2009-11-01

MCC: A Runtime Verification Tool for MCAPI User Applications

Eric G. Mercer
eric_mercer@byu.edu

Ganesh Gopalakrishnan

Jim Holt

Subodh Sharma

Original Publication Citation

S. Sharma, G. Gopalakrishnan, E. Mercer, and J. Holt, "MCC - A runtime verification tool for MCAPI user applications", in Proceedings of Formal Methods in Computer Aided Design (FMCAD), Austin, USA, November 29.

BYU ScholarsArchive Citation

MCC: A runtime verification tool for MCAPI user applications

Subodh Sharma
School of Computing
University of Utah
Salt Lake City, UT 84112
svs@cs.utah.edu
www.cs.utah.edu/˜svs

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112
ganesh@cs.utah.edu
www.cs.utah.edu/˜ganesh

Eric Mercer
Computer Science Department
Brigham Young University
Provo, UT 84602.
eric.mercer@byu.edu
http://faculty.cs.byu.edu/˜egm

Jim Holt
Networking and Multimedia Group
Freescale Semiconductor, Inc.
7700 West Parmer Lane, MD:PL51
Austin, TX 78749
Jim.Holt@freescale.com

***Abstract*—We present a dynamic verification tool MCC for Multicore Communication API applications – a new API for communication among cores. MCC systematically explores all relevant interleavings of an MCAPI application using a tailor-made dynamic partial order reduction algorithm (DPOR). Our contributions are (i) a way to model the non-overtaking message matching relation underlying MCAPI calls with a high level algorithm to effect DPOR for MCAPI that controls the lower level details so that the intended executions happen at runtime; and (ii) a list of default safety properties that can be utilized in the process of verification. To our knowledge, this is the first push button model checker for MCAPI application writers that, at present, deals with an interesting subset of MCAPI calls. Our result is the demonstration that we can indeed develop a dynamic model checker for MCAPI that can directly control the non-deterministic behavior at runtime that is inherent in any implementation of the library without additional API modifications or additions..**

I. Introduction

Future embedded systems will employ multiple and heterogeneous cores (CPU, DSP, *etc.*) and run a large amount of thread-based shared-memory and message-passing based software. To permit software reuse and derive the benefits of standardization, an API for multicore communication (MCAPI) is being developed by a group of over 25 leading companies [1]. Unlike large existing APIs such as MPI [2] that are meant for the high-end compute clusters, MCAPI is being designed ground-up from a clean slate to address the needs of embedded multicore systems. MCAPI supports connection-less messages, connection-oriented packets, and even scalar (bus based) transfers. The example in Section II-B shows how an MCAPI application might be written using POSIX threads (Pthreads, [3]) for orchestrating the overall computation.

This paper describes the first *dynamic* (or *runtime*) formal verification tool for MCAPI applications called MCC (MCAPI Checker) where dynamic means that the verification process takes place at run time using the MCAPI runtime environment. It is practically impossible to construct verification models or state transition relations that accurately model the C/Pthread semantics and the dynamic execution semantics of MCAPI functions (over 50 API calls). Thus, neither symbolic model checking methods nor model-based verification methods (*e.g.*, modeling C/Pthreads/MCAPI in say Promela) can help in verifying MCAPI applications. Dynamic verification methods were pioneered in Verisoft [4] precisely for this domain. In order to prevent the exponential growth in the number of potential thread interleavings (schedules), we will employ dynamic partial order reduction methods [5] that have been shown to be very effective in software verification.

Contributions: Our main contribution is the MCC model checker that verifies the connection-less message passing constructs of MCAPI using a reference implementation of the API. 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 [5] algorithm for verification. In our past work, we have built two such DPOR customizations for other APIs, namely Inspect [6] (for Pthreads) and ISP [7] (for MPI). This work builds on the strengths of ISP and Inspect but deviates from these tools in novel ways. For instance, in case of MPI, an explicit wildcard receive is provided whereas MCAPI, which borrows many ideas from MPI, does not do so. Therefore ISP's solution to accommodate the non-determinism by rewriting wildcard receive calls dynamically into specific receive calls so as to enforce a deterministic match with a sender at runtime, will not work in MCC's verification methodology. Unlike ISP, the scheduler of MCC also manages thread creation and thread join calls. MCC's verification methodology differs from Inspect with regard to the DPOR method that is employed. Inspect's DPOR mechanism does not support message passing. Other tools (*e.g.*, CHESS [8]) 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 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.

Funded collaoratively by NSF CCF 0903408/0903491 and SRC contract 2009-TJ-1993/1994

II. Verification 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 connection-less 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. Overview of MCC

We are building MCC even before public domain MCAPI applications are available. We also believe that MCC must be able to accept MCAPI library implementations produced by industries "as is," and use them to provide the execution semantics for MCAPI calls. We are currently employing a Pthread based reference implementation produced by the Multicore Associaton (MCA). All this ensures that (i) we will not waste time recreating the functionality of MCAPI (a very arduous task), and (ii) we can switch out one MCAPI library and switch in, say, a piece of silicon that purportedly realizes MCAPI (to see if we can find any new bugs by doing so during the *platform testing* mode).

In this paper, we focus exclusively on MCAPI's connection-less send and receive commands, and verify local assertions placed within threads, as well as deadlocks. We have also identified a list of default safety checks that are listed in [9] that we hope to incorporate in our future realizations of MCC.

MCAPI receive calls are non-deterministic in the presence of concurrent sends to a common endpoint. MCAPI receive calls only specify the destination endpoints on which the message should be received which precisely is the cause for non-determinism. Since receives are applied to endpoints and so are sends, it is possible that two sends could have a race in matching with a receive call. Our strategy to accommodate receive nondeterminism is: (i) have a dynamic algorithm to determine all senders that can match each receive, and (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 overall nature of execution control, along with DPOR is adapted from our group's tool ISP [10] and is illustrated in Figure 1.

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 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. The scheduler achieves two end goals. First, it manifests *independent* [11] thread steps according to a canonical order. *This ordering effects partial order reduction.* Second, for (non-deterministic) receives, the scheduler delays the processing of the receive till all sends that can potentially match the receive are dynamically discovered. It then replays the execution for these receives (these being the interesting *ample sets* [11]). The pseudocode of the scheduler is given in Figure 4.

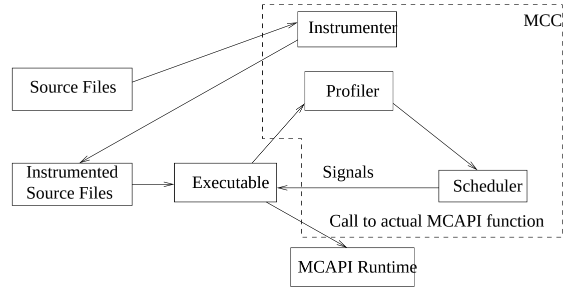

Fig. 1. MCC workflow

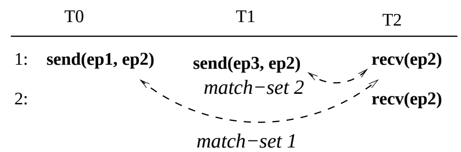

Fig. 2. MCAPI non-deterministic receive example

Figure 2 illustrates the motive behind our scheduler end-goal of delaying the processing of receive calls till all enabled matching sends are discovered. Suppose the scheduler discovers (as shown in Figure 2) 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 T2's first receive and T1's call will match T2's second receive (else there is a deadlock); and in the other execution, T1's call will match T2's first receive.

B. Illustration of MCC on an Example

Figure 3 illustrates a snippet of an executable MCAPI code prior to the instrumentation done by the MCC. The main thread in the example code spawns three threads. Threads with IDs 0 and 1 send a message to the thread with ID 2. The senders and the receiver have to explicitly create sending and receiving endpoints by issuing MCAPI create endpoint calls (lines 6,10). In order to get the address of the remote receiving endpoint, a *mcapi_get_endpoint* call is issued (line 11). Note that the *mcapi_get_endpoint* call is a blocking call. If the requested endpoint is never created then the *mcapi_get_endpoint* call may cause the system to deadlock. The MCC scheduler stores a list of endpoints that have already been created. An *mcapi_get_endpoint* call is instantly issued to the runtime if the associated endpoint has already been created, otherwise the scheduler delays the issuing of the call until the requested endpoint is created. The instrumentation component of MCC instruments the MCAPI communication calls with the same call names, however, the call names are now prefixed with "p". Additionally the POSIX thread create and join calls are also replaced by our own wrapper function calls. The

```
1:#define NUM_THREADS 3
2:#define PORT_NUM 1

3:void* run_thread (void *t) {
     ...
4:  mcapi_initialize(tid,&version,&status);
5:  if (tid == 2) {
6:  recv_endpt =
     mcapi_create_endpoint (PORT_NUM,&status);
7:  mcapi_msg_recv(recv_endpt,msg,
                   BUFF_SIZE,&recv_size,
                   &status);
8:  mcapi_msg_recv(recv_endpt,msg
                   BUFF_SIZE, &recv_size,
                   &status);
9:   } else {
10:  send_endpt = mcapi_create_endpoint
                   (PORT_NUM,&status);
11:  recv_endpt = mcapi_get_endpoint
                   (2,PORT_NUM,&status);
12:  mcapi_msg_send(send_endpt,recv_endpt,
                   msg,strlen(msg),
                   1,&status);
13:  }
14:  mcapi_finalize(&status);
15:}

16:int main () {
     ...
17:  for(t=0; t<NUM_THREADS; t++){
18:    rc = pthread_create(&threads[t],
       NULL, run_thread,
       (void *)&thread_data_array[t]);
19:  }
20:  for (t = 0; t < NUM_THREADS; t++) {
21:    pthread_join(threads[t],NULL);
22:  }
    ...
24:}
```

Fig. 3. MCAPI example C program

thread function bodies are instrumented with *thread_start* and *thread_end* calls which act as barrier points. The notion of introducing aforesaid calls is explained in Section II-C.

The wrapper calls are defined in MCC's profiler library. Subsequently, the executable is run under the controlled environment of the scheduler.

C. MCC Algorithm

Figure 4 in Section II-C explains the working of the scheduler. It assumes that all threads are created at the outset of the program, and thus is able to determine the total number of threads alive in the system (lines 2-15). Since the scheduling decisions are made once *all* the threads in the system have hit their local fence operations, it therefore becomes imperative to discover the total count of runnable threads in the system. The scheduler waits till all threads in the system have posted their respective blocking calls and have come to a halt (lines 18-28). Note that if a thread issues the *mcapi_finalize* or *thread_end* type calls then the count of alive threads is decremented (lines 25-27). At line 16, either the user spawned threads are blocked at their *thread_start* calls or they have yet to issue any MCAPI calls. Note that *thread_start* calls in the instrumented code act as barrier points that make sure that all threads are ready to run at the same state. The scheduler signals all the blocked threads to continue with their execution and continues to receive transitions from runnable threads until no thread is in a running state (lines 19-28). The scheduler then identifies *match-sets* (line 30) which consist of matching transitions that complete each other (*e.g.*, sends to a specific endpoint and receives from the same endpoint). The scheduler then liberates the threads forming the match-set (line 31). To identify the match-sets we identify the ample set [11] of transitions. The transitions in the ample set are then grouped as ⟨send, receive⟩ pairs based on compatible arguments. A deadlock is flagged if no match-sets are found and there are still runnable threads in the system (i.e., the *count* variable is still not 0).

```
 1: GenerateInterleaving( ) {
 2:    while (1) { // Computes the total number of threads alive
 3:       t_i = receive_transition ();
 4:       if (t_i is thread_create) {
 5:          num_threads++;
 6:          signal go-ahead to thread_of(t_i);
 7:       }
 8:       if (t_i is thread_join || t_i is MCAPI communication call
             by thread "main") {
 9:          signal go-ahead to thread i;
10:          break;
11:       }
12:       if (t_i is thread_start) {
13:          update the status of thread i to blocked;
14:       }
15:    }// while (1) ends here

16:    count = num_threads;
17:    signal go-ahead to all the blocked threads;

18:    while (count) { // till no more threads are alive
19:       for each (runnable thread i)  {
20:          t_i = receive_transition from thread i;
21:          update transition_list of thread_of (t_i) in the current
                state;
22:          if (t_i is of blocking_type) {
23:             update the status of thread i to blocked;
24:          }
25:          if (t_i is of type thread_end) {
26:             count --;
27:          }
28:       }
          // All threads are blocked here
29:       while (no thread is runnable) {
30:          find_matchset ();
31:          unblock the threads owning transitions in the above
                match-set;
32:       }
33:    }// while (count) ends here
34: }

35: find_matchset( ) {
36:    if (ample_list of transitions is not empty) {
37:       for each (t_i in head element of the ample_list) {
38:          give a go-ahead to thread_of (i);
39:       }
40:       return;
41:    }
42:    flag that a deadlock found;
43: }
```

Fig. 4. MCC scheduler algorithm

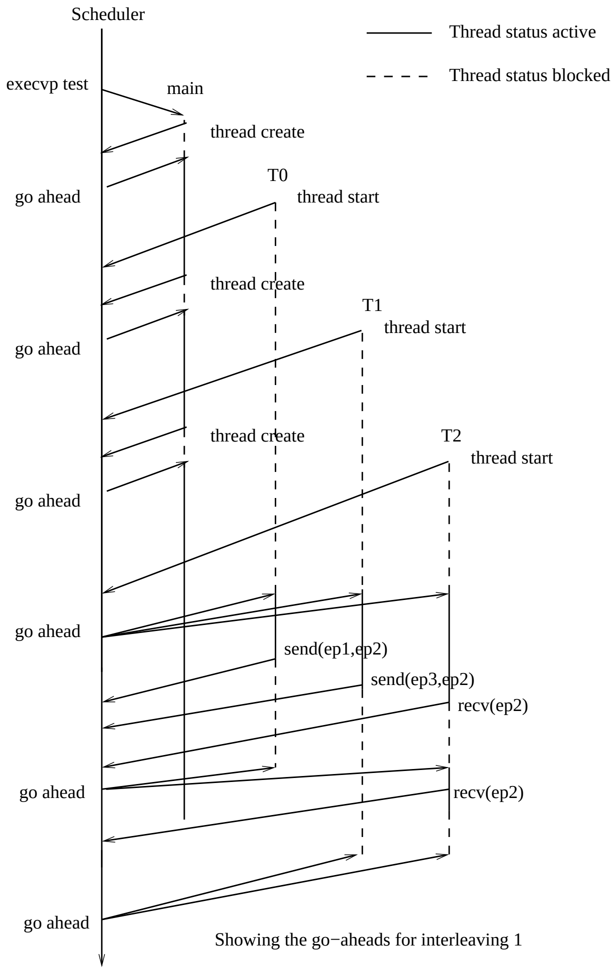

Fig. 5. Working of MCC on an example.

The procedure *GenerateInterleaving* is called in a loop until no more interleavings (replays) are left to explore. Different interleavings are verified by restarting the test target. The scheduler maintains a state consisting of a list of enabled transitions from each process. The ample set for each such state is computed in the first run. In subsequent runs, the per-state ample set is only updated by removing the match-sets that are signaled to proceed in that particular state of the run. In Figure 2, the match-set computed after the first run is the following:

- The match-set at state 1: $\{\langle send(ep1, ep2),\ recv(ep2)_1\rangle$, $\langle send(ep3, ep2),\ \ recv(ep2)_1\rangle\}$. Note that $recv(ep2)_1$ denotes the first receive call by T2.
- Assume that the scheduler signals a go-ahead to the entry $\langle send(ep1, ep2),\ recv(ep2)_1\rangle$. Thus, the match-set in state 1 is updated to $\{\langle send(ep3, ep2),\ recv(ep2)_1\rangle\}$.
- At state 2 the match-set formed is a singleton set $\{\langle send(ep3, ep2),\ recv(ep2)_2\rangle\}$. Note that $recv(ep2)_2$ denotes the second receive call by T2. After the scheduler signals a go-ahead, the match-set is reduced to an empty set.
- In the next run, the ample set at state 1 is again visited and the scheduler now decides to signal a go-ahead to $\langle send(ep3, ep2),\ recv(ep2)_1\rangle$.

Thus, two interleavings are sufficient for exhaustive exploration. The basic semantics guaranteed is that of non-overtaking (point to point FIFO ordering) as explained in [10]. Figure 5 shows the time-line diagram of an execution interleaving explored by running the instrumented example code from Figure 3. The scheduler starts running by executing the test target. The main thread of the test target issues the thread create calls and subsequently gets blocked until it receives a go-ahead signal from the scheduler. Note that all threads created by the main thread are blocked on their respective *thread_start* calls. The scheduler after assessing the total thread count, signals a go-ahead to all such threads blocked on the *thread_start* calls. The scheduler computes a match-set once all the threads have blocked on their respective MCAPI operations. It then selects one entry from the match-set and signals a go-ahead to the participating threads (in Figure 3, it is $\langle T0,\ T2\rangle$ followed by $\langle T1,\ T2\rangle$).

III. Results and Conclusions

We have developed the first dynamic verifier that handles a subset of MCAPI calls using only publicly available MCAPI resources. We have developed a scheduler with a robust run-time control method building on past work. MCC has successfully handled several simple example programs. Deterministic programs are verified in one interleaving for the absence of deadlocks and safety violation assertions. The example program from Figure 3 was verified in 2 interleavings with no deadlocks found. Through active collaborations with the MCA, we are developing public-domain MCAPI benchmark applications. We are also extending MCC to cover the full gamut of MCAPI calls, with an approach (under testing) for handling connection-less oriented non-blocking calls. Future research also includes programs that use shared memory and message passing for this mixed domain, we are expecting safe concurrency patterns to emerge, which MCC will then exploit. We acknowledge Jay Bhadra of Freescale and Neha Rungta and Topher Fischer of BYU for their help on this work.

References

[1] Multicore Association. http://www.multicore-association.org.
[2] MPI Forum. http://www.mpi-forum.org
[3] http://www.opengroup.org/onlinepubs/007908799/xsh/threads.html
[4] Patrice Godefroid. Software Model Checking: TheVeriSoft Approach. FMSD 2005, vol 26-2, 77-101.
[5] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *POPL*, 110–121, 2005.
[6] Yu Yang, Xiofang Chen, Ganesh Gopalakrishnan, R.M. Kirby. Distributed Dynamic Pratial Order Reduction Based Verification. SPIN 2007, 58-75.
[7] Sarvani Vakkalanka, Subodh V. Sharma, Ganesh Gopalakrishnan, and Robert M. Kirby. ISP: A tool for model checking MPI programs. PPoPP 2008. 285-286.
[8] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. PLDI 2007, 446-455.
[9] http://www.cs.utah.edu/formal_verification/mediawiki/index.php/MCAPI
[10] 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.
[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.