Verification Coverage of Embedded Multicore Applications

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

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

Jim Holt
Freescale Semiconductor Inc.
Austin, TX, USA
Email: jim.holt@freescale.com

***Abstract*—Verification of embedded multicore applications is crucial as these applications are deployed in many safety critical systems. Verification task is complicated by concurrency inherent in such applications. We use mutation testing to obtain a quantitative verification coverage metric for mullticore applications developed using the new Multicore Communication API (MCAPI) standard. MCAPI is a lightweight API that targets heterogeneous multicore embedded systems. We developed a mutation coverage tool and performed several experiments on MCAPI applications. Our experiments show that mutation coverage is useful in measuring and improving the quality of the test suites and ultimately the quality of the multicore application.**

I. Introduction

Multicore applications are becoming common place. Several new standards are being developed to enable the deployment of multicore applications on multicore hardware. In this paper, we develop verification coverage techniques for one of these standards, namely Multicore Communication API (MCAPI) [1], [2] by the Multicore Association. There are currently 33 industrial members of this association.

MCAPI aims to supply communication and synchronization between closely distributed embedded systems using message passing communication. Shared memory used by multicore systems can lead to nondeterminism. Message passing reduces the potential for nondeterminism by explicit messages for communication. MCAPI is a message passing API like the well-known MPI [3] but its target system and functionalities differ from MPI. MCAPI provides low latency and low overhead for heterogeneous embedded platforms (in terms of types and number of cores, different operating systems, and physical transports). MCAPI provides sufficient number of functionalities while hiding or minimizing communication overhead to get better performance. There are two open source implementations of the MCAPI standard; first by the Multicore Association [1], and the second by Mentor Graphics [4].

Coverage metrics help engineers determine when the verification task can be completed. Our coverage framework makes use of mutation testing [5], [6]. 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) inserted in the program. We developed a mutation library for MCAPI in this paper. These mutations focus on communication constructs in the standard, thereby allowing us to test concurrency related bugs as well as improving the coverage performance by not focusing on mutations at lower levels of abstraction. Mutation testing based coverage metrics have earlier been developed for applications written in different languages such as Java, C, SystemC, and Simulink [7], [8], [9].

978-3-9810801-8-6/DATE12/©2012 EDAA

II. Background on MCAPI

Basic elements of the MCAPI topology are nodes, which can be a process, a thread, a hardware accelerator, etc. Communication occurs between endpoints, which are termination points and created on nodes on each side of the communication. MCAPI has three fundamental communication types: connectionless datagrams for messages; connection-oriented packet streams for packet and 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, however our coverage 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.

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 $mcapi\_wait$ function 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

```
#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

function calls using $mcapi\_cancel$ function.

We show an example multicore program that uses MCAPI in Fig. 1. The program has two concurrent threads ($Thread1$ and $Thread2$) communicating through connectionless non-blocking message exchange. Note that the current implementations of MCAPI supports message passing using shared memory. 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 this concurrent program, 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.

III. Mutation Testing based Coverage for MCAPI

In order to increase confidence in verification results, we need a test suite that covers all possible behaviors of a given multicore program. There are several coverage metrics including line, branch, toggle, and FSM coverage among others. We use mutation testing based coverage. This is an observability-based coverage that allows to measure the impact of faults on the outputs of the design. Mutation testing is a software testing method that involves inserting faults (mutations) into user programs (obtaining mutants) and then re-running a test suite against the mutants. A mutant is *killed* (detected) by a test case that causes it to produce different output from the original multicore program. The ratio of the number of mutants killed to the number of all mutants is called *mutation coverage*.

We now demonstrate mutation coverage metric on a mutant obtained from the program in Fig. 1. First, we generate a mutant program by removing $mcapi\_wait$ function from $Thread2$'s function body. Our test suite has one test (test1) which checks the value of the $buffer$ variable variable in $Thread2$. We run both the original and the mutant programs. If $Thread1$ executes and exits and then $Thread2$ executes and exits, both programs produce the same value "$MCAPI$" for the $buffer$. This result shows that test1 can not detect this mutant, therefore we add a new test (test2) which checks the validity of the $request$ variable. Note that request variable stays true until the wait operation is completed, that is when the receive operation is completed. In this case, the original program produces false and the mutant program produces true using test2, since the wait operation has been deleted. Hence, we could improve the verification coverage by adding a new test to the test suite.

We now describe our mutation library. Note that our library includes 88 mutation operators for 40 MCAPI functions, some of which we explain here. Common bugs for concurrent message passing programs include nondeterminism, deadlock, race condition, starvation, and resource exhaustion. In addition to these bugs, it is also common to forget functions, pass incorrect parameters to functions or use incorrect functions in both concurrent or sequential programs. We developed the following set of mutation operators for MCAPI functions that will trigger these common bugs.

- Remove Communication Function (RCF): This operator removes calls to communication functions. For example, if we remove $mcapi\_wait$ from the multicore program displayed in Fig. 2, it leads to a message race condition on ep3 between ep1 and ep2.
- Modify Function Timeout (MFT): This operator changes the timeout value of the function and can be applied to $mcapi\_wait$ and $mcapi\_wait\_any$ functions since these are the only functions with timeout parameters. For example, we can modify $mcapi\_wait(time)$ to $mcapi\_wait(time * 2)$, $mcapi\_wait(time/2)$, or $mcapi\_wait(MCAPI\_INFINITE)$. These modifications may result in nondeterminism, deadlock, or race condition. For instance, when we modify $mcapi\_wait(10, request)$ with $mcapi\_wait(MCAPI\_INFINITE, request)$ in mutation1 of Fig. 3, it results in a deadlock since ep2 waits for the message from ep1, whereas ep1 waits for the message from ep2.
- Exchange Function Call with Another (EFC): This operator exchanges a communication function with another appropriate function. For example, we can exchange a blocking function with a non-blocking function such as $mcapi\_msg\_send$ and $mcapi\_msg\_send\_i$. This operator may lead to nondeterminism, deadlock, or starvation. If

```
void* run_thread_1 (void *t) { /* Thread1 has ep1 */
  mcapi_msg_send(ep1,ep3,``msg13'',size,prio,&status);
  mcapi_msg_send_i(ep1,ep2,``msg12'',size,prio,
  &request,&status);
}
void* run_thread_2 (void *t) { /* Thread2 has ep2 */
  mcapi_msg_recv_i(ep2,buffer,BUFF_SIZE,&request,&status);
  // mutation
  mcapi_wait(&request,&recv_size,MCAPI_INFINITE,&status);
  mcapi_msg_send(ep2,ep3,``msg23'',size,prio,&status);
}
void* run_thread_3 (void *t) { /* Thread3 has ep3 */
  mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);
  mcapi_msg_recv(ep3,buffer,BUFF_SIZE,&recv_size,&status);
}
```

Fig. 2. Inserting a mutation results in race condition

```
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'',size,prio,&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'',size,prio,&status);
}
```

Fig. 3. Inserting a mutation results in deadlock

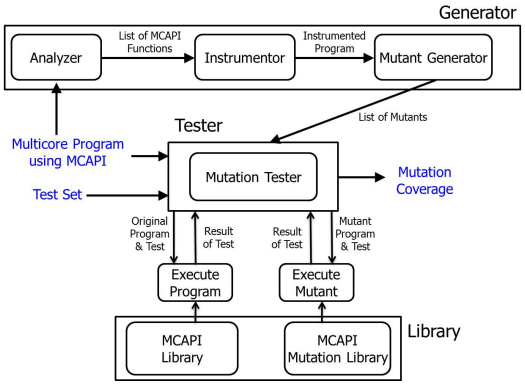

Fig. 4. Overview of Mutation Coverage Tool Architecture

we exchange $mcapi\_msg\_recv\_i$ with $mcapi\_msg\_recv$ in Fig. 3 mutation2, we cause a deadlock since ep1 waits for ep2 and ep2 waits for ep1.

IV. Experimental Results

We have developed an automated tool that inserts all possible mutations to a given multicore program and then checks for each of the mutant programs whether it is killed by any of the tests in the test suite.

Our mutation coverage tool is shown in Fig. 4. Our tool records the locations (function name, source file path, and line number) of MCAPI functions by statically analyzing the source code and then automatically replaces original function calls with wrapper function calls in order to handle mutation operations in wrappers. In each wrapper function, we check the mutation parameters (source file name, line number, mutation type) that are passed to the function and if they match with

```
void mcapi_mut_msg_recv_i(char* file, mcapi_uint32_t line,
  mcapi_endpoint_t recv_ep, void* buff, size_t
  b_size, mcapi_request_t* req, status_t* status) {
  size_t recv_size = 0;
  if (line == mut_line && strcmp(file, mut_file) == 0) {
    switch (mut_type) {
    case 1: /* remove */
      *status = MCAPI_ERR_MUTATION;
      break;
    case 2: /* exchange with blocking */
      mcapi_msg_recv(recv_ep,buff,b_size,&recv_size,status);
      break;
    default:
      mcapi_msg_recv_i(recv_ep,buff,b_size,req,status);
      break;
    }
  } else
    mcapi_msg_recv_i(recv_ep,buff,b_size,req,status);
}
```

Fig. 5. $mcapi\_mut\_msg\_recv\_i$ function from our mutation library

the current function then we activate the mutant, otherwise this function directly calls the original library function. We use the tool CIL [10] in the generator module. Figure 5 shows part of the $mcapi\_mut\_msg\_recv\_i$ function from our MCAPI mutation library.

We tested our tool on multicore programs that cover message, packet channel, scalar channel operations of MCAPI as well as blocking and non-blocking operation types. The first five multicore programs in Table I are from the Multicore Association and the remaining ones are developed by us because no publicly available MCAPI benchmarks are currently available. In the table, we denote the number of endpoints in column #ep, generated mutants in column #Mutants, the number of killed mutants in #KilledMut, the mutation coverage in Cov, and the total time that is consumed for mutant generation and execution of all mutants in Runtime (seconds) column. All the experiments were performed on a PC running Linux with a CPU of 800MHz and 4GB of memory. For each program, we ran the experiment 100 times and averaged the output results.

For each program, we manually developed three test cases and performed three sets of experiments in order to obtain mutation coverage as well as to analyze the impact of adding new tests on coverage. The first experiment set uses a single test case for each program that checks the exit code of the given multicore program. The second experiment set adds a new test for each program on top of the test in experiment set 1. For example, for pv2 program, which is the program in Fig. 1, the new test checks whether the message buffer of ep2 contains the message that is expected from ep1. Similarly, the third experiment set adds a test for each program on top of the tests in experiment set 2. Again, for pv2, the new test checks whether the request is valid or not on exit.

Different mutation coverage percentages and runtimes are expected due to the fact that these results are heavily influenced by the program. In practice, a target coverage percentage can be provided by the user. Experimental results show that mutation coverage increases with the number of new test cases, that is, the mutation coverage is over %35 with one test, over %50 with two tests, and over %70 with three tests. Whereas,

TABLE I
EXPERIMENTAL RESULTS WITH DIFFERENT NUMBER OF TESTS

				Single test			Two tests			Three tests		
Multicore program	#line	#ep	#Mutants	#KilledMut	Cov	Runtime	#KilledMut	Cov	Runtime	#KilledMut	Cov	Runtime
msg2	186	2	17	11	65	0.562	13	76	0.592	14	82	0.601
msg11	374	2	26	12	46	0.308	13	50	0.348	19	73	0.358
pkt5	402	2	34	13	38	0.294	17	50	0.355	24	70	0.360
scl1	451	8	91	57	63	0.905	60	66	0.957	66	73	0.973
multiMessage	419	12	20	10	50	1.112	12	60	1.215	15	75	1.253
pv1	200	16	11	6	55	0.297	7	64	0.341	8	73	0.357
pv2	156	2	25	15	60	0.612	17	68	0.640	18	72	0.651
drc1	183	32	17	14	82	1.115	15	88	1.217	15	88	1.223
drc2	189	3	23	12	48	0.070	15	65	0.074	16	70	0.077
drc3	200	32	11	7	64	0.411	8	73	0.433	9	82	0.441
rc1	233	3	27	16	59	0.685	17	63	0.715	20	74	0.734

the running time of our tool stays almost the same when new test cases are added and it takes less than 1.3 seconds even for three tests. For instance, our tool generated 91 mutants for scl1 and 57 of them are killed by one test in less than one second.

We now analyze how adding new tests increases the mutation coverage in the case of program pv2. The second test for pv2 kills two mutants, hence increasing the number of killed mutants from 15 to 17. Specifically, this test kills the mutant obtained after removing $mcapi\_wait$ in ep2, because when ep2 receives before ep1 sends the message, the message buffer will not contain the sent message since there will not be any waiting for the message. The second test also kills the mutant obtained by modifying the function timeout in $mcapi\_wait$ in ep2 from $MCAPI\_INFINITE$ to say 10. This is similar to the previous case except that ep2 is going to wait but for a very short time, hence not receiving the message sent by ep1. The third test kills one mutant, increasing the number of killed mutants from 17 to 18. This test kills the mutant obtained after exchanging $mcapi\_wait$ with $mcapi\_test$, because the request can become invalid only after the message is received, but when the non-blocking receive happens before the send, the request is still valid.

We note that the running time increases if an actual deadlock occurs when a mutant is executed. In order to detect deadlocks in a mutant, we used a timeout approach, which declares a deadlock if a specified time period has elapsed. The program drc1 has one of the maximum running times overall and we know that many of the generated mutants for this program results in actual deadlocks.

We obtained low coverage for programs where the injected mutation code is not triggered in the observed execution schedule, which is the case for pkt5. Also, the mutants can potentially lead to different execution schedules than the original programs, which is useful in detecting schedule dependent errors. For instance, pv2 has a mutant that uses a blocking message receive call instead of non-blocking, leading to a potential schedule change. The send operation in the mutant always completes before the receive operation since the receive operation blocks $Thread2$ until a matching send is called. Hence, depending on the test case and the schedule, the mutant may or may not be killed.

V. CONCLUSIONS AND FUTURE WORKS

We obtained a novel verification coverage solution for embedded multicore applications that use the newly emerging MCAPI standard. We presented new mutation operators for communication constructs in MCAPI and related them to actual concurrency bug patterns. We developed an automated mutation testing based coverage tool which can be seamlessly integrated with current multicore applications. Our experimental results demonstrate the effectiveness of mutation testing based coverage and confirm the necessity of developing test suites for checking concurrent features of MCAPI applications. Ultimately, the quality of the test suite and the multicore application can be improved due to our coverage approach. Our framework can also be used to optimize the test suite by removing redundant test cases that kill the same set of mutants. It would also be interesting to generate test cases in an automated manner when the coverage is low.

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] "Multicore Association, http://www.multicore-association.org," 2011.
[2] 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.
[3] "Message Passing Interface, http://www.mcs.anl.gov/mpi/," 2011.
[4] "Mentor Graphics, http://www.mentor.com/embedded-software/," 2011.
[5] T. A. Budd, "Mutation analysis: Ideas, examples, problems and prospects," in *Computer Program Testing*. North-Holland, 1981, pp. 129–148.
[6] J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*. Kluwer Academic Publishers, 2001.
[7] J. Bradbury, J. Cordy, and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," in *Workshop on Mutation Analysis*, Nov. 2006, p. 11.
[8] 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.
[9] 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.
[10] "C Intermediate Language (CIL), http://cil.sourceforge.net/," 2011.