Debugging Support Tool for MCAPI Applications

Mohamed Elwakil
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008
mohamed.elwakil@wmich.edu

Zijiang Yang
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008
zijiang.yang@wmich.edu

ABSTRACT
The recently proposed MCAPI (Multicore Association Communication API) specification provides multicore programs developers with a standard API for inter-core messages-based communication and stream-based communication. Debugging MCAPI programs that use message-based communication is expected to be very challenging due to the non-determinism associated with the order of messages' arrivals at a core. In this paper we present a tool that uses an off-the-shelf SMT solver to symbolically explore all possible orders of messages arrival in an MCAPI program. If there is a specific order of messages arrival that leads to an error state, this order is presented to the user as a debugging aid.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Model checking; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms
Reliability, Verification

Keywords
MCAPI, message race, symbolic analysis, Satisfiability Modulo Theory, multicore programs debugging

1. INTRODUCTION AND MCAPI OVERVIEW
Desktop computers and portable devices are increasingly using multicore chips. With multicore-equipped devices becoming commonplace, there is an urgent need for standards and tools to support multi-core applications development. The Multicore Association [8] has developed the MCAPI specification [9] and is developing the MRAPI specification [10] to address two basic needs for multi-core applications. The MCAPI specification provides an API library and a reference implementation for inter-core communication. The MRAPI specification will provide an API library for coordinating concurrent access to shared resources such as memory regions.

PADTAD'10, July 13, 2010, Trento, Italy.

MCAPI is radically different from MPI [5]. MPI is intended for inter-computer communication, and lives within an operating system. MCAPI is intended for inter-core communication and can be a part of an operating system or reside on a core bare-metal. Hence, MCAPI is designed to be lighter-weight and less flexible than MPI. MCAPI is also different from threads, as threads assume shared memory. MCAPI is designed to work with multicore processors with either private memories or shared memory. The MCAPI specification defines two inter-core communication types: connectionless communication and connection-oriented FIFO streams communication.

In connectionless communication, two or more cores can exchange messages without setting up a transmission channel prior to communication. On the other hand, in connection-oriented communication, a point-to-point unidirectional channel between two communicating cores must be established in advance. There are two types of FIFO streams supported by MCAPI: packet streams, and scalar (word) streams. A packet in a packet stream can contain a combination of data items. A scalar stream transfer words that belong to the same data type (e.g. integer, float…) throughout the life of the connection.

In the MCAPI terminology, a core is referred to as a *node*. A *port* is a socket-like communication termination point that belongs to a node. A node can have one or more ports, but a port is owned by one node. Ports in a node are assigned unique numerical identifiers. Communication between two nodes must go through ports. An *endpoint* is a logical entity that corresponds to a specific port on a specific node. For example, sending a message from endpoint (0,1) to endpoint (1,2) means that the message was sent from the port with identifier 1, at node 0, to the port with identifier 2 at node 1. The MCAPI specification assumes that the number of the cores used in an application is static. A node creates and removes endpoints by calling the functions *create_endpoint* and *delete_endpoint* respectively. Calling the function *get_endpoint* retrieves a handle to an endpoint by specifying a node and a port. The major functions for connectionless communication are: *msg_send*, *msg_send_i*, *msg_recv*, *msg_recv_i*, and *wait*. The functions *msg_send* and *msg_send_i* are used to send messages between endpoints. The function *msg_send* will block till the message data has been copied from an application buffer to the MCAPI runtime buffers. The non-blocking function *msg_send_i* will return immediately before completing copying the data from the application buffer to the MCAPI runtime buffers. The functions *msg_recv* and *msg_recv_i* are used to retrieve messages from the MCAPI runtime buffers. The function *msg_recv* will block till a message has been retrieved from the

runtime buffers, while *msg_recv_i* will return immediately even if there are no messages in the buffer. The MCAPI specification provides *request variables* and the *wait* function to track the status of a non-blocking call. A non-blocking function takes a request variable as an input, and sets its value to *pending*. Calling the function *wait* with a request variable will block execution till a non-blocking operation (i.e. *msg_send_i* and *msg_recv_i*) has completed. A blocking call is equivalent to a non-blocking call followed by a *wait* call.

There are two rules that govern the order of messages arrivals at a destination endpoint: 1) Messages sent from the same source endpoint are guaranteed to arrive at their destination according to their transmission order, and 2) Messages sent from different source endpoints will arrive at their destination in any order, even if these source endpoints belong to the same node. The second rule combined with the fact that *msg_recv* and *msg_recv_i* calls don't specify the source endpoint, make it possible for *message races* to take place. Two or more messages are said to be *racing* if their order of arrival at a destination (i.e. a core) is non-deterministic [6]. Figure 1 shows a snippet from an MCAPI program in which four cores (C1, C2, C3, and C4) communicate via messages. For brevity, unessential code is omitted. This program may have an assertion failure due to an unexpected order of arrival of messages.

The program in Figure 1 has six possible scenarios for the order of arrival of messages at C2 and C4 which are listed in Table 1. Only one scenario (when M0 arrives before M3 and M2 beats M1 and M4) leads to an assertion failure at C4. Figure 2 depicts the first and second scenarios in Table 1. Unfortunately, the traditional testing approach that repeatedly executes an application is not effective in detecting this message-race induced 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 it. Hence, debugging MCAPI programs that use connectionless communication is expected to be very challenging.

```
1   #define PORT_NUM 1
2
3   void* C1_routine (void *t)
4   {
5     int Msg=1;
6     initialize();
7     My_endpt = create_endpoint (PORT_NUM);
8     C2_endpt = get_endpoint (2,PORT_NUM);
9     C4_endpt = get_endpoint (4,PORT_NUM);
10    msg_send_i(My_endpt,C2_endpt, Msg,Req0);//M0
11    msg_send_i (My_endpt,C4_endpt, Msg, Req1);//M1
12    wait(Req0);
13    wait(Req1);
14    delete_endpoint(My_endpt);
15    finalize();
16  }
17
18  void* C2_routine (void *t)
19  {
20    int X=0,Y=0,Z=0;
21    initialize();
22    My_endpt = create_endpoint (PORT_NUM);
23    C4_endpt = get_endpoint (4,PORT_NUM);
24    msg_recv_i(My_endpt,X,&recv_size,Req2);
25    msg_recv_i(My_endpt,Y,&recv_size,Req3);
26    wait(Req2);
27    wait(Req3);
28    Z=X-Y;
29    msg_send(My_endpt,C4_endpt,Z);//M2
30    delete_endpoint(My_endpt);
31    finalize();
32  }
33  void* C3_routine (void *t)
34  {
35    int Msg=10;
36    initialize();
37    My_endpt = create_endpoint (PORT_NUM);
38    C2_endpt = get_endpoint (2,PORT_NUM);
39    C4_endpt = get_endpoint (4,PORT_NUM);
40    msg_send(My_endpt,C2_endpt, Msg);//M3
41    msg_send(My_endpt,C4_endpt, Msg);//M4
42    delete_endpoint(My_endpt);
43    finalize();
44  }
45
46  void* C4_routine (void *t)
47  {
48    int U=0,W=0, O=0;
49    initialize();
50    My_endpt = create_endpoint (PORT_NUM);
51    msg_recv(My_endpt,U);
52    assert(U>0)
53    msg_recv(My_endpt,W);
54    msg_recv(My_endpt,O);
55    delete_endpoint(My_endpt);
56    finalize();
57  }
```

Figure 1. A snippet of an MCAPI program

Table 1. Six orders of arrival of messages

First Race	M0 wins			M3 wins		
Second Race	M1 wins	M2 wins	M4 wins	M1 wins	M2 wins	M4 wins
Z	Z=-9	Z=-9	Z=-9	Z=9	Z=9	Z=9
U	U=1	**U=-9**	U=10	U=1	U=9	U10

In this paper we present a tool that *symbolically* explores all possible orders of messages arrival in an MCAPI program that uses connectionless communication. Our tool starts with instrumenting an MCAPI program, so that an execution will produce a trace that contains a set of *events* for every MCAPI node. These events correspond to the program statements that have actually been executed. Our tool explores all *feasible* orderings of these events. If there is a particular order of events, and hence a particular order of messages arrival that leads to an error state, this order will be presented to the tool user. Our tool is able to predict the possibility of reaching an error state from an execution trace that doesn't exhibit this error state. The rest of this paper is organized as follows: Section 2 defines formally an MCAPI program and its execution trace. In section 3 we present our SMT-based symbolic encoding of a MCAPI program trace. We review related work in section 4, and conclude in section 5.

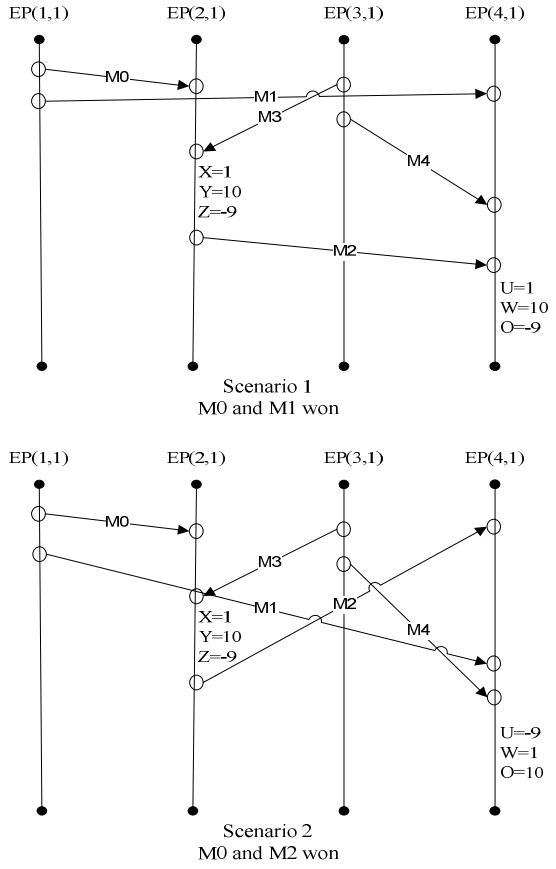

Figure 2. Two possible scenarios for messages arrival

2. An MCAPI program and its execution trace

An MCAPI program runs on a finite set of nodes and for every node there is a finite set of local variables and an ordered list of statements. Let $\mathcal{N} = \{N_1, \dots, N_{|\mathcal{N}|}\}$ be the set of nodes, $\mathcal{L}_i = \{L_{i,1}, \dots, L_{i,|\mathcal{L}_i|}\}$ be the set of local variables in node N_i, and $\mathcal{M}_i = \{\mathcal{M}_{i,1}, \dots, \mathcal{M}_{i,|\mathcal{M}_i|}\}$ be the set of statements in node N_i. An endpoint at port x on node N_i is denoted as $EP_{i,x}$. Let $\mathcal{EP}_i$ be the set of endpoints created in node N_i.

The trace of an MCAPI program running on n nodes will have n sub-traces; a sub-trace for each node. Let $\mathcal{R} = \{\mathcal{T}_1, \dots, \mathcal{T}_n\}$ be the trace of an MCAPI program with n nodes. $\mathcal{T}_i$ is the sub-trace produced by node N_i and it consists of a sequence of events $\mathcal{T}_i = T_{i,1} \dots T_{i,|\mathcal{T}_i|}$. An event $T_{i,x} \in \mathcal{T}_i$ is a tuple $< i, x, Guard, Action >$, such that i is a node identifier, x is the order of the event appearance in $\mathcal{T}_i$, $Guard$ is a condition that must be true for this event to take place, and $Action$ is an atomic computation that corresponds to an executed statement in the MCAPI program. $Guard$ is the conjunction of all conditions in the program path leading to the statement that produced the event. For example, if the statement that produces $T_{i,x}$ is in the then-part of an *if* statement, which in turn is inside a *while* loop, then $Guard$ will be the conjunction of the *if* statement condition and the *while* loop condition. $Action$ can be any of the following:

- $Assign(v, exp)$ is an action that corresponds to an assignment statement that assigns the valuation of exp to v. $v \in \mathcal{L}_i$ is a variable. exp is an expression over $\mathcal{L}_i$.
- $Send(src, dest, exp)$ is an action that corresponds to a blocking send statement that sends a message from src to $dest$, which contains the valuation of exp. $src \in \mathcal{EP}_i$ and $dest \in \mathcal{EP}_j$ are the source and destination endpoints. exp is an expression over $\mathcal{L}_i$. Similarly, $Send\_i(src, dest, exp, req)$ is an action that corresponds to a non-blocking send statement where $req \in \mathcal{L}_i$ is a request variable.
- $Recv(recv, v)$ is an action that corresponds to a blocking receive statement that receives a message at the receiving endpoint $recv \in \mathcal{EP}_i$. The message contents are assigned to variable $v \in \mathcal{L}_i$. Similarly, $Recv\_i(recv, v, req)$ corresponds to a non-blocking receive statement where $req \in \mathcal{L}_i$ is a request variable.
- $Wait(req)$ is an action that corresponds to a wait statement that waits for the completion of a non-blocking action whose status is tracked with request variable $req \in \mathcal{L}_i$.
- $Assert(exp)$ is an action that corresponds to an assert statement with the boolean expression exp. exp must be true when this event is executed; otherwise, an error state has been reached.

Table 2 shows an execution trace which corresponds to the program in Figure 1. Note that the $Guard$ condition is omitted as it is $true$ for all events.

Table 2. The trace of a MCAPI program in Figure 1

$\mathcal{T}_1$		$\mathcal{T}_3$	
$T_{1,1}$	Assign(Msg,1)	$T_{3,1}$	Assign(Msg,10)
$T_{1,2}$	Send_i(EP1,EP2,Msg,r0)	$T_{3,2}$	Send(EP3,EP2,Msg)
$T_{1,3}$	Send_i(EP1,EP4,Msg,r1)	$T_{3,3}$	Send(EP3,EP4,Msg)
$T_{1,4}$	Wait(r0)	$\mathcal{T}_4$	
$T_{1,5}$	Wait(r1)	$T_{4,1}$	Assign(U,0)
$\mathcal{T}_2$		$T_{4,2}$	Assign(W,0)
$T_{2,1}$	Assign(X,0)	$T_{4,3}$	Assign(O,0)
$T_{2,2}$	Assign(Y,0)	$T_{4,4}$	Recv(EP4,U)
$T_{2,3}$	Assign(Z,0)	$T_{4,5}$	Assert(U>0)
$T_{2,4}$	Recv_i(EP2,X,r2)	$T_{4,6}$	Recv(EP4,W)
$T_{2,5}$	Recv_i(EP2,Y,r3)	$T_{4,7}$	Recv(EP4,O)
$T_{2,6}$	Wait(r2)		
$T_{2,7}$	Wait(r3)		
$T_{2,8}$	Assign(Z,X-Y)		
$T_{2,9}$	Send(EP2, EP4,Z)		

Table 3. The symbolic variables of the trace in Table 2

$\mathcal{T}_1$	$\mathcal{T}_2$	$\mathcal{T}_3$	$\mathcal{T}_4$
$\mathcal{O}_{1,1}$	$\mathcal{O}_{2,1}$	$\mathcal{O}_{3,1}$	$\mathcal{O}_{4,1}$
$\mathcal{O}_{1,2}$	$\mathcal{O}_{2,2}$	$\mathcal{O}_{3,2}$	$\mathcal{O}_{4,2}$
$\mathcal{O}_{1,3}$	$\mathcal{O}_{2,3}$	$\mathcal{O}_{3,3}$	$\mathcal{O}_{4,3}$
$\mathcal{O}_{1,4}$	$\mathcal{O}_{2,4}$	T_3Msg_1	$\mathcal{O}_{4,4}$
$\mathcal{O}_{1,5}$	$\mathcal{O}_{2,5}$		$\mathcal{O}_{4,5}$
T_1Msg_1	$\mathcal{O}_{2,6}$		$\mathcal{O}_{4,6}$
	$\mathcal{O}_{2,7}$		$\mathcal{O}_{4,7}$
	$\mathcal{O}_{2,8}$		T_4U_1
	$\mathcal{O}_{2,9}$		T_4W_1
	T_2X_1		T_4O_1
	T_2Y_1		T_4U_2
	T_2Z_1		T_4W_2
	T_2X_2		T_4O_2
	T_2Y_2		
	T_2Z_2		

3. SMT-based Symbolic Encoding

Given a trace $\mathcal{R}$, we create a quantifier-free first-order logic formula $\mathcal{F}_{\mathcal{R}}$ that is satisfiable iff there exists a *feasible permutation* $\mathcal{P}_R$ of the events in $\mathcal{R}$ that leads to an error state (e.g. an assertion failure). A feasible permutation is a strict total order of all the events in $\mathcal{R}$, such that this order can occur in a real execution of the original program. The $\mathcal{F}_{\mathcal{R}}$ formula consists of symbolic variables and Satisfiability Modulo Theory (SMT) constraints.

There are two types of symbolic variables in $\mathcal{F}_{\mathcal{R}}$: 1) For every event $T \in \mathcal{R}$, there is a symbolic variable $\mathcal{O}_T$ that reflects the *order* of carrying out T in $\mathcal{P}_R$. 2) For every action that assigns a new value to a local variable $L \in \bigcup_{i=1}^{n} \mathcal{L}_i$, we create a new symbolic variable for L. The values of these symbolic variables record the history of the values of L. This is similar to the SSA form [4]. While the SSA form requires ϕ-functions to handle the effect of branches, we needn't have ϕ-functions because in a trace all branching decisions have already been made. We add two dummy variables $\mathcal{O}_{First}$ and $\mathcal{O}_{Last}$, such that $\mathcal{O}_{First}$ is the first event in $\mathcal{P}_R$ and $\mathcal{O}_{Last}$ is the last event in $\mathcal{P}_R$. The values assigned to these symbolic variables are governed by SMT constraints that are crafted to ensure that $\mathcal{P}_R$ is a feasible permutation. Table 3 shows the symbolic variables that are needed for encoding the trace in Table 2. A symbolic variable $\mathcal{O}_{i,j}$ represents the order of the event $T_{i,j}$. A symbolic variable T_iV_j corresponds to the value of the variable V at sub-trace T_i after being assigned a value for the j[th] time.

The $\mathcal{F}_{\mathcal{R}}$ formula is the conjunction of four sub-formulas:

$$\mathcal{F}_{\mathcal{R}} := \mathcal{F}_{order} \wedge \mathcal{F}_{asgn} \wedge \mathcal{F}_{recv} \wedge \neg\mathcal{F}_{prp} \qquad (1)$$

$\mathcal{F}_{order}$ ensures that in $\mathcal{P}_R$, no two events are assigned the same ordering and that every two events $T_{i,x}$ and $T_{i,y}$, such that $x < y$ (i.e. event $T_{i,x}$ appears in the trace before event $T_{i,y}$) will be assigned orderings $\mathcal{O}_{i,x}$ and $\mathcal{O}_{i,y}$, such that $\mathcal{O}_{i,x} < \mathcal{O}_{i,y}$. $\mathcal{F}_{order}$ is constructed using the algorithm Construct_FOrder.

Algorithm 1 Construct_FOrder($\mathcal{R}$)

```
1   F_order := true
2   for i=1 to n
3     F_order := F_order ∧ (O_First < O_{T_{i,1}})
4     for j=1 to |T_i|
5       if (j<|T_i|) then F_order := F_order ∧ (O_{T_{i,j}} < O_{T_{i,j+1}})
6       for k=i + 1 to n
7         for l=1 to |T_k|
8         F_order := F_order ∧ (O_{T_{i,j}} ≠ O_{T_{k,l}})
9         end-for
10      end-for
11    end-for
12    F_order := F_order ∧ (O_Last > O_{T_{i,j}})
13  end-for
```

$\mathcal{F}_{asgn}$ encodes events with assignment actions. $\mathcal{F}_{asgn}$ is initially set to true. For every event $T_{i,x}$ whose action is $Assign(v, exp)$:

$$\mathcal{F}_{asgn} := \mathcal{F}_{asgn} \wedge (S(v) = S(exp) \wedge S(Guard)) \qquad (2)$$

Where $S(v)$, $S(exp)$ and $S(Guard)$ replace the program variables with the corresponding symbolic ones.

$\mathcal{F}_{recv}$ encodes the events with an action that is either a blocking receive, or a wait of a non-blocking receive. To facilitate describing the $\mathcal{F}_{recv}$ constraint, we use the following notations:

For every event $T_{i,x}$ whose $Action$ is either $Send(src, dest, exp)$ or $Send\_i(src, dest, exp, req)$:

- $DestEP(T_{i,x}) = dest$
- $Exp(T_{i,x}) = exp$
- $SOrder(T_{i,x})$ is the order of $T_{i,x}$ with respect to other events in $\mathcal{T}_i$ whose actions are either $Send(src, dest, exp)$ or $Send\_i(src, dest, exp, req)$ and have the same destination endpoint as $T_{i,x}$.

For every event $T_{i,x}$ whose $Action$ is either $Recv(recv, v)$ or $Wait(req)$ such that $Wait(req)$ is associated with a non-blocking receive action $Recv\_i(recv, v, req)$:

- $RecvEP(T_{i,x}) = recv$
- $Var(T_{i,x}) = v$
- $ROrder(T_{i,x})$ is the order of $T_{i,x}$ with respect to other events in $\mathcal{T}_i$ whose actions are either $Recv(recv, v)$ or $Wait(req)$ such that $Wait(req)$ is associated with a non-blocking receive action $Recv\_i(recv, v, req)$ and have the same receiving endpoint as $T_{i,x}$
- $\mathcal{S}_{i,x}$ is the set of events whose actions are either $Send(src, dest, exp)$ or $Send\_i(src, dest, exp, req)$ and can potentially match with the receive action of $T_{i,x}$. $\mathcal{S}_{i,x}$ is defined as: $\mathcal{S}_{i,x} = \{T_{j,y} | \, Dest(T_{j,y}) = Recv(T_{i,x}) \wedge ROrder(T_{i,x}) \geq SOrder(T_{j,y})\}$. We call $\mathcal{S}_{i,x}$, the *set of potential sender events* of $T_{i,x}$.
- $\mathcal{P}_{i,x}$ is the set of events whose actions are 1) either $Recv(recv, v)$ or $Wait(req)$ such that $Wait(req)$ is associated with a non-blocking receive $Recv\_i(recv, v, req)$ 2) precede $T_{i,x}$ in $\mathcal{T}_i$, and 3) have the same receiving endpoint as $T_{i,x}$. $\mathcal{P}_{i,x}$ is defined as $\mathcal{P}_{i,x} = \{T_{i,y} | \, ROrder(T_{i,y}) < ROrder(T_{i,x})\}$. We call $\mathcal{P}_{i,x}$, the *set of related preceding receiving events* of $T_{i,x}$.

$\mathcal{F}_{recv}$ is initially set to true. For an event $T_{i,x}$ whose action is either $Recv(recv, v)$ or $Wait(req)$ such that $Wait(req)$ is associated with a non-blocking receive $Recv\_i(recv, v, req)$:

$$\mathcal{F}_{recv} := \mathcal{F}_{recv} \wedge \vee_{s \in \mathcal{S}_{i,x}} (S(Var(T_{i,x})) = S(Exp(s)) \wedge S(Guard) \wedge CON^{s}_{T_{i,x}} \wedge \wedge_{p \in \mathcal{P}_{i,x}} \neg CON^{s}_{p}) \quad (3)$$

$$CON^{s}_{r} = (O_s < O_r) \wedge \wedge_{n \in \mathcal{S}_r \wedge n \neq s} ((O_n < O_s) \vee (O_r < O_n)) \quad (4)$$

CON^{s}_{r} encodes the *conditions* needed for matching an event s with a send action to an event r with a receive action. These conditions are 1) s must precede r ($O_s < O_r$), and 2) for every event n, such that $n \in \mathcal{S}_r \wedge n \neq s$, then either n is before s or r is before n ($\wedge_{n \in \mathcal{S}_r \wedge n \neq s} ((O_n < O_s) \vee (O_r < O_n))$).

Formula 3 states that the receive action of $T_{i,x}$ will be matched with the event s, when the conditions for this matching are satisfied ($CON^{s}_{T_{i,x}}$), and when all the conditions needed for matching s with any event in $\mathcal{P}_{i,x}$, are not satisfiable ($\wedge_{p \in \mathcal{P}_{i,x}} \neg CON^{s}_{p}$).

For example, the part of $\mathcal{F}_{recv}$ that corresponds to the event $T_{4,4}$ is the disjunction of the formulas 5, 6 and 7. Formulas 5, 6, and 7 match the receive action at event $T_{4,4}$ with the send action at events $T_{1,3}$, $T_{2,9}$ and $T_{3,3}$ respectively and encodes the necessary conditions. Only one formula of these three formulas will be satisfied.

$$(T_4U_2{=}T_1Msg_1 \wedge (\mathcal{O}_{1,3} < \mathcal{O}_{4,4} \wedge (((\mathcal{O}_{3,3}{<}\mathcal{O}_{1,3}) \vee (\mathcal{O}_{4,4}{<}\mathcal{O}_{3,3})) \wedge ((\mathcal{O}_{2,9}{<}\mathcal{O}_{1,3}) \vee (\mathcal{O}_{4,4}{<}\mathcal{O}_{2,9})))) \quad (5)$$

$$(T_4U_2{=}T_2Z_1 \wedge (\mathcal{O}_{2,9} < \mathcal{O}_{4,4} \wedge (((\mathcal{O}_{3,3}{<}\mathcal{O}_{2,9}) \vee (\mathcal{O}_{4,4}{<}\mathcal{O}_{3,3})) \wedge (\mathcal{O}_{1,3}{<}\mathcal{O}_{2,9}) \vee (\mathcal{O}_{4,4}{<}\mathcal{O}_{2,9})))) \quad (6)$$

$$(T_4U_2{=}T_3Msg_1 \wedge (\mathcal{O}_{3,3} < \mathcal{O}_{4,4} \wedge (((\mathcal{O}_{2,9}{<}\mathcal{O}_{3,3}) \vee (\mathcal{O}_{4,4}{<}\mathcal{O}_{2,9})) \wedge (\mathcal{O}_{1,3}{<}\mathcal{O}_{3,3}) \vee (\mathcal{O}_{4,4}{<}\mathcal{O}_{3,3})))) \quad (7)$$

Intuitively, $\mathcal{F}_{recv}$ matches an event $s \in \mathcal{S}_r$ with *one* event r, provided that s has not been matched with any event $p \in \mathcal{P}_r$, and s can occur before r. The effect of a matching is assigning the valuation of the expression sent by s to the variable of r.

$\mathcal{F}_{prp}$ is initially set to true. For every event $T_{i,x}$ whose action as $Assert(exp)$:

$$\mathcal{F}_{prp} := \mathcal{F}_{prp} \wedge (S(exp) \wedge S(Guard)) \quad (8)$$

After the formula $\mathcal{F}_{\mathcal{R}}$ has been constructed, it is passed to an SMT solver such as Yices [2] or Z3 [1]. If $\mathcal{F}_{\mathcal{R}}$ is satisfiable, then the SMT solver will produce a solution that assigns a value for every $\mathcal{O}_T$ variable that indicates the order of carrying out the event T in the permutation $\mathcal{P}_R$. Table 4 shows the solution produced by Yices for the formula that corresponds to the program in Figure 1. For example, any of the events ($T_{1,3}$, $T_{3,3}$, and $T_{2,9}$) whose actions are send actions can match with the event $T_{4,4}$ whose action is a receive action. The fact that $\mathcal{O}_{2,9}{<}\mathcal{O}_{4,4}$, $\mathcal{O}_{1,3}{>}\mathcal{O}_{4,4}$ and $\mathcal{O}_{3,3}{>}\mathcal{O}_{4,4}$ indicates that $T_{2,9}$ is the event that will be matched with $T_{4,4}$. Similarly, all send/receive matchings can be extracted from the solution and presented to the user as the trace that led to the error state.

Table 4. The solution of the $\mathcal{F}_{\mathcal{R}}$ formula

Variable	Value	Variable	Value
$\mathcal{O}_{1,1}$	1	T_2Y_2	10
$\mathcal{O}_{1,2}$	7	T_2Z_2	-9
$\mathcal{O}_{1,3}$	19	$\mathcal{O}_{3,1}$	11
$\mathcal{O}_{1,4}$	21	$\mathcal{O}_{3,2}$	13
$\mathcal{O}_{1,5}$	22	$\mathcal{O}_{3,3}$	23
T_1Msg_1	1	T_3Msg_1	10
$\mathcal{O}_{2,1}$	2	$\mathcal{O}_{4,1}$	9
$\mathcal{O}_{2,2}$	3	$\mathcal{O}_{4,2}$	10
$\mathcal{O}_{2,3}$	4	$\mathcal{O}_{4,3}$	12
$\mathcal{O}_{2,4}$	5	$\mathcal{O}_{4,4}$	17
$\mathcal{O}_{2,5}$	6	$\mathcal{O}_{4,5}$	18
$\mathcal{O}_{2,6}$	8	$\mathcal{O}_{4,6}$	20
$\mathcal{O}_{2,7}$	14	$\mathcal{O}_{4,7}$	24
$\mathcal{O}_{2,8}$	15	T_4U_1	0
$\mathcal{O}_{2,9}$	16	T_4W_1	0
T_2X_1	0	T_4O_1	0
T_2Y_1	0	T_4U_2	-9
T_2Z_1	0	T_4W_2	1
T_2X_2	1	T_4O_2	10

4. Related work

In [7], S. Sharma et al. present MCC, the first dynamic verifier for MCAPI applications. MCC explores all possible orders of messages arrival by repeatedly executing the actual program. MCC 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. MCC reduces the number of explored orders of messages arrival via using DPOR [3] techniques. In [11], 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. Their algorithm has been applied to detect concurrency errors in shared memory multithreaded C programs.

5. Conclusion

We have presented a debugging tool for detecting assertion failures induced by message races in multi-core applications. Our tool aims at reducing the developer efforts to locate the source of an assertion failure. It uses an efficient SMT formula that is decidable if and only if there is a particular order of messages arrival that leads to an error state. Our tool reports the possibility of an assertion failure *and* the sequence of events that will lead to it. As there are no publicly available MCAPI benchmarks, we performed experiments on MCAPI applications developed by ourselves. For instance, the full code of the application in Figure 1 was found to have an assertion failure in 0.01 seconds using Yices as the SMT solver. We plan to extend our tool to support other MCAPI constructs.

6. References

[1] de Moura, L., Bjørner, N., Z3: An efficient SMT solver, Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008).

[2] Dutertre, B and de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T), 18th Computer-Aided Verification conference, volume 4144 of LNCS, pages 81–94. Springer-Verlag, 2006.

[3] Flanagan, C. and Godefroid, P., Dynamic partial-order reduction for model checking software, the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '05, pages 110–121, New York,NY, USA, 2005. ACM Press.

[4] Lee, J., Padua, D., Midkiff, S., Basic compiler algorithms for parallel programs, Principles and Practice of Parallel Programming. (1999) 1–12

[5] MPI: A Message-Passing Interface Standard, http://www.mpi-forum.org/docs/mpi-2.2/index.htm

[6] Netzer, R.H.B., Brennan, T.W., Damodaran-Kamal, S.K., Debugging Race Conditions in Message-Passing Programs. ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Philadelphia, SPDT'96 ,PA, USA, pp. 31-40, 1996

[7] Sharma, S., Gopalakrishnan, G., Mercer, E and Holt, J, MCC: A runtime verification tool for MCAPI applications, the 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA.

[8] The Multicore Association, http://www.multicore-association.org

[9] The Multicore Association Communications API, http://www.multicore-association.org/workgroup/mcapi.php

[10] The Multicore Association Resource Management API, http://www.multicore-association.org/workgroup/mrapi.php

[11] Wang C., Kundu S., Ganai M., and Gupta A., Symbolic Predictive Analysis for Concurrent Programs, the 2nd World Congress on Formal Methods, Vol. 5850 of LNCS, Pages: 256-272, Springer-Verlag, 2009.