Proving MCAPI Executions are Correct using SMT

Yu Huang, Eric Mercer and Jay McCarthy
Department of Computer Science
Brigham Young University
Provo, UT 84602
Email: {yu,egm,jay}@cs.byu.edu

***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 provides a way to encode an MCAPI execution as a Satisfiability Modulo Theories (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. In other words, our encoding with infinite-buffer controls more non-deterministic behaviors at runtime. Our results demonstrate that our SMT encoding, restricted to zero-buffer semantics, uses fewer clauses when compared to another zero-buffer technique. Our encoding also runs faster and uses less memory than that same technique. Further, our encoding scales well for programs with large number of messages and massive non-deterministic behaviors.

***Index Terms*—abstraction, refinement, SMT, message passing**

I. 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) is an industry group that has formed to define specifications for low-level communication, resource management, and task management for embedded heterogeneous multicore devices [2].

One specification that the MCA has released is the Multicore Association Communications API (MCAPI) [1]. The specification defines types and functions for 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 the possibility of a race between multiple messages to common endpoints thus giving rise to non-deterministic behavior in the runtime[18]. If an application has non-determinism, it is not possible to test and debug such an application without a way to directly (or indirectly) control the MCAPI runtime.

There are two ways to implement the MCAPI semantics: infinite-buffer semantics (the message is copied into a runtime buffer on the API call) and zero-buffer semantics (the message has no buffering) [20]. An infinite-buffer semantics provides more non-deterministic behaviors in matching send and receives because the runtime can arbitrarily delay a send to create interesting (and unexpected) send reorderings. The zero-buffer semantics follow intuitive message orderings as a send and receive essentially rendezvous.

Sharma et al. propose a method of indirectly controlling the MCAPI runtime to verify MCAPI programs under zero-buffer semantics, but it was later discovered that this method does not completely explore the entire execution space of certain kinds of MCAPI programs [19]. A key insight of the approach, however, is in its use of match pairs–couplings for potential sends and receives. Wang *et al.* propose an alternative method for resolving non-determinism for program verification using symbolic methods in the context of shared memory systems [21]. The work observes a program trace, builds a partial order from that trace called a concurrent trace program (CTP), and then creates an SMT problem from the CTP that if satisfied indicates a property violation. Elwakil *et al.* extend the work of Wang *et al.* to message passing in zero-buffer semantics, but the proposed encoding does not capture all allowed MCAPI program executions [9]. Further, the 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.

This paper presents an SMT encoding for MCAPI program executions that works for both zero and infinite buffer semantics. Further, the encoding requires fewer terms to capture all possible program behavior when compared to other proposed methods. As a result, the SMT encoding in this paper is both correct in that it enumerates all possible executions allowed by the MCAPI specification in either zero or infinite buffer semantics, and it is efficient since it uses dramatically fewer

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);

Fig. 1. An MCAPI concurrent program execution

terms in the encoding to reduce the memory and running time in the SMT solver. To summarize, the main contributions in this paper are

1) a correct and efficient SMT encoding of an MCAPI program execution that detects all program errors if the provided match pairs are precise or over approximated under zero or infinite buffer semantics; and
2) an $O(N^2)$ algorithm to generate an over-approximation of possible match-pairs, where N is the size of the execution trace in 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 common endpoints. 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 provides a solution to 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 under both zero-buffer and infinite-buffer semantics, as other existing encodings omit critical program behaviors, require more SMT clauses and only support zero-buffer semantics. Section 6 discusses related work, and Section 7 presents our conclusions and future work.

II. 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?"*

2 04 $S_{2,4}$(0, &h5)
2 05 W(&h5)
0 02 $R_{0,2}$(2, &h1)
0 03 W(&h1)
2 06 $S_{2,6}$(1, &h6)
2 07 W(&h6)
0 04 a = atoi(A);
1 03 $R_{1,3}$(2, &h3)
1 04 W(&h3)
1 05 $S_{1,5}$(0, &h4)
1 06 W(&h4)
0 05 $R_{0,5}$(1, &h2)
0 06 W(&h2)
0 07 b = atoi(B);
0 08 assume(b > 0);
0 09 assert(a == 4);

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

The intuitive trace is presented in the first four columns of Figure **??**. 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 $\mathtt{O}_{\mathtt{i,j}}(\mathtt{k}, \&\mathtt{h})$, $\mathtt{O} \in \{\mathtt{S}, \mathtt{R}\}$ or $\mathtt{W}(\&\mathtt{h})$, `i` represents the task number, `j` represents the source line number, `k` represents the destination endpoint, 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. In this example, the program takes 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.

For our example in Figure 1, if we use zero-buffer semantics, 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.

2 04 $\mathtt{S}_{2,4}(0, \&\mathtt{h5})$
2 05 W(&h5)
2 06 $\mathtt{S}_{2,6}(1, \&\mathtt{h6})$
2 07 W(&h6)
1 03 $\mathtt{R}_{1,3}(2, \&\mathtt{h3})$
1 04 W(&h3)
1 05 $\mathtt{S}_{1,5}(0, \&\mathtt{h4})$
1 06 W(&h4)
0 02 $\mathtt{R}_{0,2}(1, \&\mathtt{h1})$
0 03 W(&h1)
0 04 a = atoi(A);
0 05 $\mathtt{R}_{0,5}(2, \&\mathtt{h2})$
0 06 W(&h2)
0 07 b = atoi(B);
0 08 assume(b > 0);
0 09 assert(a == 4);

Fig. 3. XXX Two execution traces of the MCAPI program execution in Figure 1

When using infinite-buffer semantics, however, we can have another scenario shown in the fifth and sixth column of Figure **??** 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 to be 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. 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] or Z3 [16], and the non-deterministic behaviors of the program execution can be resolved.

III. SMT Encoding

Our SMT encoding is based on (1) a trace of events during an execution of an MCAPI program including control-flow assumptions and property assertions, such as Figure 2 and (2) a set of possible match pairs.

A match pair is the coupling of a receive to a particular send. Figure 6(a) is the set of possible match pairs for the program in Figure 1 using our shorthand notation. The set admits, for example, that $\mathtt{R}_{0,2}$ can be matched with either $\mathtt{S}_{1,5}$ or $\mathtt{S}_{2,4}$.

This match pair-based encoding, rather than state-based or order-based encoding [9], [8], is novel.

The purpose of the SMT encoding is to force the SMT solver to resolve the match pairs for the system in such a way that the final values of program variables meet the assumptions on control flow but violate some assertion. In essence, the SMT solver completes a partial order on events into a total order that determines the final match pair relationships.

A. Definitions

The encoding needs to express the partial order imposed by the MCAPI semantics as SMT constraints. The partial order is based on a *Happens-Before* relation over events:

Definition 1 (Happens-Before)**.** *The* Happens-Before (HB) *relation denoted as* $\prec_{\mathtt{HB}}$ *is a partial order.*

Given two events, A and B, if A must complete before B in a valid program execution, then $A \prec_{\mathtt{HB}} B$ will be an SMT constraint.

The constraints on this relation are derived from the program source and match pairs. In order to specify the constraints from the program source, we must map each operation to a order variable. For example,

Definition 2 (Wait)**.** *For a wait operation* W*, the only relevant information is when it occurs, so we create a variable* $order_{\mathtt{W}}$*.*

However, it is not enough to represent all events as simple numbers that will be ordered in this way. This would not allow the solver to discover what values would flow across communication primitives, for example. Instead, we represent each event in the trace as a set of SMT variables that record the pertinent information about the event. For example,

Definition 3 (Send)**.** *For every send operation* S*, we create four variables:*

1) $M_{\mathtt{S}}$*, the order of the matching receive event;*
2) $order_{\mathtt{S}}$*, the order of the send;*
3) $e_{\mathtt{S}}$*, the endpoint; and,*
4) $value_{\mathtt{S}}$*, the transmitted value.*

The most complex operation in MCAPI is a receive. Since receives are inherently asynchronous, it is not possible to represent them atomically. Instead, we need to associate each receive with a wait where the program expects the operation

to be completed by. A single wait can be associated with many receives due to *the message non-overtaking property* required by MCAPI. We refer to this wait as the *nearest-enclosing wait*.

Definition 4 (Nearest-Enclosing Wait)**.** *A wait that witnesses the completion of a receive by indicating that the message is delivered and that all the previous receives in the same task issued earlier are complete as well.*

Task 0	Task 1
$\mathtt{R}_{0,1}(*, \&\mathtt{h1})$	$\mathtt{S}_{1,1}(0, \&\mathtt{h3})$
$\mathtt{R}_{0,2}(*, \&\mathtt{h2})$	$\mathtt{W}(\&\mathtt{h3})$
$\mathtt{W}(\&\mathtt{h2})$	$\mathtt{S}_{1,2}(0, \&\mathtt{h4})$
$\mathtt{W}(\&\mathtt{h1})$	$\mathtt{W}(\&\mathtt{h4})$

Fig. 4. Nearest-enclosing Wait example

Figure 4 shows that the wait $\mathtt{W}(\&\mathtt{h2})$ witnesses the completion of the receive $\mathtt{R}_{0,1}$ and $\mathtt{R}_{0,2}$ in Task 0. Thus, $\mathtt{W}(\&\mathtt{h2})$ is their nearest-enclosing wait.

We assume that every receive operation has a nearest-enclosing wait. This assumption disallows a few a misbehaving programs, those that receive but never wait for the operation to complete before continuing. (We could use the same assumption for sends, since MCAPI allows waiting on send operations as well, but we deal with send buffering differently.)

Given this definition, we can define the pertinent information of a receive:

Definition 5 (Receive)**.** *For every receive operation* R *we create five variables:*

1) $M_{\mathtt{R}}$*, the order of the matching send event;*
2) $order_{\mathtt{R}}$*, the order of the receive;*
3) $e_{\mathtt{R}}$*, the endpoint;*
4) $value_{\mathtt{R}}$*, the received valued; and,*
5) $nw_{\mathtt{R}}$*, the order of the nearest enclosing wait.*

B. Constraints

These definitions so far merely establish the pertinent information about each event in the trace as SMT variables. We must also express the constraints on those variables.

The most trivial kind of constraint are those for control-flow assumptions.

Definition 6 (Assumption)**.** *For every assumption* A*, we add it as an SMT assertion.*

It may seem strange to turn *assumptions* into a *assertions*, but from a constraint perspective, the assumption that we have already observed some property (during control-flow) is equivalent to instructing the SMT solver to treat it is inviolate truth, or assertion.

The next level of constraint complexity comes from property assertions. These correspond to the invariants of the program. Our goal is to discover if they can be violated, so we instruct the SMT solver to seek for a way to satisfy their *negation* given all the other constraints.

Definition 7 (Property Assertion)**.** *For every property assertion* P*, we add* $\neg$P *as an SMT assertion.*

Finally, we must express each match pair as a set of SMT constraints. Informally, a match pair equates the shared components of a send and receive and constrains the send to occur before the nearest-enclosing wait of the receive. Formally:

Definition 8 (Match Pair)**.** *A match pair,* $\langle \mathtt{R}, \mathtt{S} \rangle$*, for a receive* R *and a send* S *corresponds to the constraints:*

1) $M_{\mathtt{R}} = order_{\mathtt{S}}$
2) $M_{\mathtt{S}} = order_{\mathtt{R}}$
3) $e_{\mathtt{R}} = e_{\mathtt{S}}$
4) $value_{\mathtt{R}} = value_{\mathtt{S}}$ *and*
5) $order_{\mathtt{S}} \prec_{\mathtt{HB}} nw_{\mathtt{R}}$

However, these match pair constraints are not simply unioned. If we were to do that, then we would be constraining the system such that a single receive were paired with multiple sends if both were valid match pairs. Therefore, we combine all the constraints for a receive's match pairs into a single disjunction:

Definition 9 (Receive Matches)**.** *For each receive* R*, if* $\langle \mathtt{R}, \mathtt{S}_0 \rangle$ *through* $\langle \mathtt{R}, \mathtt{S}_n \rangle$ *are match pairs, then* $\bigvee_i^n \langle \mathtt{R}, \mathtt{S}_i \rangle$ *is used as an SMT constraint.*

This encoding of the input ensures that the SMT solver can only use compatible send/receive pairs and ensures that sends happen "before" receives.

C. *More* Happens-Before *Constraints*

The encoding is not sufficient, though. We must extend it with additional constraints on the *Happens-Before* relation, which we do in four steps. We must ensure that sends to common endpoints occur in program order in a single task (step 1); similarly for receives (step 2); receives occur before their nearest-enclosing wait (step 3); and, that sends are received in the order they are sent (step 4).

Step 1: For each thread, if there are multiple send operations, say S and S′, from that thread to a common endpoint, $e_{\mathtt{S}} = e_{\mathtt{S}'}$, then those sends must follow program order: $order_{\mathtt{S}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$.

Step 2: For each thread, if there are multiple receive operations, say R and R′, from that thread to a common endpoint, $e_{\mathtt{S}} = e_{\mathtt{S}'}$, then those receives must follow program order: $order_{\mathtt{R}} \prec_{\mathtt{HB}} order_{\mathtt{R}'}$.

Step 3: For every receive R and its nearest enclosing wait W, $order_{\mathtt{R}} \prec_{\mathtt{HB}} order_{\mathtt{W}}$.

Step 4: For any pair of sends S and S′ on common endpoints, $e_{\mathtt{S}} = e_{\mathtt{S}'}$, such that $order_{\mathtt{S}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$, then those sends must be received in the same order: $M_{\mathtt{S}} \prec_{\mathtt{HB}} M_{\mathtt{S}'}$.

For example, consider two tasks where Task 0 sends two messages to Task 1 as shown in Figure 5.

The M variables from the sends will be assigned to the orders for $\mathtt{R}_{1,1}$ and $\mathtt{R}_{1,2}$ by the match-pairs selected by the SMT solver. The constraints added in this step force the send

Task 0	Task 1
$\mathtt{S}_{0,1}(1, \&\mathtt{h1})$	$\mathtt{R}_{1,1}(*, \&\mathtt{h3})$
$\mathtt{S}_{0,2}(1, \&\mathtt{h2})$	$\mathtt{R}_{1,2}(*, \&\mathtt{h4})$
$\mathtt{W}(\&\mathtt{h1})$	$\mathtt{W}(\&\mathtt{h3})$
$\mathtt{W}(\&\mathtt{h2})$	$\mathtt{W}(\&\mathtt{h4})$

Fig. 5. Send Ordering Example

to be received in program order using the HB relation which for this example yields $M_{\mathtt{S}_{0,1}} \prec_{\mathtt{HB}} M_{\mathtt{S}_{0,2}}$.

D. Zero Buffer Semantics

The constraints presented so far correspond to an infinite-buffer semantics, because we do not constrain how many messages may be "in transit" at once. We can add additional, orthogonal, constraints to restrict such behavior and enforce a zero-buffer semantics. There are two kinds of such constraints.

First, for each thread, if there are two sends S and S′ such that $order_{\mathtt{S}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$, and S and S′ can both match a receive R, then we add the following constraint to the encoding: $order_{\mathtt{W}(\&\mathtt{h})} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$ where $\mathtt{W}(\&\mathtt{h})$ is a wait function that witnesses the completion of R in execution.

Second, for each pair of sends S and S′, if there is a receive R such that

1) R′ is a preceding receive over S in an identical thread; Or
2) there exists a match pair $\langle \mathtt{R}'', \mathtt{S}'' \rangle$ such that $\langle \mathtt{R}, \mathtt{S} \rangle \rightarrow \langle \mathtt{R}'', \mathtt{S}'' \rangle$ and $\langle \mathtt{R}'', \mathtt{S}'' \rangle \rightarrow \langle \mathtt{R}', \mathtt{S}' \rangle$.

Then we add the following constraint to the encoding: $order_{\mathtt{W}(\&\mathtt{h})} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$ where $\mathtt{W}(\&\mathtt{h})$ is a wait function that witnesses the completion of R in execution.

E. Example

— XXX —

As an intuitive example, we revisit the example in Figure 1. In order to prevent $\mathtt{R}_{1,3}$ from being matched with $\mathtt{S}_{2,6}$ before $\mathtt{R}_{0,2}$ is matched with $\mathtt{S}_{2,4}$, add $event_{\mathtt{W}(\&\mathtt{h1})} \prec_{\mathtt{HB}} event_{\mathtt{S}_{1,5}}$.

Figure 6(b) is the final encoding for the trace which is explained here. The encoding is divided into three sections: *SMT* = {*defs constraints match*}. Stage 1 is not shown because the definitions are not novel to our solution; suffice to say that variables are created as defined. The constraints section is created by stages 2-7. Lines 00 - 06 reflect program orders in the trace. Lines 07 and 08 for the assume and assert commands of the original execution trace in Figure **??**. The first assert on line 07 (line 14 at trace 1) prevents the SMT solver from finding solutions that are not consistent with control flow which requires "$b \geq 0$". The second assert on line 08 is negated as the goal is to find schedules that violate the property. Finally, stage 8 generates the *match* "area" of the SMT encoding. In particular, we send the set of match pairs in Figure 6(a) to the algorithm and generate each match pair and collect those on the same receive into a disjunction on line 09 - 11.

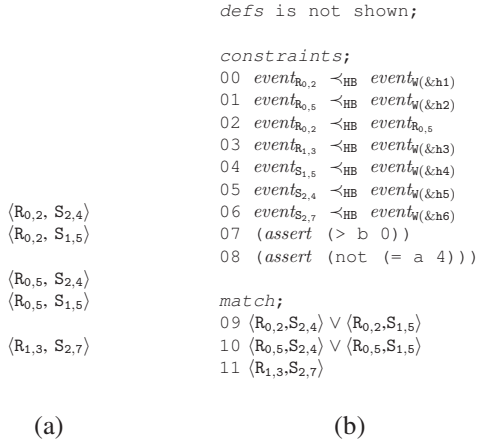

Fig. 6. A match pair set and SMT encoding of the system in Figure 1. (a) The match pairs based on endpoints. (b) The SMT encoding where $\prec_{\mathtt{HB}}$ creates a *Happens-Before* constraint, a pair surrounded by ⟨ and ⟩ creates a match pair constraint, and assume and assert creates an assume and assert, respectively.

F. Correctness

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

$$\mathrm{ANS}(smt) \mapsto \{\mathbf{SAT}, \mathbf{UNSAT}\}$$

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 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 behaviors 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 6(b) captures both execution traces, since the set of match pairs in Figure 6(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 differ only by match set.

Theorem 1. *The relation for the solutions of two SMT prob-*

lems $smt_\alpha = (defs\ constraints\ match_\alpha)$ *and* $smt_\beta = (defs\ constraints\ match_\beta)$ *is,*

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

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

```
defs is not shown;                    defs is not shown;

constraints;                          constraints;
00 event_R0,2 ≺HB event_W(&h1)        00 event_R0,2 ≺HB event_W(&h1)
01 event_R0,5 ≺HB event_W(&h2)        01 event_R0,5 ≺HB event_W(&h2)
02 event_R0,2 ≺HB event_R0,5          02 event_R0,2 ≺HB event_R0,5
03 event_R1,3 ≺HB event_W(&h3)        03 event_R1,3 ≺HB event_W(&h3)
04 event_S1,5 ≺HB event_W(&h4)        04 event_S1,5 ≺HB event_W(&h4)
05 event_S2,4 ≺HB event_W(&h5)        05 event_S2,4 ≺HB event_W(&h5)
06 event_S2,7 ≺HB event_W(&h6)        06 event_S2,7 ≺HB event_W(&h6)
07 (assert (> b 0))                   07 (assert (> b 0))
08 (assert (not (= a 4)))             08 (assert (not (= a 4)))

match;                                match;
09 ⟨R0,2, S2,4⟩                       09 ⟨R0,2, S1,5⟩
10 ⟨R0,5, S1,5⟩                       10 ⟨R0,5, S2,4⟩
11 ⟨R1,3, S2,7⟩                       11 ⟨R1,3, S2,7⟩
```

(a) (b)

Fig. 7. Two SMT encodings of the system in Figure 1. (a) The SMT encoding for Trace 1 in Figure **??**. (b) The SMT encoding for Trace 2 in Figure **??**.

Proof Sketch. Consider the MCAPI program in Figure 1 as an example. Figure 7(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 7(a) encodes trace 1 in Figure **??** and Figure 7(b) encodes trace 2 in Figure **??**. By solving the encodings in Figure 7(a) and (b) for trace 1 and 2 in Figure **??** respectively, we get an unsatisfiable solution for Figure 7(a), and a satisfiable solution for Figure 7(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 6(b), we find that the fields *defs* and *constraints* are identical except for the set of match pairs. In particular, the input set of match pairs of either Figure 7(a) or (b) is the subset of that in Figure 6(b). As discussed above, the encoding in Figure 6(b) captures the non-deterministic behaviors of our running program in Figure 1, which encodes trace 1 and 2 in Figure **??** into one single SMT problem. Thus, an SMT solver will return a satisfiable solution for the encoding in Figure 6(b). Thus, ANS on the encoding in Figure 6(b) is greater than or equal to the ANS on the encoding in either Figure 7(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[1]. The proof defines a formal operational semantics given by a term rewriting system using a *CESK*[2] 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.

Assume $set(match_\alpha)$ 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 $\text{ANS}(smt_\alpha) = \text{ANS}(smt_\beta)$ by giving the following theorem. Note that a match pair $\langle \mathtt{R}, \mathtt{S} \rangle \in set(match_\beta)/set(match_\alpha)$ is called "bogus", since it cannot exist in a real execution of the program.

Theorem 2. *Any match pair* $\langle \mathtt{R}, \mathtt{S} \rangle$ *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 $\langle \mathtt{R}, \mathtt{S} \rangle$ is a "bogus" match pair that causes $\text{ANS}(smt) =$ **SAT**. Since $\langle \mathtt{R}, \mathtt{S} \rangle$ 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, $\langle \mathtt{R}, \mathtt{S} \rangle$ 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, if $set(match_\alpha)$ 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.

IV. 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 entire problem at once 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.

[1] http://students.cs.byu.edu/~yhuang2/downloads/paper.pdf

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

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.

```
// initialization
input an MCAPI program
initialize list_r
initialize list_s

// 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 s
    if
      1. endpoint(r) = destination endpoint(s)
      2. index(r) >= index(s)
      3. index(r) =< index(s)
                     + count(sends(dest=s))
                     - count(sends(src=s, dest=s))
    then
      add pair (r, s) to match_set
    else
      continue
    end if
  end for
end for

output match_set;
```

Fig. 8. Pseudocode for generating over-approximated match pairs

Figure 8 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 (1) and the send list `list_s` is structured as in (2).

$$\begin{array}{l}(\mathtt{e}_0 \rightarrow ((0,\ \mathtt{R}_{0,1}),\ (1,\ \mathtt{R}_{0,2}),\ \ldots))\\ (\mathtt{e}_1 \rightarrow ((0,\ \mathtt{R}_{1,1}),\ (1,\ \mathtt{R}_{1,2}),\ \ldots))\\ \ldots\\ (\mathtt{e}_n \rightarrow ((0,\ \mathtt{R}_{n,1}),\ (1,\ \mathtt{R}_{n,2}),\ \ldots))\end{array} \tag{1}$$

The list in (1) 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 Definition 5 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.

$$\begin{array}{l}\text{“}dst\text{”}\quad\text{“}src\text{”}\qquad\qquad\qquad\qquad\text{“}src\text{”}\\ (\mathtt{e}_0 \rightarrow ((\mathtt{e}_1 \rightarrow ((0,\ \mathtt{S}_{1,1}),\ (1,\ \mathtt{S}_{1,2}),\ \ldots),\ (\mathtt{e}_2 \rightarrow (\ldots),\\ \ldots))))\\ (\mathtt{e}_1 \rightarrow ((\mathtt{e}_0 \rightarrow ((0,\ \mathtt{S}_{0,1}),\ (1,\ \mathtt{S}_{0,2}),\ \ldots),\ (\mathtt{e}_2 \rightarrow (\ldots),\\ \ldots))))\\ \ldots\\ (\mathtt{e}_n \rightarrow ((\mathtt{e}_0 \rightarrow ((0,\ \mathtt{S}_{0,3}),\ (1,\ \mathtt{S}_{0,4}),\ \ldots),\ (\mathtt{e}_1 \rightarrow (\ldots),\\ \ldots))))\end{array} \tag{2}$$

Also, we keep record of the send commands using the second list in (2) 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 Definition 3 of the previous section. The notations $\mathtt{S}_{1,1}$, $\mathtt{S}_{1,2}$, etc. represent the unique identifiers for the sends.

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

Fig. 9. Another MCAPI concurrent program

Figure 9 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 9, we generate our receive list and send list in equation (3) and (4), respectively.

$$\begin{array}{l}(0 \rightarrow ((0,\ \mathtt{R}_{0,1}),\ (1,\ \mathtt{R}_{0,2}),\ (2,\ \mathtt{R}_{0,4})))\\ (1 \rightarrow ((0,\ \mathtt{R}_{1,2})))\end{array} \tag{3}$$

$$\begin{array}{l}(0 \rightarrow ((1 \rightarrow ((0,\ \mathtt{S}_{1,1}),\ (1,\ \mathtt{S}_{1,3})),\ (2 \rightarrow ((0,\ \mathtt{S}_{2,1}))))))\\ (1 \rightarrow ((0 \rightarrow ((0,\ \mathtt{S}_{0,3})))))\end{array} \tag{4}$$

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 (1) and (2) 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, $n_s(*, dst)$ is the number of send actions from any source to dst 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 $I_r \leq I_s + (n_s(*, dst) - n_s(src, dst))$.

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 of $\mathtt{R}_{0,1}$ is less than the order of $\mathtt{S}_{1,3}$. We repeatedly apply the criteria to match sends and receives until we completely traverse the lists in (1) and (2). The generated set of match pairs for our example in Figure 9 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.

V. Experiments and Results

To assess the new encoding in this paper, three experiments with results are presented: a comparison to prior SMT encodings on a zero-buffer semantics, a scalability study on the effects of non-determinism in the execution time on infinite buffer semantics, and an evaluation on typical benchmark programs again with infinite buffer semantics. All of the experiments use the Z3 SMT solver ([16]) and are measured on a 2.40 GHz Intel Quad Core processor with 8 GB memory running Windows 7.

The initial program trace for the experiments is generated using MCA provided reference solution with fixed input to isolate all non-determinism to how sends and receives are matched in the runtime. This statement means that the experiments only consider one path of control flow through the program. Complete coverage of the program for verification purposes would need to generate input to exercise different control flow paths. Where appropriate, the time to generate the match-pair sets from the input trace is reported separately.

A. Comparison to Prior SMT Encoding

To the best knowledge of these authors, the current most effective SMT encoding for verification of message passing program traces is an order-based encoding that captures the happens-before relation directly in the SMT problem [9]. The encoding only exists for zero-buffer semantics, and the tool to generate the encoding is not publicly available. The authors of the order-based encoding, however, graciously encoded several contrived benchmarks used for correctness testing. These benchmarks are best understood as *toy* examples that plumb the MCAPI semantics to clarify intuition on expected behavior.

The zero-buffer encoding in this paper is compared directly to the order-based encoding on the contrived benchmarks. The order-based encoding yields incorrect answers for several programs. Where the order-based encoding returns correct answers, the new encoding, on average, requires 70% fewer clauses, uses half the memory as reported by the SMT solver, and runs eight times faster. The dramatic improvement of the new encoding over the order-based encoding is a direct result of the match-pairs that simplify the happens-before constraints and avoids redundant constraints in the transitive closure of the happens-before relation.

B. Scalability Study

The intent of the scalability study is to understand how performance is affected by the number of messages in the program trace and the level of non-determinism in choosing match-pairs where multiple sends are able to match to multiple receives. The programs for this study consist of a simple pattern of a single thread to receive messages and N threads to send messages. The single thread sequentially receives N messages containing integer values and then asserts that every message did not receive a specific value. In other words, a violation is one where each message has a specific value. The remaining N threads send a message, each containing a different unique integer value, to the single thread that receives. These programs represent the worst-case scenario for non-determinism in a message passing program as any send is able to match with any receive in the runtime, and the assertion is only violated when each send is paired with a specific receive. The SMT solver must search through the multitude of match-pairs, $N \times N$, to find the single precise subset of match-pairs that triggers the violation. In this program structure, there are $N!$ feasible ways to match N sends to N receives.

TABLE I
Scaling as a function of non-determinism

Test Programs		Performance	
N	Feasible Sets	Time (hh:mm:ss)	Memory(MB)
30	30!(∼3E32)	00:00:36	20.11
40	40!(∼8E47)	00:03:22	47.12
50	50!(∼3E64)	00:16:11	102.65
60	60!(∼8E81)	00:47:29	189.53
70	70!(∼1E100)	02:00:30	364.25

The study takes an initial program of $N = 30$, so 31 threads, and varies N to see how the SMT solver scales. A small N is an easy program while a large N is a hard program. Table I shows how the new encoding scales with hardness. The first column is the number of messages, or N, and the second column is the number of feasible match-pair subsets that correctly match every receive to a unique send. As expected, running time and memory consumption increase non-linearly with hardness.

The case where $N = 70$ represents having 70 concurrent messages in flight from 70 different threads of execution. Such a scenario is not entirely uncommon in a high performance computing application, and it appears the new encoding is able to reasonably scale to such a level of concurrency. Elaborate...

C. Typical Benchmark Programs

The prior section suggested that ... As a final note, the number of messages is not the deciding factor in hardness for the new encoding; rather, hardness is measured by the number of feasible match-sets. As is shown in the next section, some programs are easy, even if there are many messages, while other programs are hard, even though there are only a few messages.

The goal of these experiments is to measure the new encoding on several benchmark programs. MCAPI is a new interface, and to date, the authors are not aware of publicly available programs written against the interface aside from the few toy programs that come with the library distribution. As such, the benchmarks in the experiments come from a variety of sources.

- *LE* is the leader election problem and is common to benchmarking verification algorithms.
- *Router* is an algorithm to update routing tables. Each router node is in a ring and communicates only with immediate neighbors to update the tables. The program ends when all the routing tables are updated.
- *MultiM* is an extension to an program in MCAPI library distribution Figure 9. The extension adds extra iterations to the original program execution to generate longer execution trace.
- *Pktuse* is a benchmark from the MPI test suite [3]. The program creates 5 tasks—each of which randomly sends several messages to the other tasks.

The benchmark programs are intended to cover a spectrum of program properties. As before, the primary measure of hardness in the programs in not the number of messages but rather the size of the match-pair set and the number of feasible subsets. The *LE* program is the easiest program in the suite. Although it sends 620 messages, there is only a single feasible match-pair set. The programs *Router*, *MultiM*, and *Pktuse* respectively increase in hardness, which again is not related to the total number of messages but rather the total number of feasible match-sets that must be considered. For example, even though *Router* has 200 messages, it is an easier problem that *MultiM* that has 100 messages. The *Pktuse* program does have the most number of messages, 512, and in this case, the largest number of feasible match-pair sets.

TABLE II
PERFORMANCE ON SELECTED BENCHMARKS

Test Programs			Performance			
Name	# Mesg	Feasible Sets	EG(s)	MG(s)	Time (hh:mm:ss)	Memory(MB)
LE	620	1	1.49	0.051	<00:00:01	33.41
Router	200	∼6E2	0.417	0.032	00:00:02	15.03
MultiM	100	∼1E40	0.632	0.436	00:16:40	135.19
Pktuse	512	∼1E81	10.190	9.088	02:06:09	1539.90

Table II shows the results for the benchmark suite. Other than the metrics used in Table I, the time of generating the encoding and the match pairs is included in the third and fourth columns respectively. Note that the time shown in the third column includes the time in the fourth column. As before, the running time tracks hardness and not the total number of messages. The table also shows the cost of match-pair generation as it dominates the encoding time for the *Pktuse* program. Future work is to address the high-cost of match-pair generation, which the authors believe to be NP-complete. The proof is part of the suggested future work.

The benchmark suite demonstrates that a message passing program may have a large degree of non-determinism in the run time that is prohibitive to verification approaches that directly enumerate that non-determinism such as a model checker. The SMT encoding, however, pushes the problem to the SMT solver by generating the possible match-pairs and then relying on advances in SMT technology to resolve the non-determinism in a way that violates the assertion. Of course, the SMT problem itself is NP-complete, so performance is only reasonable for small problem instances. The benchmark suite suggests that problem instances with astonishingly large numbers of feasible match pair sets are able to complete in a reasonable amount of time using the new encoding in this paper; though, the time to generate the match-pairs may quickly become prohibitive.

VI. 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 [20], Vakkalanka et al. provided POE, a DPOR algorithm [10] of their dynamic verifier, ISP, for MPI programs[17]. another message passing interface standard. 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. [20] also defines *Intra-Happens-Before-Order*(*Intra-HB*), a set of partial orders 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 [20].

In [19], 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 [21]. 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 [21] 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. [22] and [4] are the approaches for MPI. [11] presented a system that uses static analysis to determine offline the topology of the communications and nodes in the input MCAPI program.

VII. 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 behaviors 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] for a set of "toy" examples under zero-buffer, our encoding is capable of capturing correct behaviors of an MCAPI program execution and providing efficient solutions. As for the experiment results, we have demonstrated that our encoding scales well for programs with large number of messages and numerous match pairs under infinite-buffer semantics. Also, zero-buffer semantics can be adapted if required.

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] API, T.M.A.R.M.: The multicore association resource management API, http://www.multicore-association.org/workgroup/mcapi.php

[2] Association, T.M.: The multicore association, http://www.multicore-association.org

[3] Benchmark, M.: Mpptest benchmark, http://www.mcs.anl.gov/research/projects/mpi/mpptest/

[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, pp. 332–347. Springer-Verlag (2012)

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

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

[18] 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)

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

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

[21] 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)

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