CRI: Symbolic Debugger for MCAPI Applications*

Mohamed Elwakil[1], Zijiang Yang[1], and Liqiang Wang[2]

[1] Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008
[2] Department of Computer Science, University of Wyoming, Laramie, WY 82071
{mohamed.elwakil,zijiang.yang}@wmich.edu, wang@cs.uwyo.edu

Abstract. We present a trace-driven SMT-based symbolic debugging tool for MCAPI (Multicore Association Communication API) applications. MCAPI is a newly proposed standard that provides an API for connectionless and connection-oriented communication in multicore applications. Our tool obtains a trace by executing an instrumented MCAPI. The collected trace is then encoded into an SMT formula such that its satisfiability indicates the existence of a reachable error state such as an assertion failure.

Keywords: MCAPI, Message Race, Symbolic Analysis, Satisfiability Modulo Theories.

1 Introduction

As multicore-enabled devices are becoming ubiquitous, development of multicore applications is inevitable, and debugging tools that target multi-core applications will be in demand. Inter-core communication, in which data is passed between cores via messages, is an essential part of multicore applications. The Multicore Association has developed the MCAPI standard [1] and a reference runtime implementation for it to address inter-core communication needs. In an MCAPI application, a core is referred to as a *node*. Communication between nodes occurs through endpoints. A node may have one or more endpoints. An endpoint is uniquely defined by a node identifier, and a port number. The MCAPI specification supplies APIs for initializing nodes, creating endpoints, obtaining addresses of remote endpoints, and sending and receiving messages. Fig. 1 shows a snippet from an MCAPI application in which four cores communicate via messages. For brevity, the variables declarations and initialization calls are omitted. In this application, each node has one endpoint; hence there is no need for different port numbers per node. Endpoints are created by issuing the *create_ep* calls (e.g. lines 5 and 24). To obtain the address of a remote endpoint, the *get_ep* calls are used (e.g. lines 6 and 25). Messages are sent using the *msg_send* call (e.g. lines 8 and 27). Messages are received using the *msg_recv* call (e.g. lines 16 and 34).

* The work was funded in part by NSF grant CCF-0811287 and ONR Grant N000140910740.

A. Bouajjani and W.-N. Chin (Eds.): ATVA 2010, LNCS 6252, pp. 353–358, 2010.

```
1  #define PORT 1
2  void* C1_routine (void *t)
3  {
4   int Msg=1;
5   My_ep=create_ep(PORT);
6   C2_ep = get_ep(2,PORT);
7   C4_ep = get_ep(4,PORT);
8   msg_send(My_ep,C2_ep,Msg); //M0
9   msg_send(My_ep,C4_ep,Msg); //M1
10 }
11 void* C2_routine (void *t)
12 {
13  int X,Y,Z;
14  My_ep = create_ep(PORT);
15  C4_ep = get_ep(4,PORT);
16  msg_recv(My_ep,X);
17  msg_recv(My_ep,Y);
18  Z=X-Y;
19  msg_send(My_ep,C4_ep,Z); //M2
20 }
21 void* C3_routine (void *t)
22 {
23  int Msg=10;
24  My_ep = create_ep(PORT);
25  C2_ep = get_ep(2,PORT);
26  C4_ep = get_ep(4,PORT);
27  msg_send(My_ep,C2_ep,Msg); //M3
28  msg_send(My_ep,C4_ep,Msg);//M4
29 }
30 void* C4_routine (void *t)
31 {
32  int U;
33  My_ep = create_ep(PORT);
34  msg_recv(My_ep,U);
35  assert(U>0);
36 }
```

Fig. 1. A snippet of an MCAPI application

The MCAPI runtime provides each endpoint with FIFO buffers for incoming and outgoing messages. Messages sent from an endpoint S to another endpoint D, are delivered to D according to their order of transmission from S. However, the order at which a destination endpoint D, receives messages originating from endpoints $S1$, and $S2$, is *non-deterministic*, even if endpoints $S1$ and $S2$ belong to the same node. Two or more messages are said to be *racing* if their order of arrival at a destination (i.e. a core) is non-deterministic [2]. The *msg_recv* calls specify only the receiving endpoint, which is the reason for the possibility of message races. In Fig. 1, messages *M0* and *M3* are racing towards core *C2* and messages *M1, M2*, and *M4* are racing towards core *C4*. There are twelve possible scenarios for the order of arrival of messages at *C2* and *C4*. In only two scenarios (when *M0* beats *M3,* and *M2* beats *M1* and *M4*), there will be an assertion failure at *C4*.

Testing the application in Fig. 1 by multiple executions does not necessarily expose the single scenario that leads to an assertion failure. Even if an assertion failure takes place during testing, it is very difficult to find out the specific order of messages arrival that caused the assertion failure.

In this paper we present our tool (CRI) that *symbolically* explores all possible orders of messages arrival in an MCAPI application execution. Our approach is based on encoding the trace of an execution as an SMT formula in quantifier-free first order logic. This formula can be decided efficiently using any of the available SMT solvers such as Yices [3].

Our main contributions are 1) the modeling of MCAPI constructs as SMT constraints, and 2) developing a tool that generates these constraints from an MCAPI application execution trace. Our tool predicts errors that may not be discovered by testing and guides an execution to reproduce the same error when an error is detected. The rest of this paper is organized as follows: Section 2 describes our tool in detail. Section 3 reviews related work. We conclude in Section 4.

2 CRI

Fig. 2 shows the workflow of our tool. An application source code is instrumented so that an execution will produce a trace that contains all the statements that has been executed such as MCAPI functions calls and C statements.

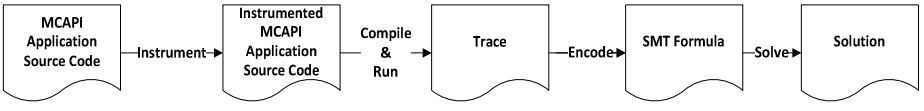

Fig. 2. CRI Workflow

The trace is then encoded as an SMT formula. If the formula is satisfiable, then there is a reachable error state (e.g. an assertion failure) and the SMT solver solution will provide enough information to guide a controlled execution to reach this error state. Otherwise, we can conclude that for all possible execution scenarios involving the same statements in the input trace, no error state is reachable. In the following we describe the structure of the trace, the variables used in the encoding, and present the encoding of some statements.

A trace consists of a list of nodes: $\mathbb{N} = \{N_1, \dots, N_{|\mathbb{N}|}\}$, and for every node N_{nid}, a set of local variables: $\mathbb{L}_{nid} = \{L_{nid,1}, \dots, L_{nid,|\mathbb{L}_{nid}|}\}$, a set of endpoints used in this node: $\mathbb{EP}_{nid} = \{EP_{nid,1}, \dots, EP_{nid,|\mathbb{EP}_{nid}|}\}$, and an ordered list of statements: $\mathbb{S}_{nid} = \{S_{nid,1}, \dots, S_{nid,|\mathbb{S}_{nid}|}\}$.

For a trace with B statements, there will be $B + 1$ symbolic states ($\mathcal{s}^0$, $\mathcal{s}^1$, …, $\mathcal{s}^i$, … $\mathcal{s}^B$), such that $\mathcal{s}^0$ is the state before carrying out any statement, and $\mathcal{s}^i$ is the state at the ith time instant, after carrying out the ith statement. A state $\mathcal{s}^i$ is a valuation of all symbolic variables at time instant i. To capture the $B + 1$ states, we create $B + 1$ copies for the variables in the trace. For example, $L^i_{nid,x}$ denotes the copy of variable $L_{nid,x}$ at the ith time instant.

At any instant of time, one statement, called the *pending statement*, at one node, called the *active node*, will be *symbolically* carried out. The node selector variable NS^i identifies the node that will be active at time instant i. At any time instant i, the value of NS^i is selected by the SMT solver. The selection of NS^i value is not totally random, but is governed by scheduling constraints.

The pending statement in a node N_{nid} is identified using the node counter variable NC_{nid}. The domain of a NC_{nid} is $\{1\dots|\mathbb{S}_{nid}|, \perp\}$. NC_{nid}=x indicates that the pending statement in the node N_{nid} is $S_{nid,x}$. $NC_{nid} = \perp$ means that all statements in node N_{nid}, has been symbolically executed.

The MCAPI runtime buffers associated with endpoints are modeled as queues. For a receiving endpoint $EP_{n,x}$ that receives a message or more, there will be a corresponding queue $Q_{n,x}$. $\mathbb{Q}_n$ is the set of all queues needed for the receiving endpoints at node N_{nid}. A queue $Q_{n,x}$ is encoded as an array with two variables $head_{n,x}$ and $tail_{n,x}$ that indicate the head and tail positions in the array.

The MCAPI standard provides non-blocking send and receive calls: *msg_send_i*, and *msg_recv_i*, respectively. MCAPI runtime uses request objects to track the status of a non-blocking call. A non-blocking call initiates an operation (i.e. a send or a receive operation), sets a request object to pending, and returns immediately. The completion of a non-blocking call could be checked by issuing the blocking call *wait*, and passing to it the request object associated with the non-blocking call. The *wait* call will return when the non-blocking call has completed. A non-blocking send is completed when the message has been delivered to the MCAPI runtime. A non-blocking receive is completed when a message has been retrieved from the MCAPI runtime buffers. A request object will be encoded as a symbolic variable with three possible values: NLL, PND, and CMP.

Four constraints formulas make up the SMT formula: the initial constraint ($\mathbb{I}$), the statements constraint ($\mathbb{S}$), the scheduling constraint ($\mathbb{C}$), and the property constraint ($\mathbb{P}$). The initial constraint ($\mathbb{I}$) assigns the values of the symbolic variables at time instant *0*. All node counters are initialized to 1. $\mathbb{I}$ is expressed as $\bigwedge_{n=1}^{|\mathbb{N}|}\Big((NC_n^0 = 1) \wedge \Big(\bigwedge_{v=1}^{|\mathbb{L}_n|} L_{n,v}^0 = iv_{n,v}\Big) \wedge \Big(\bigwedge_{q=1}^{|\mathbb{Q}_n|} head_{n,q}^0 = tail_{n,q}^0 = 0\Big)\Big)$, where $iv_{n,v}$ is the initial value for the variable $L_{n,v}$. Note that the request variables used in a node N_{nid}, are among the node local variables ($\mathbb{L}_n$), and that they are initialized to NLL in the initial constraint.

The statements constraint ($\mathbb{S}$) mimics the effect of carrying out a pending statement. It is a conjunction of B constraints ($\mathbb{S} = \bigwedge_{i=1}^{B} \mathbb{s}^i$), such that $\mathbb{s}^i$ corresponds to the statement chosen to be carried out at time instant i. The $\mathbb{s}^i$ constraint is dependent on the statement type. Our tool handles eight types of statements: assignment, conditional, assert, blocking send, blocking receive, non-blocking send, non-blocking receive, and wait statements. In this paper we present the encoding of five types, and omit the rest.

1) For an assignment statement in the format of S=('assign', nid, sn, sn', v, $Expr$), where nid is the identifier of the node to which S belongs, sn is the node counter of *S,* sn' is the node counter of the statement to be executed next in this node ($\perp$ if S is the last statement in $\mathbb{S}_{nid}$), and $Expr$ is an expression whose valuation is assigned to variable v, the corresponding constraint formula is $\big(NS^i = nid \wedge NC_{nid}^i = sn\big) \rightarrow (NC_{nid}^{i+1} = sn' \wedge v^{i+1} = Expr^i \wedge \delta(\{v^i\}))$. This formula states that, at time instant i, if node N_{nid} is the active node ($NS^i = nid$) and node N_{nid}'s node counter is equal to sn ($NC_{nid}^i = sn$), then the node counter in the time instant $i+1$ is set to sn' ($NC_{nid}^{i+1} = sn'$), the value of variable v in the time instant $i+1$ is set to the valuation of the expression $Expr$ at time instant i ($v^{i+1} = Expr^i$), and that all local variables but v and all queues heads and tails should have in time instant $i+1$, the same values they had in time instant i ($\delta(\{v^i\})$).

2) For a conditional statement in the format of S=('condition', nid, sn, sn', $Expr$), the corresponding constraint formula is $\big(NS^i = nid \wedge NC_{nid}^i = sn\big) \rightarrow (Expr^i \wedge NC_{nid}^{i+1} = sn' \wedge \delta(\phi))$. Note that we enforce the condition $Expr$ to be true so only the executions with the same control flow in this node are considered. $\delta(\phi)$ states that all variables retain their values from time instant i to $i+1$.

3) For a blocking send statement in the format of S=('send_b', nid, sn, sn', $SrcEP_{nid,x}, DestEP_{uid,y}, Exp$), where $SrcEP_{nid,x}$ is the source endpoint, $DestEP_{uid,y}$ is the destination endpoint, and Exp is the expression whose valuation is being sent, the corresponding constraint formula is $(NS^i = nid \wedge NC_{nid}^i = sn) \rightarrow (NC_{nid}^{i+1} = sn' \wedge Q_{uid,y}[tail_{uid,y}^i] = Exp^i \wedge tail_{uid,y}^{i+1} = tail_{uid,y}^i + 1 \wedge \delta(\{tail_{uid,y}^i\}))$. This formula states that, at time instant i, if node N_{nid} is the active node and N_{nid}'s node counter is equal to sn, then the node counter in the time instant $i+1$ is set to sn', the valuation of the sent expression is enqueued to the destination endpoint queue ($Q_{uid,y}[tail_{uid,y}^i] = Exp^i \wedge tail_{uid,y}^{i+1} = tail_{uid,y}^i + 1$), and all local variables and all queues heads and tails but $tail_{uid,y}^i$ should have in time instant $i+1$, the same values they had in time instant i ($\delta(\{tail_{uid,y}^i\})$).

4) For a blocking receive statement in the format of S=('recv_b', nid, sn, sn', $RecvEP_{nid,x}, v$), the corresponding constraint formula is $(NS^i = nid \wedge NC_{nid}^i = sn) \rightarrow (NC_{nid}^{i+1} = sn' \wedge v^{i+1} = Q_{nid,x}[head_{nid,x}^i] \wedge head_{nid,x}^{i+1} = head_{nid,x}^i + 1 \wedge \delta(\{head_{nid,x}^i, v^i\}))$. In this formula, the relevant queue is dequeued, and the dequeued value is assigned to the receiving variable ($v^{i+1} = Q_{nid,x}[head_{nid,x}^i] \wedge head_{nid,x}^{i+1} = head_{nid,x}^i + 1$).

5) For a non-blocking send statement in the format of S=('send_nb', nid, sn, sn', $SrcEP_{nid,x}, DestEP_{uid,y}, Exp, R$), such that R is the request variable associated with S, the corresponding constraint formula is $(NS^i = nid \wedge NC_{nid}^i = sn) \rightarrow (NC_{nid}^{i+1} = sn' \wedge Q_{uid,y}[tail_{uid,y}^i] = Exp^i \wedge tail_{uid,y}^{i+1} = tail_{uid,y}^i + 1 \wedge R^{i+1} = PND \wedge \delta(\{R^i, tail_{uid,y}^i\}))$. In addition to enqueuing the valuation of the sent expression, the value of the request variable is set to pending ($R^{i+1} = PND$).

Like the statements constraint, the scheduling constraint ($\mathbb{C}$) is the conjunction of B constraints ($\mathbb{C} = \bigwedge_{i=1}^{B} \mathbb{c}^i$). Each $\mathbb{c}^i$ constraint consists of four parts which ensure that 1) a node that is done carrying out all its statements, will not be an active node, 2) the variables of an inactive node will not change, 3) a blocking receive will not take place if the relevant queue is empty, and 4) a wait associated with a non-blocking receive, will not take place if the relevant queue is empty. Due to the limited space, we present only the first ($\mathbb{c}_{done}^i$) and the third ($\mathbb{c}_{blocked\_recvb}^i$) parts of the scheduling constraint. $\mathbb{c}_{done}^i$ is expressed as: $\bigwedge_{n=1}^{|\mathbb{N}|}[(NC_n^i = \perp) \rightarrow (NS^i \neq n)$. This formula states that: when all the statements in a node have been executed ($NC_n^i = \perp$), then this node can't be an active node ($NS^i \neq n$). $\mathbb{c}_{blocked\_recv}^i$ is expressed as $\bigwedge_{n=1}^{|\mathbb{N}|}[((NC_n^i = A_{n,x}) \wedge (A_{n,x} = ('\text{m\_recv\_b}, A_{n,x}, EP_{recv}, v\,)) \wedge (head_{n,EP_{recv}}^i = tail_{n,EP_{recv}}^i)) \rightarrow (NS^i \neq n)]$. This formula states that: if the pending action in a node is a blocking receive ($(NC_n^i = A_{n,x}) \wedge (A_{n,x} = ('\text{m\_recv\_b}, A_{n,x}, EP_{recv}, v\,))$) and the relevant queue is empty ($head_{n,EP_{recv}}^i = tail_{n,EP_{recv}}^i$), then this node can't be the active node ($NS^i \neq n$).

The property constraint ($\mathbb{P}$) is derived from either a user-supplied application-specific property expressed as an assert statement, or from a built-in safety property such as "No message races exists". For an assert statement in the format of S=('assert', nid, sn, sn' ,$Expr$), the corresponding property constraint, is $(NS^i = nid \wedge NC_{nid}^i = sn) \rightarrow (Expr^i = true)$.

The overall formula that is passed to the SMT solver is the conjunction of the initial constraint, the statements constraint, the scheduling constraint and the negation of property constraint, and is expressed as $\mathbb{I} \wedge \mathbb{S} \wedge \mathbb{C} \wedge \sim \mathbb{P}$.

3 Related Work

To the best of our knowledge, the only other tool for analyzing MCAPI applications is MCC [4]. MCC explores all possible orders of messages arrival by repeated executions. It creates a scheduling layer above the MCAPI runtime, which allows MCC to discover all potentially matching send/receive pairs by intercepting calls to the MCAPI runtime. In [5], C. Wang et al. introduce a symbolic algorithm that detects concurrency errors in all feasible permutations of statements in an execution trace. They use concurrent static single assignment (CSSA) based encoding to construct an SMT formula. The algorithm has been applied to detect concurrency errors in shared memory multithreaded C programs.

4 Conclusion

We have presented CRI, a tool for symbolically debugging MCAPI applications. Our tool builds an SMT formula that encodes the semantics of an MCAPI application execution trace. By such analysis we are able to detect and reproduce errors that may not be discovered by traditional testing approaches. Due to the lack of publicly available MCAPI benchmarks, we performed experiments on MCAPI applications developed by ourselves. For example, the full code of the application in Fig. 1 was found to have an assertion failure in 0.03 seconds using Yices [3] as the SMT solver. We plan to extend our tool to support connection-oriented calls, and investigate optimizations to improve the performance.

References

1. The Multicore Association Communications API, `http://www.multicore-association.org/workgroup/mcapi.php`
2. Netzer, R.H.B., Brennan, T.W., Damodaran-Kamal, S.K.: Debugging Race Conditions in Message-Passing Programs. In: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, SPDT 1996, PA, USA, pp. 31–40 (1996)
3. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
4. Sharma, S., Gopalakrishnan, G., Mercer, E., Holt, J.: MCC: A runtime verification tool for MCAPI applications. In: 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, Austin, Texas, USA, November 15-18 (2009)
5. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic Predictive Analysis for Concurrent Programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)