2011 12th International Workshop on Microprocessor Test and Verification

Verification Tests for MCAPI

Alper Sen
Department of Computer Engineering
Bogazici University
Istanbul, Turkey
Email: alper.sen@boun.edu.tr

Etem Deniz
Department of Computer Engineering
Bogazici University
Istanbul, Turkey
Email: etem.deniz@boun.edu.tr

***Abstract*—Simulation based verification is the most commonly used verification technique in the industry. However, the effort of generating new tests is non-trivial especially for the newly emerging multicore applications. Automated test generation greatly reduces the cost of testing ultimately improving the quality of multicore software. We develop an automated test generation framework using static analysis techniques for embedded multicore applications. We use an emerging multicore standard for multicore applications, named Multicore Communication API (MCAPI). MCAPI is a lightweight API that targets heterogeneous multicore embedded systems. Specifically, our techniques leverage mutation testing and model checking. We present preliminary experimental results to validate the effectiveness of our approach.**

I. Introduction

The emergence of multicore heterogeneous embedded systems requires the development of multicore software standards [1]. Such standards have several benefits including code portability, reusability, and consistent behavior. Multicore Communication API (MCAPI) is a standard developed by Multicore Association [2] for heterogeneous embedded systems. MCAPI is a message passing programming library, similar to the well-known MPI [3] library for distributed systems. However, MCAPI is a lightweight API that targets heterogeneous multicore embedded systems and it has a smaller memory footprint than MPI since performance and power consumption are crucial for embedded systems such as consumer electronics, medical instruments, or networking equipment. A sample MCAPI library implementation is available from the Multicore Association [2].

Multicore software is notoriously hard to design and analyze due to the complexity brought by concurrency and nondeterminism. Verification becomes even a bigger challenge for multicore software. Among several verification techniques, testing (simulation) is the most commonly used in the industry. In this paper, we develop techniques to improve testing of multicore software.

Test generation is an important complement to the directed or random tests that are used for testing in practice. Automated test generation can reduce the cost of testing as it allows to improve coverage in an automated manner. Automated test generation (or model based testing) has been studied in the literature mainly through the use of static (model checkers) or dynamic analysis of software. See [4] for a detailed analysis of using model checkers for test generation. Model checking can be used to determine whether the original program and the program obtained by inserting a fault are equivalent or not. If they are not equivalent, then a counter example is generated, and this counter example serves as a test case that detects the inserted fault. Often times model checkers are limited in the size and the complexity of programs they can handle hence abstractions are used to make the program manageable. We employ such an abstraction of the MCAPI library in this paper. The ANSI-C Bounded Model Checker (CBMC) [5] has been used for test generation for ANSI-C programs, concurrent SystemC programs and recently for Simulink [6], [7]. Also, automated test pattern generation has widely been studied for hardware descriptions [8]. The generated patterns are used to test semiconductor devices after manufacturing.

Our test generation framework makes use of mutation testing [9], [10] for inserting faults into the programs. Mutation testing is a software testing technique that allows to measure the quality of a test suite by evaluating whether the test suite can detect the mutations (syntactic changes, faults) in the program. In this paper, we use a previously developed mutation library for MCAPI standard that targets message passing communication constructs. There is also several work in the literature on using mutation testing for Java, C, and SystemC [11], [12], [13].

In this paper, our main contribution is the development of an automated test generation tool for embedded multicore applications using the MCAPI standard. To the best of our knowledge, there is no previous work on automated test generation for embedded applications using message passing communication. We used mutation testing and bounded model checking together. Our preliminary experiments show that we can obtain high coverage but we are limited by the capacity of the model checker.

II. Background on MCAPI

MCAPI has three fundamental communication types: connectionless datagrams for messages; connection-oriented, unidirectional, FIFO packet streams for packet channels; and connection-oriented single word uni-directional, FIFO packet streams for scalar channels. Scalar channels are aimed at systems that have hardware support for sending small amounts of data (for example, a hardware FIFO). For lack of space, we describe connectionless messages in the rest of the paper,

978-0-7695-4594-3/11 $26.00 © 2011 IEEE
DOI 10.1109/MTV.2011.18

however our test generation tool implements all communication types.

MCAPI messages can be sent or received in either blocking or non-blocking fashion. Blocking send function ($mcapi\_msg\_send$) in our MCAPI library will block if there is insufficient memory space available at the system buffer. When sufficient memory space becomes available, the function will complete. The non-blocking send function ($mcapi\_msg\_send\_i$), returns immediately even if there is no memory space available. MCAPI stores messages in a queue at the receiver endpoint and the size of the queue can be configured according to the users demands. Blocking receive function ($mcapi\_msg\_recv$) returns once a message is available in endpoint's message queue, whereas a non-blocking receive function ($mcapi\_msg\_recv\_i$) returns immediately even if there is no message available. Message receive functions do not specify the sender endpoint and can match any of the senders depending on the execution schedule. Packet channels use connection-oriented communication. They use FIFO order and they can have blocking or non-blocking send and receive functions. Scalar channels are aimed at systems that have hardware support for sending small amounts of data (for example, a hardware FIFO) and scalar channels have only blocking functions due to high performance.

For non-blocking function requests (send or receive), the user program receives a handle for each request and can then use the non-blocking management functions to test if the request has completed with $mcapi\_test$ function, or wait for it either singularly with $mcapi\_wait$ or wait for any one of requests in an array of requests with $mcapi\_wait\_any$ function. The user program can also cancel non-blocking function calls using $mcapi\_cancel$ function.

We show an example multicore program that uses MCAPI library in Fig. 1. The program has two concurrent threads ($Thread1$ and $Thread2$) communicating through connectionless non-blocking message exchange. Each thread initializes the MCAPI environment and then creates an endpoint to communicate with the other thread using $mcapi\_endpoint\_create$. $Thread1$ gets $Thread2$'s endpoint by using $mcapi\_endpoint\_get$ function. $Thread1$ then sends a message to $Thread2$ and finalizes the MCAPI environment before exiting. $Thread2$ receives the message from $Thread1$ using non-blocking message receive function. In concurrent programs, the order in which threads are scheduled is nondeterministic. If $Thread1$ executes $mcapi\_msg\_send\_i$ before $Thread2$ executes $mcapi\_msg\_recv\_i$ then $Thread2$ receives the message from $Thread1$. However, if $Thread2$ executes $mcapi\_msg\_recv\_i$ before $Thread1$ executes $mcapi\_msg\_send\_i$, $Thread2$ returns from $mcapi\_msg\_recv\_i$ without receiving a message since there is no message available in its receive queue. $Thread2$ then waits until the message is received by using $mcapi\_wait$ function.

```
#define DOMAIN 1 #define NODE1 1 #define NODE2 2
#define PORT_NUM 100 #define NUM_THREADS 2

void* run_thread_1 (void *t) {
  ...
  mcapi_initialize(DOMAIN,NODE1,&parms,&version,&status);
  ep1=mcapi_endpoint_create(PORT_NUM,&status);
  ep2=mcapi_endpoint_get(DOMAIN,NODE2,PORT_NUM,
  MCA_INFINITE,&status);
  mcapi_msg_send_i(ep1,ep2,``MCAPI'',size,priority,
  &request,&status);
  mcapi_finalize(&status);
  ...
}
void* run_thread_2 (void *t) {
  ...
  mcapi_initialize(DOMAIN,NODE2,&parms,&version,&status);
  ep2 = mcapi_endpoint_create(PORT_NUM,&status);
  mcapi_msg_recv_i(ep2,buffer,BUFF_SIZE,&request,&status);
  ...
  mcapi_wait(&request,&recv_size,MCA_INFINITE,&status);
  mcapi_finalize(&status);
  ...
}
int main () {
  ...
  /* run all threads */
  pthread_create(&threads[0],NULL,run_thread_1,NULL);
  pthread_create(&threads[1],NULL,run_thread_2,NULL);
  /* wait for all threads */
  for (t = 0; t < NUM_THREADS; t++) {
      pthread_join(threads[t],NULL); }
  ...
}
```

Fig. 1. Multicore program using MCAPI

III. Automated Test Generation for MCAPI

We use mutation testing in order to enable test generation. Mutation testing is a software testing method that involves inserting faults (mutations) into user programs and then rerunning a test set against the mutated program, called a *mutant*. A mutant is *killed* (detected) by a test case that causes it to produce different output from the original program. The ratio of the number of mutants killed to the number of all mutants is called *mutation coverage*. Our goal is to generate tests that will improve mutation coverage in an automated manner. It is also important to know what types of faults can be inserted into the program. For this purpose, we leverage a mutation library and tool for MCAPI. These mutations allow us to insert concurrency related bugs into an MCAPI application such as nondeterminism, deadlock, race condition, starvation, resource exhaustion, incorrect parameter, incorrect function, or forgetting to use a function. For instance, when we modify $mcapi\_wait(10, request)$ with $mcapi\_wait(MCAPI\_INFINITE, request)$ in Fig. 2 mutation1, it results in a deadlock since ep2 waits for the message from ep1, whereas ep1 waits for the message from ep2. If we exchange $mcapi\_msg\_recv\_i$ with $mcapi\_msg\_recv$ in Fig. 2 mutation2, we cause a deadlock since ep1 waits for ep2 and ep2 waits for ep1.

Model checkers can exhaustively analyze the state space of a given program with respect to a given property. By performing symbolic execution along a particular path that leads to a state that violates the property and conjoining the predicates that

```
void* run_thread_1 (void *t) { /* Thread1 has ep1 */
  ...
  mcapi_msg_recv(ep1,buffer,BUFF_SIZE,&recv_size,&status);
  mcapi_msg_send(ep1,ep2,''msg1'',msgSize,priority,&status);
    ...
}
void* run_thread_2 (void *t) { /* Thread2 has ep2 */
  ...
  // mutation2
  mcapi_msg_recv_i(ep2,buffer,BUFF_SIZE,&request,&status);
  mcapi_wait(&request,&recv_size,10,&status); // mutation1
  mcapi_msg_send(ep2,ep1,''msg2'',msgSize,priority,&status);
  ...
}
```

Fig. 2. Inserting a mutation results in deadlock

guard its branch points, they can calculate the condition that the desired test input must satisfy. Then a satisfiability solver can be used to find a solution. In mutation testing based test generation, both the original program and the mutant are given to the model checker. The property to be checked is that the outputs of both programs are the same, which makes it an equivalence checking problem. The counterexample produced by the model checker is then the test case desired.

Some model checkers that can be used with MCAPI programs are SATABS [14] and CBMC [5]. SATABS transforms a C/C++ program into a Boolean program, which is an abstraction of the original program in order to handle large amounts of code. The Boolean program is then passed to a model checker such as SMV. SATABS can handle concurrent C programs, which makes it possible to be used for MCAPI applications. However, since SATABS ultimately abstracts a program into a Boolean program, it is challenging to convert the counterexamples generated for the Boolean program back to the original program's inputs. Another model checker, CBMC is a bounded model checker. Bounded model checking (BMC) is a variation of model checking which restricts the exploration to execution traces up to a certain length k. BMC either provides a guarantee that the first k execution steps of the program are correct with respect to the property P or a counterexample of length at most k. The ability to report counterexamples is the essential feature we use to generate test cases. However, CBMC cannot handle concurrent C programs. Also, model checkers are limited in terms of the size and complexity of the programs they can handle. Hence, often abstraction is used to make the programs amenable to model checking. In our case, the MCAPI library includes various complex C functions including shared memory manipulations, semaphores, and locks, resulting in unbounded loops, all of which make it difficult for CBMC to analyze the library. Hence, we abstracted the MCAPI library such that the essential message passing communication is kept and other implementation details in terms of shared memory operations are removed.

CBMC has been successfully used for automated test generation before and we also chose to use it in this paper. In order to use CBMC on a concurrent program, the thread structure is mapped into a state flow, that is into an equivalent sequential C program. Similar translation techniques are explored in [15], [16] for SystemC, which is also a concurrent language. Although the level of abstraction is high, the state flow simulates the blocking messages accurately.

TABLE I
EXPERIMENTAL RESULTS

Multicore program	#line	#ep	#mutants	#killed mutants	#tests
ex1	136	2	6	6	1
ex2	244	3	18	18	6
ex3	145	2	11	10	4

IV. AUTOMATED TEST GENERATION TOOL FOR MCAPI

We have developed an automated test generation tool that inserts all possible mutations to the multicore programs and then runs the model checker to generate counterexamples, if the mutant program can be killed by a test. Note that we can only mutate the constructs for which a mutation operator is defined. Given an MCAPI application we perform the following steps.

- Step 1: We create an abstract version of the program without implementation details and where the thread structure is mapped to a state flow.
- Step 2: We create a set of mutant programs.
- Step 3: We combine the mutant programs with that of the original program and add assertions to check the equivalence of the outputs of the original and the mutant program.
- Step 4: We execute the model checker and use the counterexamples as test cases.

We create an abstract version of the MCAPI application such that the thread structure is mapped into a state flow. We now show this step with an example in Fig. 3. This example is a transformation of the example in Fig. 1, where we used blocking messages and we removed wait, initialization, finalization and endpoint creation functions, which are not used for test generation purposes. Sending is triggered when both send and receive channels are at the READY state, changing the send status to RUNNING. When sending is completed the state is immediately set to COMPLETED, whereas the receiver can change from READY to RUNNING state. At this point, sender status is set READY in order to be triggered as soon as the receive process is completed and its state is set READY again. Next, the original function calls are replaced with wrapper function calls from mutation library, resulting in mutants. All mutations can be inserted into the program at the same time since a SAT solver can take advantage of incremental solution. However, we have not implemented that strategy in this paper. Then, we add assertions that check the equivalence of the outputs of two designs. Finally, the model checker is executed on the instrumented program.

We tested our framework on programs developed by us. Table I shows our experimental results for three examples. The column denoted by #ep shows the number of the endpoints created during the multicore program execution, the column denoted by #mutants gives the number of mutants. We also display the number of killed mutants, as well as the number of unique tests that were generated to kill these mutants in

```
// ORIGINAL
int transbuffer = 0; int msgr = 0;
int org (int* msgo) {
  unsigned char send_status = STATUS_READY;
  unsigned char recv_status = STATUS_READY;
  unsigned char runnable_count = 2;
  while ((runnable_count > 0))
      {
    // thread handling
    if ((send_status == STATUS_READY) &&
        (recv_status == STATUS_READY)) {
      send_status = STATUS_RUNNING;}
    else if ((send_status == STATUS_COMPLETED) &&
        (recv_status == STATUS_READY)) {
      recv_status = STATUS_RUNNING;
      send_status = STATUS_READY;}
    else if ((send_status == STATUS_READY) &&
        (recv_status == STATUS_COMPLETED)){
      recv_status = STATUS_READY;
      send_status = STATUS_READY;}
    // send
    if (send_status == STATUS_RUNNING){
      mcapi_send(msgo, &transbuffer);
      send_status = STATUS_COMPLETED;
      runnable_count--;}
    //receive
    if (recv_status == STATUS_RUNNING){
      mcapi_recv(&transbuffer, &msgr);
      recv_status = STATUS_COMPLETED;
      runnable_count--;}
  }
}
// MUTANT
int transbuffer_m=0 ;int msgr_m=0;
int mut (int* msgm)
{
  unsigned char send_status_m = STATUS_READY;
  unsigned char recv_status_m = STATUS_READY;
  unsigned char runnable_count_m = 2;
  while ((runnable_count_m > 0))
      {
    // thread handling
    if ((send_status_m == STATUS_READY) &&
        (recv_status_m == STATUS_READY)) {
      send_status_m = STATUS_RUNNING;}
    else if ((send_status_m == STATUS_COMPLETED) &&
        (recv_status_m == STATUS_READY)) {
      recv_status_m = STATUS_RUNNING;
      send_status_m = STATUS_READY;}
    else if ((send_status_m == STATUS_READY) &&
        (recv_status_m == STATUS_COMPLETED)){
      recv_status_m = STATUS_READY;
      send_status_m = STATUS_READY;}

    if (send_status_m == STATUS_RUNNING){
      mcapi_send(msgm, &transbuffer_m);
      send_status_m = STATUS_COMPLETED;
      runnable_count_m--;}

    if (recv_status_m == STATUS_RUNNING){
          // mutation removed receive
      //mcapi_recv(&transbuffer_m, &msgr_m);
      recv_status_m = STATUS_COMPLETED;
      runnable_count_m--;}
  }
}

int main () {
  int msg=nondet_int();
  org(&msg); mut(&msg);
// Compare the outputs
  __CPROVER_assert(msgr == msgr_m  , "equal" ) ;
  return 0; }
```

Fig. 3. MCAPI example

the table. Our framework was able to generate test inputs that kill every mutant but one mutant for program ex3. Upon further investigation, we realize that the resulting mutant is functionally the same as the original program, that is, the mutation is not activated in the observed execution. For the example in Fig. 3, CBMC generates a test input with msg value 1.

We did not hit a timeout for CBMC runs. This is partly because the examples were relatively small. It took 3 seconds on a Linux machine with 2 GB memory, to complete for the second example, which had the highest number of mutants. Since there is no publicly available benchmark using MCAPI, the scalability of our approach is not explored yet. Also, note that the same test can kill multiple mutants, however we display the unique test cases in this study.

V. Conclusions and Future Works

We presented the first time an automated test generation for multicore applications that use the newly emerging message passing MCAPI standard. Ultimately, our solution can improve the reliability of heterogeneous embedded multicore systems by aiding in the costly test generation process. Our framework made use of mutation testing for fault insertion, and model checking for test generation purposes. We experimentally demonstrated the applicability of our technique on simple examples, where we obtain high mutation coverage in an automated way. In the future, we plan to automate the process of code abstraction and transformation. We are working on additional techniques that can further enhance the usability and scalability of our tools. More importantly, MCAPI lacks larger benchmarks, hence we plan to work on developing benchmarks that will allow us to better measure the effectiveness of our technique.

Acknowledgments

This research was supported by Semiconductor Research Corporation under task 2082.001, Marie Curie International Reintegration Grant within the 7th European Community Framework Programme, BU Research Fund, and the Turkish Academy of Sciences.

References

[1] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister, "Software Standards for the Multicore Era," *Micro, IEEE*, vol. 29, no. 3, pp. 40 –51, may-june 2009.
[2] "Multicore Association, http://www.multicore-association.org," 2011.
[3] "Message Passing Interface, http://www.mcs.anl.gov/mpi/," 2011.
[4] G. Fraser, "Automated Software Testing with Model Checkers," Ph.D. dissertation, Graz University of Technology, Austria, 2007.
[5] D. Kroening, E. Clarke, and F. Lerda, " A tool for checking ANSI-C programs," in *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2004, pp. 168–176.
[6] D. Angeletti, E. Giunchiglia, M. Narizzano, A. Puddu, and S. Sabina, "Automatic test generation for coverage analysis using cbmc," in *Computer Aided Systems Theory - EUROCAST 2009*, 2009, pp. 287–294.
[7] N. He, P. Ruemmer, and D. Kroening, "Test-Case Generation for Embedded Simulink via Formal Concept Analysis," in *Proceedings of the Design Automation Conference (DAC)*, 2011.
[8] M. Abramovici, B. M. A., and A. D. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, New York, 1990.

[9] T. A. Budd, "Mutation analysis: Ideas, examples, problems and prospects," in *Computer Program Testing*. North-Holland, 1981, pp. 129–148.

[10] J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*. Kluwer Academic Publishers, 2001.

[11] J. Bradbury, J. Cordy, and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," in *Workshop on Mutation Analysis*, Nov. 2006, p. 11.

[12] N. Bombieri, F. Fummi, and G. Pravadelli, "A Mutation Model for the SystemC TLM 2.0 Communication Interfaces," in *Proceedings of the Conference on Design Automation and Test in Europe (DATE)*. ACM, 2008, pp. 396–401.

[13] A. Sen and M. S. Abadir, "Coverage Metrics for Verification of Concurrent SystemC Designs Using Mutation Testing," in *Proceedings of the IEEE International High-Level Design Validation and Test Workshop (HLDVT)*, 2010.

[14] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, vol. 3440, 2005, pp. 570–574.

[15] D. Grosse, H. M. Le, and R. Drechsler, "Proving Transaction and System-level Properties of Untimed SystemC TLM Designs," in *Proceedings of the International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, 2010, pp. 113–122.

[16] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: A software model checking approach," in *Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, 2010, pp. 51–59.