed by the wrapper by allocating a 4M location. (Lines 2 and 5)
- Since the 4M runtime simply waits for API calls to execute, the wrapper must marshal each call to 4M. This process entails encoding parameters as 4M values. Line 3 prepares references, then lines 4 and 5 use the function `reg_var` to associate the C memory references with 4M store locations. The `gem_call` function (line 6) automatically expands its arguments to the correct 4M encoding based on the placeholders in the format string. More complicated data conversion may be necessary where C datatypes do not match the 4M

core datatypes: 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.
– Inside `gem_call`, the 4M runtime takes over and can delay arbitrarily long until the result of the API call is computed by 4M and the search strategy.
– Once `gem_call` completes, the C memory locations associated with 4M store locations (as established in lines 4 and 5) are updated with their new values, and the result is returned (line 8).

In summary, 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 model where the state capture component takes over. Once the next state has been computed and reified into the C program, the model returns control to the wrappers.

Our 4M implementation is written in PLT Redex [7], a domain-specific language that ships with Racket [8] for encoding operational semantics as rewriting systems.[4] We employed PLT Redex for its development environment which provides a richer set of test and debug tools than say Maude, another term rewriting system. Further, PLT Redex is tightly integrated with Racket letting us embed arbitrary Racket code into the term rewriting system. Such integration is useful as some transformations over the machine state are more naturally expressed in Racket than in PLT Redex. As 4M is defined in machine semantics as a term rewriting system, the encoding in PLT Redex is obvious.

4 MCAPI Model Results

We validate our process on the connectionless message passing portion of the MCAPI communications library [26]. 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 model comprises 488 lines of code utilizing 3 daemon transitions for internal processing which is quite small compared to the roughly 30 pertinent pages of the 169 page English description. The API state itself only contains 4 unique variables. The 4M model compiles into 284 lines of the 4M core calculus.

As there is no "formalism" of the MCAPI API to which we can relate our 4M model, we validate our model through empirical test. Specifically, we have developed a suite of API scenarios (i.e., regression tests) for which we have validated with the API designers the possible outcomes. We run each of these scenarios through our API model ensuring that our model captures all the allowed behaviors specified in the scenarios. In the end, we have no definitive test that proves our model more correct than say the C reference implementation; however, as

[4] We use an unpublished compiler for PLT Redex that drastically improves performance by specializing Redex to deterministic reduction semantics where at most one reduction is applicable. With this new compiler, an exhaustive test that takes 12 minutes on the stock system completes in a few seconds.

Benchmark	Lines	API Calls	Paths	Run Time
Self Send	42	6	1	3.969s
Topher Scenario	128	18	27	6.595s
Leader	168	24	42	13.487s

Table 1. Benchmark Runtimes

our model is not so semantically apart from the documentation as say the C reference implementation, we subjectively have a greater assurance (or at least we are more able to convince ourselves) that our model is correct. In other words, it is much easier to argue through inspection that our model implements the API than it is to argue similarly that the C reference solution implements the API.[5]

To quantify the overhead in the model, we report running times on several examples as measured with the Unix `time` command on an Intel Core 2 Quad 2.66 GHz machine with 8 GB of memory running Fedora 14 as well as the number of bytes sent through the pipes. Unfortunately, the tested examples are all inhouse as there is no MCAPI code in the wild, to the knowledge of the authors, at the time of writing. Running the scenario in Fig. 2(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 3 seconds (2.6 of which is starting the Racket server). This overhead in starting the server is mitigated in longer running programs. In the C execution, ∼2KB are communicated between the C runtime and the Racket server. The C runtime spends 403ms waiting for the Racket runtime (22ms on an average call). An insignificant amount of time is spent preparing, sending, and parsing IPC messages. Clearly this would grow with the size of the scenario and the size of the API state.

As a reference point, the running time for the MCAPI dynamic verifier MCC on a scenario with 3 threads, two performing parallel sends, and the third making two sequential receives is under 1 second [23], whereas in our implementation it takes 2.9 seconds total (2.6s to start the server and 0.3s to compute.) Recall that the MCC tool relies critically on a reference implementation that, as discussed previously, is buggy and does not include all the behavior allowed in the API. As a note, our 4M model can be a drop-in replacement for the reference implementation in the tool as our model provides the exact interface.

We implemented Dijkstra's self-stabilization algorithm [3] in C using MCAPI. This algorithm runs in 8.2 seconds at $n = 4$. Of this 8.2s, 5.5 was spent in the 4M implementation and 2.6 starting the Racket server. During the execution, ∼14KB of data is communicated to Racket and ∼3KB from Racket. This suggests that real programs do incur significant overhead using 4M but can still run feasibly. With $n = 6$, the algorithm completes in 35s. Table 1 summarizes the results obtained from other MCAPI benchmark programs with our tool.

As expected, the exhaustive search using our 4M model found both executions in the example scenario of Fig. 2(b) which is the *Topher Scenario* in the table. The non-intuitive scenario triggers an assertion violation in the test harness. In

[5] See [1] for the MCAPI 4M model and the 4M tool set.

addition to these MCAPI benchmarks, we converted a control program for an amusement park used in an operating system class to MCAPI. The program is 1,192 lines of C code, creates 45 distinct threads, and issues thousands of MCAPI calls. The program has been run hundreds of times on the MCA reference solution and has never failed. When using the 4M model, the program immediately failed. Further inspection revealed three distinct race conditions latent in the code that can only be realized by specific message orderings allowed by the specification but not present in the MCA reference implementation.

5 Related Work

Verifying concurrent systems has long be a topic of active research. There are several modeling and specification languages with complete frameworks for analysis and model checking. These include Promela, Murphi, TLA+, Z, Alloy, and B, to name a few [11, 4, 14, 25, 12, 2]. 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 term rewriting systems expressed in PLT Redex are expressed naturally in mathematical notation; thus there is a reasonable assurance that the model runtime corresponds directly to the mathematical expression of the semantics. And second, the other solutions target analysis in the model's runtime whereas our work is intended as a model of the API that serves as a replacement for the actual API implementation when testing and debugging applications written against the API in the intended target language.

There have been attempts to model MPI in extant specification languages including conversion from C programs using MPI to the specification language [24, 9, 20, 15]. Recent work takes CUDA and C to SMT languages [29, 16, 6, 5]. Such implementations are only suitable to scenario evaluation and not drop-in replacement. They must also prove a correct translation to the analysis language.

Recent work in dynamic verification uses the program directly with the actual API implementation to perform model checking (i.e., the API implementation serves as the API model itself) [10, 18, 21, 19, 30, 27, 31, 28]. Although search through continuations rather than repeated program invocation is similar to [18, 21], 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, and even then, on just those implementation aspects that are controllable. For example, it is not possible to affect arbitrary buffering in MPI or MCAPI runtime libraries and as a result, behaviors such as those in our example scenario are omitted in the analysis [23]. Some recent work can test threaded or distributed libraries written against POSIX or Windows APIs, exploring all possible execution paths in the implementation itself, but they depend on a specific implementation [31, 28].

6 Conclusion and Future Work

English specification of concurrent APIs catalog interfaces and list effects of correct and incorrect calls to those interfaces. Unfortunately, they provide no framework with which a programmer or designer might experiment to further understand the API in the presence of many concurrent calls. The work in this paper provides a replacement for concurrent APIs using a formal model by (i) creating the 4M language to intuitively model natural language API descriptions; (ii) defining a novel role-based architecture to directly connect the C runtime to 4M to use the model as an instance of the API that is explorable, testable, and capable of exhaustive search; (iii) building an implementation of 4M as a rewriting system in Racket; and (iv) validating the process in a portion of the MCAPI communication API. The result is that when an API is formally modeled in 4M, it is possible to use the same model with native programs written against the API to explore system-wide program behavior.

Future work includes (i) adapting reductions that leverage SMT technology from model checking to mitigate state explosion in data and scheduling non-determinism [29]; (ii) partial order reduction based on persistent sets; (iii) improving the communication between the different runtimes using search order and undo stacks; (iv) case study in other APIs and in particular the MCA API for resource allocation as it deals with shared memory; and (v) an implementation of the 4M core in Maude to improve running time performance.

References

1. The 4M modeling language. https://github.com/ericmercer/4M
2. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press (1996)
3. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17, 643–644 (November 1974)
4. Dill, D.L., Drexler, A.J., Hu, A.J., Yang, C.H.: Protocol verification as a hardware design aid. In: IEEE International Conference on Computer Design: VLSI in Computers and Processors. pp. 522–525 (1992)
5. Elwakil, M., Yang, Z.: CRI: Symbolic debugger for MCAPI applications. In: Automated Technology for Verification and Analysis (2010)
6. Elwakil, M., Yang, Z.: Debugging support tool for MCAPI applications. In: Parallel and Distributed Systems (2010)
7. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. The MIT Press (2009)
8. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010), http://racket-lang.org/tr1/
9. Georgelin, P., Pierre, L., Nguyen, T.: A formal specification of the MPI primitives and communication mechanisms. Tech. rep., LIM (1999)
10. Godefroid, P.: Model checking for programming languages using Verisoft. In: Principles of Programming Languages. pp. 174–186 (1997)
11. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23, 279–295 (1997)

12. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (April 2006)
13. Jongmans, S.S., Hindriks, K., van Riemsdijk, M.: Model checking agent programs by using the program interpreter. In: Computational Logic in Multi-Agent Systems, vol. 6245, pp. 219–237 (2010)
14. Lamport, L.: TLA - the temporal logic of actions. `http://research.microsoft.com/users/lamport/tla/tla.html`
15. Li, G., DeLisi, M., Gopalakrishnan, G., Kirby, R.M.: Formal specification of the MPI-2.0 standard in TLA+. In: Principles and Practices of Parallel Programming. pp. 283–284 (2008)
16. Li, G., Gopalakrishnan, G., Kirby, R.M., Quinlan, D.: A symbolic verifier for CUDA programs. In: Principles and Practice of Parallel Programming. pp. 357–358 (2010)
17. McMillan, K.L.: Symbolic Model Checking: An approach to the state explosion problem. Ph.D. thesis, Carnegie Mellon University (1992)
18. Mercer, E.G., Jones, M.: Model checking machine code with the GNU debugger. In: International SPIN Workshop. pp. 251–265 (2005)
19. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Programming Language Design and Implementation (2007)
20. Palmer, R., Delisi, M., Gopalakrishnan, G., Kirby, R.M.: An approach to formalization and analysis of message passing libraries. In: Workshop on Formal Methods for Industrial Critical Systems (2007)
21. Păsăreanu, C.S., Mehlitz, P.C., Bushnell, D.H., Gundy-Burlet, K., Lowry, M., Person, S., Pape, M.: Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: International Symposium on Software Testing and Analysis. pp. 15–26 (2008)
22. Roscoe, A.W.: Model-checking CSP, pp. 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK (1994)
23. Sharma, S., Gopalakrishnan, G., Mercer, E., Holt, J.: MCC: A runtime verification tool for MCAPI user applications. In: Formal Methods in Computer-Aided Design. pp. 41–44 (2009)
24. Siegel, S.F., Avrunin, G.: Analysis of mpi programs. Tech. Rep. UM-CS-2003-036, Department of Computer Science, University of Massachusetts Amherst (2003)
25. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall International Series In Computer Science p. 155 (1989)
26. The Multicore Association: `http://www.multicore-association.org`
27. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Computer Aided Verification. pp. 66–79 (2008)
28. Šimša, J., Bryant, R., Gibson, G.: dbug: systematic testing of unmodified distributed and multi-threaded systems. In: Proceedings of the 18th international SPIN conference on Model checking software. pp. 188–193. Springer-Verlag, Berlin, Heidelberg (2011)
29. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: The Foundations of Software Engineering. pp. 23–32 (2009)
30. Wang, C., Yang, Y., Gupta, A., Gopalakrishnan, G.: Dynamic model checking with property driven pruning to dectect data race conditions. In: International Symposium on Automated Technology for Verification and Analysis (2008)
31. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MODIST: transparent model checking of unmodified distributed systems. In: Networked systems design and implementation. pp. 213–228 (2009)