

Formal Verification of MCAPI Applications Using the Dynamic Verification tool MCC

Subodh Sharma
School of Computing
University of Utah
Salt Lake City, UT 84112
Email: svcs@cs.utah.edu

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112
Email: ganesh@cs.utah.edu

Abstract—Multicore Communication API (MCAPI) is an API specification that offers low latency and high bandwidth communication service between the cores of a distributed embedded system. In this paper, we examine the application of formal methods to aid MCAPI application writers. We present MCC (MCAPI Checker), a tool under construction that performs direct code verification for the absence of deadlocks and safety assertion violations under a controlled environment by looking at *relevant* interleavings only. Redundant interleavings are removed from exploration space by employing MCAPI specific dynamic partial order reduction algorithm. We also propose an extension to MCAPI library with a probing functionality and present a successful implementation and usage of it in MCC’s verification process. As far as we know, this is the first push button model checker for MCAPI application writers right in the early stages of MCAPI development.

I. INTRODUCTION

The coming era of multicore computing will employ multiple and heterogeneous cores that will run large amount of thread-based shared memory and message passing based software. This will call for a standardization of an API for inter-core communication and synchronization. Multicore communication API (MCAPI [1]) is precisely such an API that is under active development by a group of more than 25 leading companies in the embedded system’s market. MCAPI provides collection of data transfer and synchronization functions that can be invoked by user applications running on the cores. MCAPI supports connectionless messages, connection-oriented packets and even scalar (bus-based) transfers. The example code shown in Section II-A illustrates how an MCAPI application might be written using POSIX threads. It should be noted at this juncture that to accurately model such APIs that have threaded and message passing based function semantics in to verification models and state transition relations becomes practically impossible. Thus, adopting symbolic model checking techniques or traditional model checking methods by constructing models of applications will not suffice for a domain like MCAPI. Runtime verification methods pioneered in Verisoft [4] is precisely meant for domains under discussion. The work presented in this paper describes the very *first* dynamic (or runtime) verification tool for MCAPI applications called MCC (MCAPI Checker). Our main contribution is the detailed description of MCC which now works for the

connectionless message passing constructs of MCAPI. A large number of new concurrency APIs are being introduced to program future multi-core systems. We predict that each such API will require a DPOR-based algorithm for verification. In our past work, we have built two such DPOR customizations for other APIs, namely Inspect [9] (for Pthreads [3]) and ISP [10] (for MPI [2]). This work builds on the strengths of these two past projects, but continues to deviate in novel ways from these tools. The main differences are in the manner in which we handle the MCAPI semantics, and the fact that MCC handles both threading and message passing. Other tools (*e.g.*, CHESS [11]) follow approaches to contain the number of interleavings by bounding the number of preemptions. In addition to being prone to bug omissions, preemption bounding is not a suitable approach for formal verification when message passing concurrency is involved. This is because in message passing systems, many actions are largely independent of other actions (and hence commute) – and for these steps, exploring different interleavings is wasteful. Section II begins with an informal overview of MCAPI. It also identifies the bug classes that an application writer needs to worry about in writing an MCAPI application. Section II-C describes the design of MCC followed by initial results and concluding remarks in Section III.

II. OVERVIEW OF MCAPI

An MCAPI node serves as a logical abstraction for a thread of activity (which can be realized in multiple ways). It has multiple endpoints, each being a ⟨node id, port id⟩ pair. MCAPI also provides packet channels and scalar channels (not supported in MCC yet). Communication occurs within MCAPI through connectionless messages, connected packet channels, or connected scalar channels. All communications occur with respect to endpoints. Typical API calls include MCAPI_INITIALIZE, MCAPI_FINALIZE, as well as calls to create endpoints, send/receive messages in non-blocking mode, and later await for the completion of the send/receive. Details are available from [1].

A. MCAPI example code

Figure 2 illustrates a completely executable MCAPI code. The main thread in the example code spawns three threads.

Threads with odd IDs send a message to the thread with the even ID. The senders and the receiver have to explicitly create sending and receiving endpoints by issuing MCAPi create endpoint calls. In order to get the address of the remote receiving endpoint, `mcapi_get_endpoint` call is issued. Note that `mcapi_get_endpoint` call is a blocking call. If the remote endpoint whose address is being requested does not exist then the system deadlocks. A website [12] presents a list of bugs that can potentially affect MCAPi applications.

B. Early Efforts in MCAPi Verification

We started our work by first trying to formally model the MCAPi specification in SAL ([5]) and look for any holes in the specification. This is very similar to what we have already done in [6], [7] in which we formally specified 50 of the nearly 320 MPI functions. The mere act of formalization revealed unto us six omissions in the MPI documents we had access to. It is unclear how actual implementations of the MPI library treat these six cases. Suffices to say that it is precisely these differences in implementations for the unstated cases that should worry an engineer in terms of portability of codes and the likelihood of security holes. In our project, we preferred to postpone this direction of research for now, and instead decided to focus on the dynamic verification of MCAPi user applications. The reason for this decision stems from the fact that a dynamic verifier can immediately start helping MCAPi application writers by finding bugs in their specifications. With a dynamic verifier in mind and given that MCAPi has threaded implementation, our initial attempt was to directly employ Inspect to check MCAPi applications. However, our first experiment itself was discouraging. Even for single threaded user application, model checking problem blew up consuming 5 hours and exploring 32,000 interleavings before we terminated the verification process. The reason for this blow-up was traced to the lack of awareness (on part of Inspect) about MCAPi applications notion of action independence. In particular, when MCAPi calls are flattened and treated as Pthread calls, the high level commutability properties enjoyed by MCAPi calls are lost! This prompted us to build MCC along the lines of ISP, but with thread awareness built in (unlike ISP, which currently cannot handle threading).

C. Overview of MCC

We are currently employing a Pthread based reference implementation produced by MCA. In this paper, we focus almost exclusively on MCAPi’s connectionless send and receive commands, and verify for local assertions placed within threads, as well as deadlocks. The overall nature of execution control, along with DPOR is adapted from our group’s tool ISP [13] and is illustrated in Figure 1.

MCC has three components: Instrumenter, Profiler, Scheduler. The compile time instrumenter runs through the program body and converts all MCAPi calls and Pthread create and join calls to our own wrapper calls. The profiler intercepts these wrapper calls made by the user application, performs the required book keeping, and subsequently communicates

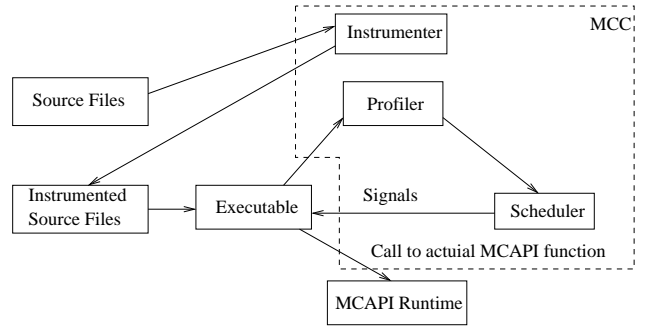


Fig. 1. MCC workflow

with the verification scheduler. The verification scheduler can either give a *go ahead* to a calling MCAPi thread or refrain from doing so to arrest the progress of that thread. MCAPi receive calls are implicitly non-deterministic. Our strategy to accommodate receive nondeterminism is: (i) have a runtime algorithm to determine all senders that can match each receive, (ii) then replay the execution of the entire MCAPi application, where for each replay we ensure that one of these sends matches the receive. The scheduler delays the processing of the receive calls till all sends that can potentially match the receive are dynamically discovered. However, the matching decision taken by the scheduler may not be consistent with the actual matching that happens at runtime. Figure 3 illustrates how such an inconsistency can arise

Suppose the scheduler discovers that the send calls from threads T0 and T1 can both potentially match the receive posted by thread T2. Clearly, we must replay the execution for both these matches: in one execution, T0’s call will match this receive and T1’s call will match another receive; and in the other execution, T1’s call will match this receive. However, unlike in MPI (where one could dynamically rewrite a wildcard receive with a *specific source receive* – thus forcing certain receive/send matches to occur), MCAPi does not have any mechanisms to force a specific send and receive to match! This can cause the following race condition to manifest within the MCAPi runtime: (i) the scheduler gives a go-ahead to T0’s send and T2’s receive, (ii) then the scheduler gives a go-ahead to T1’s send and some other matching receive. But after step (i), it is not ensured that the send/receive would have occurred in the MCAPi runtime! It is possible that T1’s send races ahead and enqueues itself into T2’s input-side end point. The only speed-independent (robust) solution is to implement a blocking probe feature that can check for specific send/receive matches to have occurred. We engineered such an extension into MCA’s MCAPi library. *This is an example of how early formal verification research can contribute to an API before it is too late.* The new probe functionality takes the receiving endpoint as an input argument and probes the dedicated FIFO ordered receive queue until the message from the sender’s endpoint is received. The scheduler, in the meantime, is blocked and starts issuing the rest of the *go-aheads* only after the successful return of the probing function.

```

1:#include <mcapi.h>
2:#include <stdio.h>
3:#include <pthread.h>
4:#define NUM_THREADS 3
5:#define BUFF_SIZE 64
6:#define PORT_NUM 1

7:mcapi_endpoint_t endpoints[NUM_THREADS];

8:typedef struct {
9:  int thread_id;
10:  mcapi_endpoint_t endpoint;
11:  int rcv_node;
12:} thread_data;

13:thread_data thread_data_array[NUM_THREADS];

14:void* run_thread (void *t) {

15:  mcapi_status_t status;
16:  mcapi_version_t version;
17:  char msg[BUFF_SIZE];
18:  mcapi_endpoint_t send_endpt,
19:    rcv_endpt;
20:  int tid,rcv_size;

21:  thread_data *my_data = (thread_data *)t;
22:  tid = my_data->thread_id;

23:  mcapi_initialize(tid,&version,&status);
24:  if (tid == 1) {
25:    rcv_endpt =
26:    mcapi_create_endpoint (PORT_NUM,&status);
27:    mcapi_msg_rcv(rcv_endpt,msg,
28:      BUFF_SIZE,&rcv_size,
29:      &status);
30:    mcapi_msg_rcv(rcv_endpt,msg,
31:      BUFF_SIZE, &rcv_size,
32:      &status);
33:  } else {
34:    /* even threads send */
35:    send_endpt = mcapi_create_endpoint
36:      (PORT_NUM,&status);
37:    rcv_endpt = mcapi_get_endpoint
38:      (1,PORT_NUM,&status);
39:    sprintf (msg,"Guten tag MCAPI from ep:
40:      %i to ep:%i tid:%i",
41:      (int)send_endpt,
42:      (int)rcv_endpt,tid);
43:    mcapi_msg_send(send_endpt,rcv_endpt,
44:      msg,strlen(msg),
45:      1,&status);
46:  }
47:  mcapi_finalize(&status);
48:  return NULL;
49:}

36:int main () {
37:  pthread_t threads[NUM_THREADS];
38:  int rc, t;

39:  for(t=0; t<NUM_THREADS; t++){
40:    thread_data_array[t].thread_id = t;
41:  }
42:  /* run all the threads */
43:  for(t=0; t<NUM_THREADS; t++){
44:    rc = pthread_create(&threads[t],
45:      NULL, run_thread,
46:      (void *)&thread_data_array[t]);
47:  }
48:  for (t = 0; t < NUM_THREADS; t++) {
49:    pthread_join(threads[t],NULL);
50:  }
51:  return rc;
52:}

```

Fig. 2. MCAPI example C program

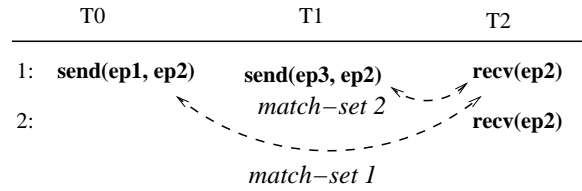


Fig. 3. MCAPI Receive Problem

D. Overview of the MCC algorithm

MCC assumes that all threads are created at the very outset in a single statement block by one *main* thread. Thus, the scheduler knows the total count of threads in the system before even the verification process starts. The instrumentation component of MCC instruments the thread function bodies with *thread-start* and *thread-end* call. The thread count is incremented whenever a *thread-start* call is trapped by the scheduler. Note that the status of such threads is updated to blocked state. The scheduler waits until the *main thread* issues an MCAPI communication call or a *thread-join* call. At that moment, the scheduler has successfully ascertained the total thread count and all blocked threads are given a *go-ahead*.

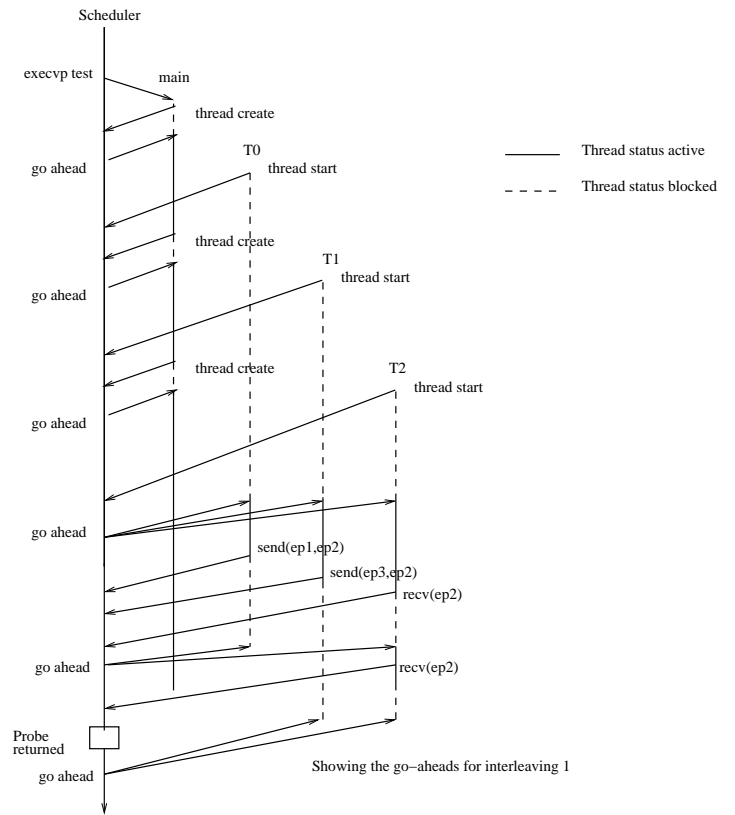


Fig. 4. Working of MCC on an example.

The verification stage of MCC starts at this point forward.

- The scheduler starts receiving MCAPI calls (transitions) from each runnable thread.
- After receiving each such transition, the current state of the scheduler is updated. The state of the scheduler

maintains an explicit transition list of enabled transition from each runnable thread. Each state also maintains the ample list [14] of transitions that are yet to be explored.

- If the received transition is of blocking type, the status of the thread is updated to be in blocked state. If the transition is of *thread-end* type then the thread is given a *go-ahead* and thread count is decreased.
- Scheduler continues to receive transitions from *runnable* threads until all threads are in blocked state.
- Once no thread is in runnable state, the scheduler then identifies *match-sets* which consists of matching transitions that complete each other (e.g., sends to a specific endpoint and receives from the same endpoint). Note that these match sets are formed out of the ample list of transitions at a state. The transitions in the ample set are paired as $\langle \text{send}, \text{receive} \rangle$ based on compatible arguments. Ample sets are formed only once during the first run. In the subsequent runs, per-state ample sets are updated by striking out entries that have already been explored. For instance, in Figure 3, the ample set computed after the first run is the following: (i) at state 1: $\{ \langle \text{send}(\text{ep1}, \text{ep2}), \text{rcv}(\text{ep2}) \rangle, \langle \text{send}(\text{ep3}, \text{ep2}), \text{rcv}(\text{ep2}) \rangle \}$; (ii) Scheduler signals a go-ahead to $\langle \text{send}(\text{ep1}, \text{ep2}), \text{rcv}(\text{ep2}) \rangle$. Thus ample set in state 2 is $\{ \langle \text{send}(\text{ep3}, \text{ep2}), \text{rcv}(\text{ep2}) \rangle \}$; (iii) In the next run, ample set at state 1 is again visited and scheduler now decides to signal a go-ahead to $\langle \text{send}(\text{ep3}, \text{ep2}), \text{rcv}(\text{ep2}) \rangle$. Thus in state 3 the ample set reduces to $\{ \langle \text{send}(\text{ep1}, \text{ep2}), \text{rcv}(\text{ep2}) \rangle \}$.
- The scheduler then liberates the threads forming match sets. A deadlock is flagged if in case, no match-sets are found and there are still non-finalized threads in the system.

The whole aforementioned process is recursed until there are no more interleavings to explore. In short, when the *ample-list* at the start state become empty. It is important to note here that we re-execute the target for each interleaving exploration. The basic semantics guaranteed is that of non-overtaking (point to point FIFO ordering) as explained in [13].

Figure 4 shows the time-line diagram of running the example code from Figure 2 under the control of MCC scheduler. Note that the scheduler thread blocks after signaling the match-sets to move to runtime *and returns only after the probing function has returned successfully*.

III. RESULTS

We have developed the first dynamic verifier that handles a subset of MCAPAPI calls using only publicly available MCAPAPI resources. We develop a scheduler with a robust runtime control method building on past work. MCC has successfully handled several simple example programs. Deterministic programs are verified in one interleaving. Through active collaborations with MCA, we are developing public-domain MCAPAPI benchmark applications. We are also extending MCC to cover the full gamut of MCAPAPI calls, with an approach (under testing) for handling connection oriented packet channels. Unstructured shared memory and message passing programs

can be too tricky: for this domain, we are expecting safe concurrency patterns to emerge, which MCC will then exploit.

IV. CONCLUDING REMARKS AND FUTURE WORK

We realize that many crucial differences exist between MCAPAPI and other communication APIs. As an important part of our future work, we will try to fully understand these differences and engineer algorithms to make the following goals reachable:

- The role of formal semantics of multicore communication APIs will be very important to build sound API implementation-independent analysis tools. The attempt to write a formal semantics can also help us come up with default properties that can be checked by a dynamic analyzer. We see this as an important future work to pursue.
- Previous dynamic verification techniques were driven by specific test harnesses. This decision was mainly due to the complexity of attempting verification for all test harnesses (environments). In case of MCAPAPI, it may be feasible (and even necessary) to try and achieve better coverage than for one specific test harness by employing symbolic analysis techniques.

Acknowledgement: This work is supported by NSF award CCF-0811429 and SRC grant 1847.001.

REFERENCES

- [1] <http://www.multicore-association.org>.
- [2] <http://www.mcs.anl.gov/mpi/>
- [3] <http://www.llnl.gov/computing/tutorials/pthreads/>
- [4] Patrice Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. CAV 1997, 476-479.
- [5] Natarajan Shankar. Symbolic analysis of transition systems. ASM 2000, LNCS 1912, pages 287-302.
- [6] R. Palmer, M. Delisi, G. Gopalakrishnan, R.M. Kirby. An approach to formalization and analysis of message passing libraries. FMICS 2007, pages 164-181, LNCS 4916.
- [7] Guodong Li, Michael DeLisi, Ganesh Gopalakrishnan, and Robert M. Kirby. Formal specification of the MPI-2.0 standard in tla+. In Principles and Practices of Parallel Programming (PPoPP), pages 283-284, 2008.
- [8] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. POPL, 110-121, 2005.
- [9] Yu Yang, Xiofang Chen, Ganesh Gopalakrishnan, R.M. Kirby. Distributed Dynamic Partial Order Reduction Based Verification. SPIN 2007, 58-75.
- [10] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. PPoPP 2008. 285-286.
- [11] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI 2007, 446-455.
- [12] http://www.cs.utah.edu/formal_verification/mediawiki/index.php/MCAPAPI
- [13] Sarvani Vakkalanka, Ganesh Gopalakrishnan, and Robert M. Kirby. Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. CAV 2008, 66-79.
- [14] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.