Proving MCAPI Executions are Correct

Applying SMT Technology to Message Passing

Abstract

Asynchronous message passing is an important paradigm in writing applications for embedded heterogeneous multicore systems. The Multicore Association (MCA), an industry consortium promoting multicore technology, is working to standardize message passing into a single API, MCAPI, for bare metal implementation and portability across platforms. Correctness in such an API is difficult to reason about manually, and testing against reference solutions is equally difficult as reference solutions implement an unknown set of allowed behaviors, and programmers have no way to directly control API internals to expose or reproduce errors. This paper encodes an MCAPI execution as an SMT problem, which if satisfiable, yields a feasible execution schedule on the same trace, such that it resolves non-determinism in the MCAPI runtime in the way that it now fails user provided assertions. Such an encoding is useful for test, debug, and verification of MCAPI program execution. The novelty in the encoding is the use of match pairs (potential send and receive couplings). Match-pair encoding for MCAPI executions, when compared to other encoding strategies, is simpler to reason about, results in significantly fewer terms in the SMT problem, and captures feasible behaviors that are ignored in previously published techniques. Further, to our knowledge, this is the first SMT encoding that is able to run in *infinite-buffer* semantics, meaning the runtime has unlimited internal buffering as opposed to no internal buffering. Our results demonstrate that our SMT encoding uses 70% fewer clauses of that of the existing technique. In addition, our encoding runs on average eight times faster and uses two times less memory than the existing technique on the benchmarks in our experiment.

Keywords Abstraction, refinement, SMT, message passing

1. Introduction

Embedded devices fill all sorts of crucial roles in our lives. They exist as medical devices, as network infrastructure, and they control our automobiles. Embedded devices continue to become more powerful as computing hardware becomes smaller and more modular. It is now becoming commonplace to find multiple processing units inside a single device. The Multicore Association (MCA) [2] is an industry group that has formed to define specifications for low-level communication, resource management, and task management for multicore devices.

One specification that the MCA has released is the Multicore Association Communications API (MCAPI) [3]. The specification defines basic data type, data structures, and functions that can be used to perform simple message passing operations between different computing entities within a device. Messages can be passed across persistent channels that force an ordering of the messages, or they can be passed to specific *endpoints* within the system. The specification places few ordering constraints on messages passed from one endpoint to another. This freedom introduces into the system possibilities that two or more messages are racing if the order of their arrival at the destinations are non-deterministic [17]. Without a way to explore this non-determinism in the MCAPI runtime, it is not possible to test and debug the program executions.

Sharma et al. created a method of using concrete execution to verify MCAPI programs, but it was later discovered that this method does not completely explore the entire execution space of certain kinds of MCAPI programs [18]. The method provides match pairs – couplings for potential sends and receives that we use in our work. Instead of exploring all relevant interleavings of a program in the concrete execution in [18], Wang et al. provided a symbolic algorithm that verifies each partitioned concurrent trace program (CTP) with shared memory semantics using a satisfiability solver [20]. Elwakil et al. also defined a method of representing MCAPI program executions as Satisfiability Modulo Theory (SMT) problems building on the method of [20] adapting it to message passing, but this method does not capture all allowed MCAPI program executions [9]. Further, this method assumes that the user is able to provide the exact set of match pairs. Such an assumption is not reasonable for large complex program traces. In this paper, however, we provide an algorithm that generates an over-approximated set of match pairs. In order to prevent the problems discussed above, this paper discusses a new method that provides both efficiency and correctness.

There are two ways to implement the MCAPI semantics for matching sends to receives including an *infinite-buffer* semantics (the message is copied into a runtime provided buffer on the API call) or a *zero-buffer* semantics (the message is copied into a endpoint provided buffer) [19]. An *infinite-buffer* semantics provides more non-deterministic behaviors because a particular endpoint can receive more messages under the circumstance of *infinite-buffer* semantics while the *zero-buffer* semantics can only allow one message to be received by a specific endpoint at a time. We present an encoding for MCAPI program executions that works for both *zero-buffer* and *infinite-buffer* semantics that requires fewer terms than [9] but includes all possible program executions unlike [9]. In other words, the solution for our encoding provides efficiency. Our encoding uses match pairs from [18] that captures the non-determinism of program executions. Most importantly, this method correctly captures all possible execution traces allowed by the MCAPI specification in a trace of a concurrent program execution as an SMT problem enabling the exploration of inherent

non-determinism that may exist in any MCAPI runtime implementation. Our main contributions in this paper include:

1. A correct and efficient SMT encoding of an MCAPI program execution that detects all program errors if the user provided match pairs are precise or over-approximated.
2. An $O(N^2)$ algorithm to generate an over-approximation of possible match-pairs, where N is the size of the execution trace as lines of code.

The rest of the paper is organized as follows: Section 2 presents an MCAPI program execution in which two program traces exist due to non-determinism in the MCAPI runtime. Section 3 defines an SMT encoding of an MCAPI program execution that captures non-determinism in message order by defining the set of possible match pairs between sends and receives on a common endpoint. It is proved that an over-approximated match set in our SMT encoding of an MCAPI program execution can reflect the actual execution traces as the true set does. Section 4 solves the outstanding problem in other encodings of generating feasible match pairs to use in the encoding. It presents a $O(N^2)$ algorithm that over-approximates the precise match set, where N is the size of the execution trace in lines of code. Section 5 presents the experimental results that show our encoding to be correct and efficient, as other existing encodings both omit critical program behaviors and require more SMT clauses. Section 6 discusses related work, and Section 7 presents our conclusions and future work.

2. Example

It is a challenge to explain intended behavior in simple scenarios consisting of a handful of calls when dealing with concurrency. Consider the MCAPI program execution in Figure 1 that includes three tasks that use send (`mcapi_msg_send_i`) and receive (`mcapi_msg_recv_i`) calls to communicate with each other. Line numbers appear in the first column for each task, and the declarations of the local variables are omitted for space. Picking up the scenario just after the endpoints are defined, lines `02` and `05` of task 0 receive two messages on the endpoint *e0* in variables A and B which are converted to integer values and stored in variables a and b on lines `04` and `07`; task 1 receives one message on endpoint *e1* in variable *C* on line `03` and then sends the message *"1"* on line `05` to *e0*; and finally, task 2 sends messages *"4"* and *"Go"* on lines `04` and `06` to endpoints *e0* and *e1* respectively. Task 0 has additional code (lines `08` - `09`) to assert properties of the values in a and b. The `mcapi_wait` calls block until the associated send or receive buffer is able to be used. Given the scenario, we might ask the question: *"What are the possible values of* `a` *and* `b` *after the scenario completes?"*

The intuitive trace is presented in the first four columns of Figure 2. Note that the first column contains the line number of each command shown in the trace. The second column contains the task number in Figure 1. The third column presents the commands identified by the source line numbers shown in Figure 1. Also, we define a shorthand for each command of send (denoted as `S`), receive (denoted as `R`), or wait (denoted as `W`) in the fourth column for presenting future examples. For each command $\mathtt{O}_{\mathtt{i},\mathtt{j}}(\mathtt{k}, \&\mathtt{h})$, $\mathtt{O} \in \{\mathtt{S}, \mathtt{R}\}$ or $\mathtt{W}(\&\mathtt{h})$, `i` represents the task number, `j` represents the source line number, `k` represents the destination task number, and `h` represents the command handler. Note that a specific destination task number can be assigned to each receive for a provided program trace, which implies the matched send in the program runtime. We use the "*" notation when a receive has yet to be matched to a specific send. From the trace, variable a should contain 4 and variable b should contain 1 since task 2 must first send message *"4"* to *e0* before it can send message *"Go"* to *e1*; consequently, task 1 is then able to send message *"1"* to *e0*. The assume notation asserts the control flow taken by the program execution which in this example, asserts the true branch of the condition on line `08` of task 0. At the end of execution the assertion on line `09` of task 0 holds and no error is found. Such intuition is a valid program execution.

For our example in Figure 1, if we use *zero-buffer* semantics as buffer setting, the send call on line `05` of task 1 cannot match with the receive call on line `02` of task 0. This is because the send call on line `04` of task 2 must be completed before the send call on line `06` of task 2 can match with the receive call on line `03` of task 1. When using *infinite-buffer*, however, we can have another scenario shown in the fifth and sixth column of Figure 2 written in the shorthand notation. The variable *a* contains 1 instead of 4, since the message *"1"* is sent to *e0* after sending the message *"Go"* to *e1* as it is possible for the send on line `04` of task 2 buffered in transit. The specification indicates that the wait on line `05` of task 2 returns once the buffer is available. That only means the message is somewhere in the MCAPI runtime under *infinite-buffer* semantics; it does not mean the message is delivered. As such, it is possible for the message to be buffered in transit allowing the send from task 1 on line `05` to arrive at *e0* first and be received in variable *"a"*. Such a scenario is a program execution that results in an assertion failure at line `09` in Figure 1.

From the discussion above, it is important to consider non-determinism in the MCAPI runtime when testing or debugging an MCAPI program execution. Note that we use *infinite-buffer* for the program execution in the rest of our discussion but describe how the encoding changes for *zero-buffer* semantics.. The next section presents an encoding algorithm that takes an MCAPI program execution with a set of possible send-receive match pairs and generates an SMT problem that if satisfied proves that non-determinism can be resolved in a way that violates a user provided assertion and if unsatisfiable proves the trace correct (meaning the user assertions hold on the given execution). The encoding can be solved by an SMT solver such as Yices [7] and Z3 [16], and the non-deterministic behavior of the program execution can be resolved.

3. SMT Model

The novelty of the SMT encoding in this paper is its use of match pairs rather than the state-based or order-based encoding of prior work [8, 9]. The algorithm to create the encoding takes as input a set of possible match pairs and a trace through an MCAPI program with the appropriate assumes and asserts as shown in Figure 2. A match pair is the coupling of a receive to a particular send. Figure 3(a) is the set of possible match pairs for the program in Figure 1 using our shorthand notation defined in Figure 2. The set admits, for example, that $\mathtt{R}_{0,2}$ can be matched with either $\mathtt{S}_{1,5}$ or $\mathtt{S}_{2,4}$. The SMT encoding in this paper asks the SMT solver to resolve the match pairs for the system in such a way that the final values of program variables meet the assumption on control flow but violate some assertion.

Before presenting our SMT encoding for a specific example, we need to explain auxiliary data structures and support functions in detail. We give the definitions in the Yices [7] input language.

```
(define HB :: (→ a::int b::int
  (subtype (c::bool) (ite (< a b) (= c true) (= c false)))))
```
(1)

The `HB` function in equation (1) creates a happens-before relationship between two events. The notation "ite" refers to a condition statement such that if the condition is satisfied, then the first subclause is true; otherwise, the second subclause is true. The function takes two events a and b as parameters, such that $a, b \in \mathbb{N}$,

Task 0	Task 1	Task 2
00 initialize(NODE_0,&v,&s); 01 e0 = create_endpoint(PORT_0,&s); 02 msg_recv_i(e0,A,sizeof(A),&h1,&s); 03 wait(&h1,&size,&s,MCAPI_INF); 04 a = atoi(A); 05 msg_recv_i(e0,B,sizeof(B),&h2,&s); 06 wait(&h2,&size,&s,MCAPI_INF); 07 b = atoi(B); 08 if(b > 0); 09 assert(a == 4); 10 finalize(&s);	00 initialize(NODE_1,&v,&s); 01 e1 = create_endpoint(PORT_1,&s); 02 e0 = get_endpoint(NODE_0,PORT_0,&s); 03 msg_recv_i(e1,C,sizeof(C),&h3,&s); 04 wait(&h3,&size,&s,MCAPI_INF); 05 msg_send_i(e1,e0,"1",2,N,&h4,&s); 06 wait(&h4,&size,&s,MCAPI_INF); 07 finalize(&s);	00 initialize(NODE_2,&v,&s); 01 t2 = create_endpoint(PORT_2,&s); 02 t0 = get_endpoint(NODE_0,PORT_0,&s); 03 t1 = get_endpoint(NODE_1,PORT_1,&s); 04 msg_send_i(e2,e0,"4",2,N,&h5,&s); 05 wait(&h5,&size,&s,MCAPI_INF); 06 msg_send_i(e2,e1,"Go",3,N,&h6,&s); 07 wait(&h6,&size,&s,MCAPI_INF); 08 finalize(&s);

Figure 1. An MCAPI concurrent program execution

	Trace 1			Trace 2	
Line	Task	Command	Shorthand	Task	Command
0	2	04 msg_send_i(e2, e0, "4", 2, N, &h5, &s);	$S_{2,4}(0, \&h5)$	2	04 $S_{2,4}(0, \&h5)$
1	2	05 wait(&h5, &size, &s, MCAPI_INF);	W(&h5)	2	05 W(&h5)
2	0	02 msg_recv_i(e0, A, sizeof(A), &h1, &s);	$R_{0,2}(2, \&h1)$	2	06 $S_{2,6}(1, \&h6)$
3	0	03 wait(&h1, &size, &s, MCAPI_INF);	W(&h1)	2	07 W(&h6)
4	2	06 msg_send_i(e2, e1, "Go", 3, N, &h6, &s);	$S_{2,6}(1, \&h6)$	1	03 $R_{1,3}(2, \&h3)$
5	2	07 wait(&h6, &size, &s, MCAPI_INF);	W(&h6)	1	04 W(&h3)
6	0	04 a = atoi(A);		1	05 $S_{1,5}(0, \&h4)$
7	1	03 msg_recv_i(e1, C, sizeof(C), &h3, &s);	$R_{1,3}(2, \&h3)$	1	06 W(&h4)
8	1	04 wait(&h3, &size, &s, MCAPI_INF);	W(&h3)	0	02 $R_{0,2}(1, \&h1)$
9	1	05 msg_send_i(e1, e0, " 1", 2, N, &h4, &s);	$S_{1,5}(0, \&h4)$	0	03 W(&h1)
10	1	06 wait(&h4, &size, &s, MCAPI_INF);	W(&h4)	0	04 a = atoi(A);
11	0	05 msg_recv_i(e0, B, sizeof(B), &h2, &s);	$R_{0,5}(1, \&h2)$	0	05 $R_{0,5}(2, \&h2)$
12	0	06 wait(&h2, &size, &s, MCAPI_INF);	W(&h2)	0	06 W(&h2)
13	0	07 b = atoi(B);		0	07 b = atoi(B);
14	0	08 assume(b > 0);		0	08 assume(b > 0);
15	0	09 assert(a == 4);		0	09 assert(a == 4);

Figure 2. Two execution traces of the MCAPI program execution in Figure 1

where $\mathbb{N}$ is the set of natural numbers. In our SMT encoding, every location in an execution trace is assigned an event, and the HB function orders those locations (i.e., program order, etc.). This function creates a constraint such that the first event must be less than the second event, indicating that the first event occurs before the second event. For example, action a happens before b if and only if $event_a$ for a is greater than $event_b$ for b. This function is not only used to assert the program order of statements within the same thread, but also to assert, for any matched pair, that the send operation occurs before the receive.

$$\begin{array}{l}(\textbf{define-type}\ \texttt{R}\ (\textbf{record}\ \textit{M::int e::int value::int event::int}))\\(\textbf{define-type}\ \texttt{S}\ (\textbf{record}\ \textit{O::int ID::int e::int value::int event::int}))\end{array} \quad (2)$$

We define two records in equation (2) that are essential to encoding our SMT problem. The first record R defined above represents a receive action such that M, e, $value$, $event \in \mathbb{N}$. M represents a variable that is assigned the $event$ field of the matched record S. e is the endpoint of the receive action R, $value$ represents a value received by the receive operation, and $event$ is the event term which indicates the program order of the receive action R. The second record S defined above is a send action such that O, ID, e, $value$, $event \in \mathbb{N}$. The fields e and $event$, as those in tuple R, are the destination endpoint and event term, respectively. The field $value$ is the value being sent. O is a variable used for match pairs, and it is assigned an event term of a receive operation if the receive operation is to be matched to the send operation. ID is a unique number that identifies each send operation. Note that the integer value assigned to $event$ indicates the program order. Also, we use two match variables, M and O, for different purposes. M is used for preventing two receive operations from being matched with the same send operation. O is used for ordering two receive operations with respect to the program order of two matched send operations. The SMT solver must assign the M and $event$ fields of the send tuple, and it must assign the $event$, M, and $value$ fields of the receive tuple in a way that is consistent with all the constraints in the encoding.

$$\begin{array}{l}(\textbf{define}\ \texttt{MATCH} :: (\rightarrow\ \textit{r::}\texttt{R}\ \textit{s::}\texttt{S}\ (\textbf{subtype}\ \textit{(c::bool)}\ (\textit{ite}\\ \quad (\text{and}\ (=\ (\text{select}\ r\ e)\ (\text{select}\ s\ e))\\ \quad\quad (=\ (\text{select}\ r\ \textit{value})\ (\text{select}\ s\ \textit{value}))\\ \quad\quad (\text{HB}\ (\text{select}\ s\ \textit{event})\ (\text{select}\ r\ \textit{event}))\\ \quad\quad (=\ (\text{select}\ r\ M)\ (\text{select}\ s\ \textit{ID}))\\ \quad\quad (=\ (\text{select}\ s\ O)\ (\text{select}\ r\ \textit{event})))\\ \quad (=\ c\ \text{true})\\ \quad (=\ c\ \text{false})))))\end{array} \quad (3)$$

The MATCH function in equation (3) takes tuples R and S as parameters. Each "select" statement selects a field of tuple R or S. The function creates a number of constraints that represent the pairing of the specified send and receive operations. This function asserts that the destination endpoint of the send operation is the same as the endpoint used by the receive operation, that the value sent by the send operation is the same value received by the receive operation, that the send operation occurs before the receive operation, that the "match" value (*M*) in the receive tuple is equal to the "*ID*" value in the send tuple, and that the "*O*" value in the send tuple is equal to the "*event*" value in the receive tuple. Note that

the encoding is going to use the "*O*" value in conjunction with the HB function to enforce message non-overtaking when messages are sent from the same endpoint. Consider a simple example below that sends two messages from a Task 0 to Task 1,

Task 0	Task 1
$S_{0,1}(1, \&h1)$	$R_{1,1}(*, \&h3)$
$S_{0,2}(1, \&h2)$	$R_{1,2}(*, \&h4)$
W(&h1)	W(&h3)
W(&h2)	W(&h4)

The "*O*" values in the send records will be assigned to the order tracking events for $R_{1,1}$ and $R_{1,2}$ in the final encoding. Message non-overtaking from transactions on common endpoints is encoded by forcing the send events to be received in program order using the HB function as below in our simple example:

$$(\texttt{HB}\ (\textit{select}\ S_{0,1}\ O)\ (\textit{select}\ S_{0,2}\ O))$$

$$\begin{array}{l}(\textbf{define}\ \texttt{NE}\ ::\ (\rightarrow\ r1::\texttt{R}\ r2::\texttt{R}\ (\textbf{subtype}\ (c::bool)\ (ite \\ \quad (=\ (\text{select}\ r1\ M)\ (\text{select}\ r2\ M)) \\ \quad (=\ c\ \text{false}) \\ \quad (=\ c\ \text{true})))))\end{array} \tag{4}$$

The NE function in equation (4) is used to assert that no two receive operations are matched with the same send operation. The parameters for this function are two receive tuples. An assertion is made that the values of their "match" fields, *M*, are not equal. This shows that they are paired with two different send operations.

Prior SMT models of MCAPI program executions implicitly compute match pairs by adding possible happens-before relations on sends and receives with the conditions under which those orderings are valid. The SMT solver is then asked to resolve the happens-before relation by choosing specific orders of sends and receives. Match pair encoding, as presented in this work, though it has the same computational complexity as order based encoding (you still need to figure out match pairs on endpoints), is simpler to reason about directly rather than implicitly through orders, results in significantly fewer terms in the SMT problem, and does not restrict out possible matches that exist under infinite-buffer semantics.

The presentation of the SMT encoding is structured as

$$smt\ =\ (defs\ constraints\ match)$$

where the field *defs* represents all definitions of the send, receive operations and the variables; The field *constraints* represents all constraint clauses, including the assert clauses and the clauses built by the HB functions; and the field *match* represents the clauses built by the MATCH function on a set of match pairs and the NE function.

The input for generating the SMT encoding is an execution trace of an MCAPI program together with a set of match pairs. The first trace in Figure 2 with the set of match pairs in Figure 3(a) is such an example. Given an input, the algorithm for generating our encoding is enumerated as follows. For convenience, we use the notation ".event" following each command to represent the *event* field in our encoding.

1. create event variables for every location in the execution trace. Further create send and receive records for all appropriate locations in the trace. Put these in the *def* section;
2. for each assume, add a statement to *constraints*, and for each assert, add a negated assert to *constraints*;
3. add HB relations in *constraints* to assert program order in each thread in the event variables;
4. in the *match* section, add MATCH clauses for the input send and receive pairs. If a receive can match on multiple sends, then inclose each possible match for that receive in a disjunct.

```
                        defs is not shown;

                        constraints;
                        00 (HB R0,2.event W(&h1).event)
                        01 (HB W(&h1).event R0,5.event)
                        02 (HB R0,5.event W(&h2).event)
                        03 (HB W(&h2).event assume.event)
                        04 (HB assume.event assert.event)
                        05 (HB R1,3.event W(&h3).event)
(R0,2 S2,4)             06 (HB W(&h3).event S1,5.event)
(R0,2 S1,5)             07 (HB S1,5.event W(&h4).event)
                        08 (HB S2,4.event W(&h5).event)
(R0,5 S2,4)             09 (HB W(&h5).event S2,7.event)
(R0,5 S1,5)             10 (HB S2,7.event W(&h6).event)
                        11 (assert (> b 0))
(R1,3 S2,7)             12 (assert (not (= a 4)))

                        match;
                        13 (or (MATCH R0,2 S2,4)
                        14 (MATCH R0,2 S1,5))
                        15 (or (MATCH R0,5 S2,4)
                        16 (MATCH R0,5 S1,5))
                        17 (MATCH R1,3 S2,7)
                        18 (NE R0,2 R0,5)
```

(a) (b)

Figure 3. A match pair set and SMT encoding of the system in Figure 1. (a) The match pairs based on endpoints. (b) The SMT encoding where HB creates a happens-before constraint, MATCH creates a match pair constraint, and NE creates a constraint to preclude the use of conflicting match pairs.

5. for two sends $S_{i,j}$ and $S_{i,k}$ in the same task i sent to an identical endpoint such that i < k, if the relation (HB $S_{i,j}$.event $S_{i,k}$.event) is defined in *constraints*, add the clause (HB (select $S_{i,j}$ 0) (select $S_{i,k}$ 0)) to *constraints*.
6. and collect into a NE relation for all receives that are matched with any identical send and add it to *match*.

We use our execution trace in Figure 2 (left side) with the set of match pairs in Figure 3(a) as input to go through the algorithm above for generating an SMT encoding shown in Figure 3(b). Note that we use the wait command with handler in Figure 2 to denote the wait command for a send or receive. First, we use step 1 of the algorithm above to generate the *defs* "area" that is not shown because the definitions are not novel to our solution. Second, we use step 2 to generate the asserts shown on lines 11 and 12 for the assume and assert commands of the original execution trace in Figure 2. The first assert on line 11 works as the assume (line 14 at trace 1) in Figure 2, such that it prevents the SMT solver from finding solutions that are not consistent with control flow which requires "$b \geq 0$". The second assert on line 12 is negated as the goal is to find schedules that violate the property. Third, we use step 3 to generate lines 00 - 10, which asserts program order within each thread. Finally, we use steps 4 and 6 to generate the *match* "area" of the SMT encoding. In particular, we send the set of match pairs in Figure 3(a) to the algorithm and use step 4 to generate a MATCH statement for each match pair and collect those on the same receive into a disjunction on line 13 - 17; and finally add the NE relation by step 6 that precludes $R_{0,2}$ and $R_{0,5}$ from matching to the same send on line 18.

Other than the basic structure of the SMT encoding, we use the function

$$\text{ANS}(smt) \mapsto \{\textbf{SAT}, \textbf{UNSAT}\}$$

to return the solution of an SMT problem, such that **SAT** represents a satisfiable solution that finds a trace of the MCAPI program execution that violates the user defined correctness property, and **UNSAT** represents an unsatisfiable solution indicating that all possible execution traces either meet the correctness property in

```
defs is not shown;

constraints;
00 (HB R0,2.event W(&h1).event)
01 (HB W(&h1).event R0,5.event)
02 (HB R0,5.event W(&h2).event)
03 (HB W(&h2).event assume.event)
04 (HB assume.event assert.event)
05 (HB R1,3.event W(&h3).event)
06 (HB W(&h3).event S1,5.event)
07 (HB S1,5.event W(&h4).event)
08 (HB S2,4.event W(&h5).event)
09 (HB W(&h5).event S2,7.event)
10 (HB S2,7.event W(&h6).event)
11 (assert (> b 0))
12 (assert (not (= a 4)))

match;
13 (MATCH R0,2 S2,4)
14 (MATCH R0,5 S1,5)
15 (MATCH R1,3 S2,7)
```

(a)

```
defs is not shown;

constraints;
00 (HB R0,2.event W(&h1).event)
01 (HB W(&h1).event R0,5.event)
02 (HB R0,5.event W(&h2).event)
03 (HB W(&h2).event assume.event)
04 (HB assume.event assert.event)
05 (HB R1,3.event W(&h3).event)
06 (HB W(&h3).event S1,5.event)
07 (HB S1,5.event W(&h4).event)
08 (HB S2,4.event W(&h5).event)
09 (HB W(&h5).event S2,7.event)
10 (HB S2,7.event W(&h6).event)
11 (assert (> b 0))
12 (assert (not (= a 4)))

match;
13 (MATCH R0,2 S1,5)
14 (MATCH R0,5 S2,4)
15 (MATCH R1,3 S2,7)
```

(b)

Figure 4. Two SMT encodings of the system in Figure 1. (a) The SMT encoding for Trace 1 in Figure 2. (b) The SMT encoding for Trace 2 in Figure 2.

the same control flow, or follow a different control flow. Note that **UNSAT** and **SAT** are ordered such that **UNSAT** $<$ **SAT**.

The SMT encoding defined above is used to capture the non-deterministic behavior of an MCAPI program execution by giving a complete set of match pairs. As we discussed in the previous section, the MCAPI program in Figure 1 contains two outcomes of execution as defined in the MCAPI specification under the infinite-buffer semantics. The SMT encoding we present in Figure 3(b) captures both execution traces, since the set of match pairs in Figure 3(a) is a complete set where all matches that can occur in the real execution for our running example in Figure 1 are included, and all matches that cannot occur in the real execution are not included. Further, the following theorem states that we can over-approximate the true set of match pairs and still prove correctness. If there is no error with the over-approximated set, then there is no error arising from non-determinism in the runtime on that program execution. If there is an error from the over-approximated set, that error is also guaranteed to be a real error in the program runtime. Note that two SMT problems smt_α and smt_β in the following theorem have identical *defs* and *constraints* sets, and the match set of smt_α is the subset of that of smt_β so that smt_β represents an over-approximation of smt_α.

Theorem 1. The relation for the solutions of two SMT problems $smt_\alpha = (defs\ constraints\ match_\alpha)$ and $smt_\beta = (defs\ constraints\ match_\beta)$ is,

$$\mathrm{ANS}(smt_\alpha) \leq \mathrm{ANS}(smt_\beta)$$

where $set(match_\alpha) \subseteq set(match_\beta)$. Note that $set(match)$ represents the input set of match pairs for the `MATCH` clauses in the $match$ field.

Proof Sketch. Consider the MCAPI program in Figure 1 as an example. Figure 4(a) and (b) are two different SMT encodings for our running example in Figure 1 generated from different sets of possible match pairs, such that Figure 4(a) encodes trace 1 in Figure 2 and Figure 4(b) encodes trace 2 in Figure 2. By solving the encodings in Figure 4(a) and (b) for trace 1 and 2 in Figure 2 respectively, we get an unsatisfiable solution for Figure 4(a), and a satisfiable solution for Figure 4(b). As we discussed in Section 2, trace 1 is an execution trace without failure of the assertion, and trace 2 is the one that fails the assertion. Compare both encodings with that in Figure 3(b), we find that the fields *defs* and *constraints* are identical except for the `MATCH` clauses. In particular, the input set of match pairs of either Figure 4(a) or (b) is the subset of that in Figure 3(b). As discussed above, the encoding in Figure 3(b) captures the non-deterministic behavior of our running program in Figure 1, which encodes trace 1 and 2 in Figure 2 into one single SMT problem. Thus, an SMT solver will return a satisfiable solution for the encoding in Figure 3(b). Thus, ANS on the encoding in Figure 3(b) is greater than or equal to the ANS on the encoding in either Figure 4(a) or (b). In other words, adding more match pairs can only move the ANS from **UNSAT** to **SAT**.

The formal proof of Theorem 1 is in the long version of our paper at ("http://students.cs.byu.edu/∼yhuang2 /downloads/paper.pdf"). The proof defines a formal operational semantics given by a term rewriting system using a *CESK*[1] style machine only the machine is augmented to include additional structure for modeling message passing. The operational framework defines how to execute a program, following the specified trace, and defines when that execution is a success (causes no assertion violation), a failure (causes an assertion violation), infeasible (causes an assume to not hold), or an error (execution is not allowed by the MCAPI semantics.). Further, the machine generates the terms of the SMT encoding as it rewrites the machine states. The proof defines a combination operator and shows that several SMT encodings can be combined such that the combined SMT encoding returns "SAT" if one of those encodings has a satisfiable solution. As such, Theorem 1 is formally proved by applying the combination operator for smt_α and smt_β.

Given $set(match_\alpha)$ and $set(match_\beta)$ in the theorem above a complete set of match pairs and an over-approximated set, respectively, we can further prove that $\mathrm{ANS}(smt_\alpha) = \mathrm{ANS}(smt_\beta)$ by giving the following theorem. Note that a match pair $(R, S) \in set(match_\beta)/set(match_\alpha)$ is called "bogus", since it cannot exist in a real execution of the program.

Theorem 2. Any match pair (R, S) used in a satisfying assignment of an SMT encoding smt is a valid match pair and reflects an actual possible MCAPI program execution.

Proof. Proof by contradiction. Assume that (R, S) is a "bogus" match pair that causes $\mathrm{ANS}(smt) =$ **SAT**. Since (R, S) is not a valid match pair, match R and S requires program order, message non-overtaking, or no-multiple match to be violated. In other words, the `HB` constraints encoded in smt are not satisfied. Based on the fact above, the answer of smt is **UNSAT** and it contradicts the previous hypothesis. Thus, (R, S) is a valid match pair in smt and reflects an actual possible MCAPI program execution. □

By proving Theorem 2, we infer that a "bogus" match pair can only cause an unsatisfying assignment of an SMT problem. Further, given that $set(match_\alpha)$ and $set(match_\beta)$ reflect a complete set and a over-approximated set respectively, the answers of smt_α and smt_β discussed above are equal since any "bogus" match pair involved in smt_β is only used in unsatisfying assignments.

It is possible to use our encoding for *zero-buffer* semantics as well. For *zero-buffer* semantics, reorganize the encoding as follows: For each match pair that cannot exist in the program runtime under the *zero-buffer* setting, add extra `HB` clauses in the SMT encoding to prevent incorrect behavior. For example in Figure 1, to prevent $\mathtt{R}_{1,3}$ from being matched with $\mathtt{S}_{2,6}$ before $\mathtt{R}_{0,2}$ is matched with $\mathtt{S}_{2,4}$ as prohibited by the *zero-buffer* semantics, an `HB` relation is added such that the wait command `W(&h1)` for $\mathtt{R}_{0,2}$ happens before $\mathtt{S}_{2,6}$.

[1] The *CESK* machine state is represented with a **C**ontrol string, **E**nvironment, **S**tore, and **K**ontinuation.

```
//initialization
input an MCAPI program
initialize list_r by traversing each task of the program and storing
each receive command
initialize list_s by traversing each task of the program and storing
each send command

//check each receive and send with the same endpoint
for r in list_r
  for s in list_s
    //check matching criteria for r and send
    if
      1.the endpoint of r is equal to the destination endpoint of s, and
      2.the index of r is larger or equal to the index of s, and
      3.the index of r is less or equal to the index of s
        plus the size of list_s minus the number of sends with the same
        source and destination endpoints of s
    then
      add pair (r, s) to match_set
    else
      do next iteration
    end if
  end for
end for

output match_set;
```

Figure 5. Pseudocode for generating over-approximated match pairs

4. Generating Match Pairs

The exact set of match pairs can be generated by simulating the program trace and using a depth-first search to enumerate non-determinism arising from concurrent sends and receives. Such an effort, however, solves the very problem we are trying to solve because if you simulate the program trace exploring all non-determinism, then you may as well verify all runtime choices for property violations at the same time. In this section, we present an algorithm that does not require an exhaustive enumeration of runtime behavior in simulation. Our algorithm over-approximates the match pairs such that match pairs that can exist in the runtime are all included and some "bogus" match pairs that cannot exist in the runtime may or may not be included.

The algorithm generates the over-approximated match pair set by matching each pair of the send and receive commands at common endpoints and then pruning obvious matches that cannot exist in any runtime implementation of the specification. Figure 5 presents the major steps of the algorithm. The algorithm proceeds by first linearly traversing each task of the program storing each receive and send command into two distinct structured lists. The receive list `list_r` is structured as in (5) and the send list `list_s` is structured as in (6).

$$\begin{array}{l}(\mathtt{e_0} \rightarrow ((0,\ \mathtt{R_{0,1}}),\ (1,\ \mathtt{R_{0,2}}),\ \ldots))\\ (\mathtt{e_1} \rightarrow ((0,\ \mathtt{R_{1,1}}),\ (1,\ \mathtt{R_{1,2}}),\ \ldots))\\ \ldots\\ (\mathtt{e_n} \rightarrow ((0,\ \mathtt{R_{n,1}}),\ (1,\ \mathtt{R_{n,2}}),\ \ldots))\end{array} \tag{5}$$

The list in (5) keeps record of the receive commands uniquely identified by the endpoint e_x, where x is the number of the endpoint. The integer in the first field of each pair indicates the program order of the receive commands within each task on the specified endpoint. Note that the receive command with lower program order should be served first in the program runtime. The receive command in the second field of the list is defined as a record R in equation 2 of the previous section. The notations $\mathtt{R_{0,1}}$, $\mathtt{R_{0,2}}$, etc. represent the unique identifiers for the receives. Recall that we operate on a program trace so there are no loops and all instances of send and receive operations are uniquely identified.

Task 0	Task 1	Task 2
$\mathtt{R_{0,1}}(*, \&\mathtt{h1})$	$\mathtt{S_{1,1}}(0, \&\mathtt{h5})$	$\mathtt{S_{2,1}}(0, \&\mathtt{h8})$
$\mathtt{R_{0,2}}(*, \&\mathtt{h2})$	$\mathtt{R_{1,2}}(*, \&\mathtt{h6})$	
$\mathtt{S_{0,3}}(1, \&\mathtt{h3})$	$\mathtt{S_{1,3}}(0, \&\mathtt{h7})$	
$\mathtt{R_{0,4}}(*, \&\mathtt{h4})$		

Figure 6. Another MCAPI concurrent program

$$\begin{array}{l}\text{“}dst\text{”}\quad\text{“}src\text{”}\qquad\qquad\qquad\qquad\text{“}src\text{”}\\ (\mathtt{e_0} \rightarrow ((\mathtt{e_1} \rightarrow ((0,\ \mathtt{S_{1,1}}),\ (1,\ \mathtt{S_{1,2}}),\ \ldots),\ (\mathtt{e_2} \rightarrow (\ldots),\ \ldots))))\\ (\mathtt{e_1} \rightarrow ((\mathtt{e_0} \rightarrow ((0,\ \mathtt{S_{0,1}}),\ (1,\ \mathtt{S_{0,2}}),\ \ldots),\ (\mathtt{e_2} \rightarrow (\ldots),\ \ldots))))\\ \ldots\\ (\mathtt{e_n} \rightarrow ((\mathtt{e_0} \rightarrow ((0,\ \mathtt{S_{0,3}}),\ (1,\ \mathtt{S_{0,4}}),\ \ldots),\ (\mathtt{e_1} \rightarrow (\ldots),\ \ldots))))\end{array} \tag{6}$$

Also, we keep record of the send commands using the second list in (6) that is uniquely identified by the destination endpoint e_x. Each sublist for the destination endpoint is uniquely identified by the source endpoint e_y. Similarly, the integer in the first field of each pair indicates the program order of the send command within the same task from a common source and to a common endpoint. The send command in the second field of the list is defined as a record S in equation 2 of the previous section. The notations $\mathtt{S_{1,1}}$, $\mathtt{S_{1,2}}$, etc. represent the unique identifiers for the sends.

Figure 6 is an example program in our shorthand notation to present our algorithm. The sends $\mathtt{S_{1,1}}$,$\mathtt{S_{1,3}}$ and $\mathtt{S_{2,1}}$ have Task 0 as an identical destination endpoint. The send $\mathtt{S_{0,3}}$ has Task 1 as the destination endpoint. In our running example in Figure 6, we generate our receive list and send list in equation (7) and (8), respectively.

$$\begin{array}{l}(0 \rightarrow ((0,\ \mathtt{R_{0,1}}),\ (1,\ \mathtt{R_{0,2}}),\ (2,\ \mathtt{R_{0,4}})))\\ (1 \rightarrow ((0,\ \mathtt{R_{1,2}})))\end{array} \tag{7}$$

$$\begin{array}{l}(0 \rightarrow ((1 \rightarrow ((0,\ \mathtt{S_{1,1}}),\ (1,\ \mathtt{S_{1,3}})),\ (2 \rightarrow ((0,\ \mathtt{S_{2,1}}))))))\\ (1 \rightarrow ((0 \rightarrow ((0,\ \mathtt{S_{0,3}})))))\end{array} \tag{8}$$

Note that $\mathtt{R_{0,1}}$ is the first receive operation in endpoint 0, and $\mathtt{S_{2,1}}$ is the first send from the source endpoint 2 to the destination endpoint 0.

The second step for our algorithm is to linearly traverse both lists to generate match pairs between send and receive commands. Observe that the sends defined with an identical destination endpoint and an identical source endpoint are processed in a FIFO order according to the MCAPI runtime specification. The same is true of receives on a common endpoint in a common task. The FIFO ordering, sometimes referred to as message non-overtaking, lets us prune match pairs that obviously cannot be generated in any valid implementation of the MCAPI runtime.

Let I_r and I_s be the program order indication in the lists of (5) and (6) respectively and similarly `R` and `S` be the corresponding actions. Note that I_r and I_s are increased by 1 for two consecutive receives or sends, respectively. Further, N_s is the number of send actions in the program and $n_s(src, dst)$ is the number of sends from src to dst, where src and dst represent the source and destination endpoints, respectively. A send and receive can be matched if and only if

1. the destination endpoint of `S` is identical to the endpoint of `R`;
2. $I_r \geq I_s$;
3. and $I_r \leq I_s + (N_s - n_s(src, dst))$.

If all rules of the criteria are satisfied for a send and a receive, we build a match pair for the send and the receive in the result set `match_set`. In our concrete example, $\mathtt{R_{0,1}}$ is matched with $\mathtt{S_{1,1}}$ or $\mathtt{S_{2,1}}$, but it cannot be matched with $\mathtt{S_{1,3}}$ since the second rule is not satisfied such that the order for $\mathtt{R_{0,1}}$ is less than the order of $\mathtt{S_{1,3}}$.

	Program Order	Matches	Assume&Assert	Extra Clauses
Our Encoding	11	4	2	0
Encoding in [9]	22	13	3	8

Table 1. Comparison of two encodings for the MCAPI program execution in Figure 1.

We repeatedly apply the criteria to match sends and receives until we completely traverse the lists in (5) and (6). The generated set of match pairs for our example in Figure 6 is imprecise such that some extra match pairs, which we call "bogus", that cannot exist in the real execution trace are included. In particular, the match pair $(\mathtt{S}_{2,1}\ \mathtt{R}_{0,4})$ is a "bogus" match pair because it is not possible to order $\mathtt{S}_{1,3}$ before $\mathtt{R}_{0,2}$ since $\mathtt{R}_{1,2}$ can only match with $\mathtt{S}_{0,3}$ that must occur after $\mathtt{R}_{0,2}$. Fortunately, the encoding in this paper is strong enough to preclude that bad match pair in any satisfying solution.

Now let us analyze the time complexity of the algorithm. Traversing the tasks linearly takes $O(N)$ to complete, where N is the total lines of code of the program. Traversing the list of receives and the list of sends takes $O(mn)$ to complete, where m is the total number of sends and n is the total number of receives. Note that $m + n \leq N$. Totally, the algorithm takes $O(N + mn) \leq O(N + N^2) \approx O(N^2)$ to complete.

5. Experiments and Results

We used the MCA provided reference solution to generate MCAPI program traces. The reference solution uses the *PThread* library to create multiple MCAPI tasks. Our tool takes the trace as input, computes the match-pair set, and outputs the trace encoding. We use the *Yices* SMT solver [7] to check the satisfiability of the generated SMT encoding.

We use *zero-buffer* semantics and compare our results to that in [9]. Two sets of benchmarks are used in the comparison. The first set consists of four "toy" examples, including the example in Figure 1. The other three benchmarks are generated manually. *Small1* is a program execution with two tasks, where two sends are in the first task, and two receives are in the second task. *Small2* is a program execution with only one task, where one send and one receive are contained. *Small3* is a program execution with three tasks, where three sends are in the first task, two receives are in the second task, and one receive is in the third task. Those examples are very short execution traces. The other set of benchmarks consists of five large program traces. One of them, we call "leader election", basically elects a leader from several candidates by message passing. The other four program traces, are all extensions to the one in Figure 1. In particular, extra iterations are added to the original program execution to generate longer execution traces. In all benchmarks, the correctness properties are numerical assertions over variables. The experiments are conducted on a PC with a 1.6 GHz Intel Quad Core processor and 8GB memory running Ubuntu 14.

As for the first set of benchmarks, Table 1 shows the comparison of our encoding and the encoding in [9] on the example in Figure 1. Basically, we encode 17 clauses for modeling the program execution in Figure 1, omitting the definition of variables at the beginning of the encoding. Instead, the encoding in [9] has 47 clauses for the same program execution, omitting the definitions as well. The number of clauses that constrain program order, match pairs, assume/asserts, and any other extra uncategorized clauses are also shown in Table 1. Program order comprises the bulk of the encoding. The extra clauses in [9] come from auxiliary variables in the encoding.

Table 2 shows the properties of our encodings and the encodings in [9] on each benchmark. The SMT solver returns the correct answer of our encoding under *zero-buffer* semantics for each example shown in Table 2. In particular, our encodings return "sat" (a satisfiable solution) for the example in Figure 1 and *Small3* since violations are detected for both scenarios. Also, our encodings return "unsat" (an unsatisfiable solution) for *Small1* and *Small2*, meaning there are no errors in program runtimes. From the results, it is also shown that the encoding in [9] does not always encode the correct program behavior. In addition to the correctness properties, the number of clauses shown in Table 2 follows the performance of the SMT solver with fewer clauses leading to better runtime. Table 2 shows that our encoding generates 1/5 – 1/3 the clauses of that of the encoding in [9]. We believe these reductions will generalize to any given program trace and match pair set.

Test Programs		Our Encoding		Encoding in [9]	
Name	# Messages	Property	# Clauses	Property	# Clauses
EP in Figure 1	3	sat	17	unsat	47
Small1	2	unsat	8	unsat	33
Small2	1	unsat	4	unsat	18
Small3	3	sat	11	unsat	44

Table 2. Comparison of two encodings for four "toy" MCAPI program executions with property and clause number.

Test Programs		Our Encoding		Encoding in [9]	
Name	# Messages	Time(ms)	Memory(MB)	Time(ms)	Memory(MB)
LeaderElect	30	–	–	–	–
5iterations	15	72	18.5195	392	37.2188
6iterations	18	72	18.918	636	48.7031
7iterations	21	80	19.375	940	62.7188
8iterations	24	–	–	–	–

Table 3. Comparison of two encodings for five "large" MCAPI program executions with runtime and used memory.

As for the second set of benchmarks, we also use *zero-buffer* semantics to match [9]. The runtime and memory usage for each program execution in Table 3 are compared. In this series of experiments, the encodings in [9] are not capable of resolving all the non-deterministic behaviors without *infinite-buffer* semantics. As such, we revise our encodings for the same scenarios in *zero-buffer* setting in order to obtain the same behaviors of the encodings in [9]. Table 3 shows the experimental results for the "large" MCAPI program executions. The runtimes and the memory usage of our encodings are smaller than those of the encodings in [9] for all benchmarks in Table 3. In average, our encodings run eight times faster than the encodings in [9] and use two times less memory than the encodings in [9].

By comparing to the encoding in [9] on two series of experiments shown above, our encoding is demonstrated that it can correctly encode the non-determinism of an MCAPI program execution, and can be executed efficiently.

6. Related Work

Morse et al. provided a formal modeling paradigm that is callable from C language for the MCAPI interface [15]. This model correctly captures the behavior of the interface and can be applied for model checking analysis for C programs that use the API.

In [19], Vakkalanka et al. provided POE, a DPOR algorithm [10] of their dynamic verifier, ISP, for MPI programs. POE explores all interleavings of an MPI program, and is able to detect hidden deadlocks under *zero-buffer* setting. If *infinite-buffer* is applied, however, some interleavings may not be found. [19] also defines *Intra-Happens-Before-Order*(*Intra-HB*), a set of partial order of MPI events that constrains the Happens-Before relations between

commands in any control flow path. Those relations are essential to their POE algorithm of their dynamic verifier ISP. Our SMT structure encodes all the relations in *Intra-HB*, such that it can follow any control flow path executed by the algorithm in [19].

In [18], Sharma et al. provided MCC, a dynamic verifier for MCAPI programs. MCC systematically explores all interleavings of an MCAPI program by concretely executing the program repeatedly. MCC uses DPOR [10] to reduce the redundant interleavings of the execution, however, it does not include all possible traces allowed by the MCAPI specification. Wang et al. provided a symbolic algorithm that detects errors in all feasible permutations of statements in an execution trace in the shared memory system [20]. In this method, the program is partitioned into several concurrent trace programs (CTPs), and the encoding for each CTP is verified using a satisfiability solver. Elwakil et al. provided a similar work with ours [8, 9]. In their work, the method of [20] is used and adapted to the message passing system. As shown in the previous section of this paper, their method does not correctly encode all possible execution traces of an MCAPI program.

The SMT/SAT based Bounded Model Checking is one avenue of verifying and debugging systems. Burckhardt et al. presented CheckFence prototype in [5], which exhaustively checks all executions of a test program by translating the program implementation into SAT formulas. The standard SAT solver can construct counterexample if incorrect executions exist. Instead of over-approximating at the beginning and then further compressing the observations, CheckFence in [5] increments the observations each time step by adding constraints to SAT formulas. Dubrovin et al. provides a method in [6] that translates an asynchronous system into a transition formula for three partial order semantics. Other than adding constraints to the SAT/SMT formulas in order to compress the search space, the method in [6] decreases the search bound by allowing several actions to be executed simultaneously within one step. Kahlon et al. presented a Partial Order Reduction method called *MPOR* in [12]. This method cannot only be applied to the explicit state space search as other partial order reduction methods do, but also can be applied to the SMT/SAT based Bounded Model Checking. *MPOR* guarantees that exactly one execution is calculated per each Mazurkiewicz trace, in order to reduce the search space. There are several applications of the SMT/SAT based Bounded Model Checking. [14] presented a precise verification of heap-manipulating programs using SMT solvers. [13] presented some challenges in SMT-based verification for sequential system code, and tackled these issues by extending the standard SMT solvers.

The application of static analysis is another interesting thread of research to test or debug message passing programs. [21] and [4] are the approaches for MPI [1], another message passing interface standard. [11] presented a system that uses static analysis to determine offline the topology of the communications and nodes in the input MCAPI program.

7. Conclusions and Future Work

We have presented an SMT encoding of an MCAPI program execution that uses match pairs rather than the state-based or order-based encoding in the prior work. Unlike the existing method of SMT encoding, our encoding is the first encoding that correctly captures the non-deterministic behavior of an MCAPI program execution under *infinite-buffer* semantics allowed in the MCAPI specification. In this paper, we proved that we can generate such an encoding by giving an execution trace and a complete set of match pairs. Further, we have proved that the same results can be obtained even if the match pairs are over-approximated as input. Also, we have provided an algorithm with $O(N^2)$ time complexity that over-approximates the true set of match pairs, where N is the total number of code lines of the program. By comparing to the existing work in [9], the experimental results shows that our encoding is capable of capturing correct behavior of an MCAPI program execution. Further, our encoding reduces 70% clauses of that of the work in [9]. Also, it runs average eight times faster and uses two times less memory than that in [9].

The precision of the set of match pairs is essential to the efficiency of SMT encodings. Currently we have an over-approximated generation method which still keeps several "bogus" match pairs in the generated set. New methods for generating a much more precise set of match pair are required. Also The method of DPOR are considered to combine with our SMT encoding to generate the execution traces. We can then use the SMT encoding to further improve the DPOR technique.

References

[1] MPI: A message-passing interface standard. `http://www.mpi-forum.org/docs/mpi-2.2/index.htm`

[2] The multicore association. `http://www.multicore-association.org`

[3] The multicore association resource management API. `http://www.multicore-association.org/workgroup/mcapi.php`

[4] Bronevetsky, G.: Communication-sensitive static dataflow for parallel message passing applications. In: CGO. IEEE Computer Society (2009)

[5] Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: ACM SIGPLAN PLDI. San Diego, California, USA (June 10–13, 2007)

[6] Dubrovin, J., Junttila, T., Heljanko, K.: Exploiting step semantics for efficient bounded model checking of asynchronous systems. In: Science of Computer Programming. pp. 77(10–11):1095–1121 (2012)

[7] Dutertre, B., de Moura Leonardo: A fast linear-arithmetic solver for DPLL(T). In: CAV. vol. 4144 of LNCS, pp. 81–94. Springer-Verlag (2006)

[8] Elwakil, M., Yang, Z.: CRI: Symbolic debugger for mcapi applications. In: Automated Technology for Verification and Analysis (2010)

[9] Elwakil, M., Yang, Z.: Debugging support tool for mcapi applications. In: PADTAD '10: Proceedings of the 8th Workshop on Parallel and Distributed Systems (2010)

[10] Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for modeling checking software. In: POPL. pp. 110–121. ACM Press, New York, NY, USA (2005)

[11] Gray, I., Audsley, N.: Targeting complex embeeded architectures by combining the multicore communications API (mcapi) with compile-time virtualisation. In: LCTES. ACM, Chicago, Illinois, USA (2011)

[12] Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: ACM CAV. pp. 398–413. Springer, Berlin/Heidelberg, Grenoble, France (June 26–July 02, 2009)

[13] Lahiri, S.: SMT-based modular analysis of sequential systems code. In: CAV. Springer-Verlag (2011)

[14] Lahiri, S., Qadeer, S.: Back to the future revisiting precise program verification using SMT solvers. In: POPL. ACM, San Francisco, California, USA (2008)

[15] Morse, E., Vrvilo, N., Mercer, E., McCarthy, J.: Modeling asynchronous message passing for C program. In: VMCAI. vol. 7148 of LNCS (2012)

[16] de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

[17] Netzer, R., Brennan, T., Damodaran-Kamal, S.: Debugging race conditions in message-passing programs. In: ACM SIGMETRICS Symposium on Parallel and Distributed Tools. pp. 31–40. Philadelphia, PA, USA (1996)

[18] Sharma, S., Gopalakrishanan, G., Mercer, E., Holt, J.: Mcc - a runtime verification tool for mcapi user applications. In: FMCAD (2009)

[19] Vakkalanka, S., Vo, A., Gopalakrishnan, G., Kirby, R.: Reduced execution semantics of mpi: From theory to pratice. In: FM. pp. 724–740 (2009)

[20] Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: ESEC/FSE. pp. 23–32. ACM, New York, NY, USA (2009)

[21] Zhang, Y., Evelyn, D.: Barrier matching for programs with textually unaligned barriers. In: PPoPP. pp. 194–204. ACM, San Jose, California, USA (2007)