# 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 a 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 direct 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. Results demonstrate that the SMT encoding, restricted to zero-buffer semantics, uses fewer clauses when compared to another zero-buffer technique, and it runs faster and uses less memory. As a result the encoding scales well for programs with a lot of non-determinism in how sends and receives are matched by the runtime in the message passing.**

*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 [19]. 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) [22]. 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 to indirectly control the MCAPI runtime to verify MCAPI programs under zero-buffer semantics, so it does not verify the full behavior space of the MCAPI program as it does not address infinite-buffer semantics [21]. A key insight from the approach is its direct 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 [23]. 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 and claim the encoding supports both semantics. A careful analysis of the encoding, however, shows it to not work under infinite-buffer semantics and to miss behaviors under zero-buffer semantics [9]. Interestingly, the encoding assumes the user provides a precise set of match-pairs as input with the program trace, and it then uses those match-pairs in a non-obvious way to constraint the happens-before relation in the encoding. The work does not discuss how to generate the match-pairs, which is a non-trivial input to manually generate for large or complex program traces. An early proof claims that the problem of finding a precise set of match-pairs given a program trace is NP-complete [20].

This paper presents an SMT encoding for MCAPI program

Task 0

```
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);

0a finalize(&s);
```

Task 1

```
10 initialize(NODE_1,&v,&s);
11 e1=create_endpoint(PORT_1,&s);
12 e0=get_endpoint(NODE_0,PORT_0,&s);

13 msg_recv_i(e1,C,sizeof(C),&h3,&s);
14 wait(&h3,&size,&s,MCAPI_INF);

15 msg_send_i(e1,e0,"1",2,N,&h4,&s);
16 wait(&h4,&size,&s,MCAPI_INF);

17 finalize(&s);
```

Task 2

```
20 initialize(NODE_2,&v,&s);
21 t2=create_endpoint(PORT_2,&s);
22 t0=get_endpoint(NODE_0,PORT_0,&s);
23 t1=get_endpoint(NODE_1,PORT_1,&s);

24 msg_send_i(e2,e0,"4",2,N,&h5,&s);
25 wait(&h5,&size,&s,MCAPI_INF);

26 msg_send_i(e2,e1,"Go",3,N,&h6,&s);
27 wait(&h6,&size,&s,MCAPI_INF);

28 finalize(&s);
```

Fig. 1. An MCAPI concurrent program execution

executions that works for both zero and infinite buffer semantics. The encoding does require an input set of match-pairs as in prior work, but unlike prior work, the match-set can be over-approximated and the encoding is still sound and complete. The encoding requires fewer terms to capture all possible program behavior when compared to other proposed methods making it more efficient in the SMT solver. To address the problem of generating match-pairs, an algorithm to generate the over-approximated set is given. 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 under zero or infinite buffer semantics given the input set of potential match-pairs contains at least the precise set of match-pairs; 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 illustrating the non-determinism in runtime specification. Section 3 defines the SMT encoding using potential math-pairs. The encoding is shown to be sound and complete even under an over-approximated set of match-pairs. Section 4 provides a solution to the outstanding problem of generating feasible match pairs. Section 5 presents the experimental results that show the encoding to be efficient. Section 6 discusses related work. And Section 7 presents 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 with the first digit being the task ID, 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` 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 `13` and then sends the message *"1"* on line `15` to *e0*; and finally, task 2 sends messages *"4"* and *"Go"* on lines `24` and `26` to endpoints *e0* and *e1* respectively. The additional code (lines `08` - `09`) asserts 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, a developer might ask the question: *"What are the possible values of* `a` *and* `b` *after the scenario completes?"*

24 $\mathtt{S}_{2,4}(0, \&\mathtt{h5})$
25 $\mathtt{W}(\&\mathtt{h5})$

---

02 $\mathtt{R}_{0,2}(2, \&\mathtt{h1})$
03 $\mathtt{W}(\&\mathtt{h1})$

---

26 $\mathtt{S}_{2,6}(1, \&\mathtt{h6})$
27 $\mathtt{W}(\&\mathtt{h6})$

---

04 $\mathtt{a} = \mathtt{atoi}(\mathtt{A});$

---

13 $\mathtt{R}_{1,3}(2, \&\mathtt{h3})$
14 $\mathtt{W}(\&\mathtt{h3})$
15 $\mathtt{S}_{1,5}(0, \&\mathtt{h4})$
16 $\mathtt{W}(\&\mathtt{h4})$

---

05 $\mathtt{R}_{0,5}(1, \&\mathtt{h2})$
06 $\mathtt{W}(\&\mathtt{h2})$
07 $\mathtt{b} = \mathtt{atoi}(\mathtt{B});$
08 $\mathtt{assume}(\mathtt{b} > 0);$
09 $\mathtt{assert}(\mathtt{a} == 4);$

Fig. 2. A feasible execution traces of the MCAPI program execution in Figure 1

The intuitive trace is shown in Figure 2 using a shorthand notation for the MCAPI commands: send (denoted as `S`), receive (denoted as `R`), or wait (denoted as `W`). The shorthand notation further preserves the thread ID and line number as follows: 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 ID, `j` represents the source line number, `k` represents the destination endpoint, and `h` represents the command handler. A specific destination task ID is indicated in the notation when a trace is fully resolved, otherwise "*" notation indicates that a receive has yet to be matched to a specific send. The lines in the trace indicate the context switch where a new task executes.

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. At the end of execution the assertion on line 09 holds and no error is found.

```
24 S_{2,4}(0, &h5)
25 W(&h5)
26 S_{2,6}(1, &h6)
27 W(&h6)

13 R_{1,3}(2, &h3)
14 W(&h3)
15 S_{1,5}(0, &h4)
16 W(&h4)

02 R_{0,2}(1, &h1)
03 W(&h1)
04 a = atoi(A);
05 R_{0,5}(2, &h2)
06 W(&h2)
07 b = atoi(B);
08 assume(b > 0);
09 assert(a == 4);
```

Fig. 3. A second feasible execution traces of the MCAPI program in Figure 1

There is another feasible trace for the example shown in Figure 3 which is reachable under the infinite-buffer semantics. In this trace, 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 24 to be buffered in transit. The MCAPI specification indicates that the wait on line 25 returns once the buffer is available. That only means the message is somewhere in the MCAPI runtime under the 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 on line 15 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.

From the discussion above, it is important to consider non-determinism in the MCAPI runtime when testing or debugging an MCAPI program execution. 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 under all possible runtime behaviors). The encoding can be solved by an SMT solver such as Yices [7] or Z3 [17].

## III. SMT Encoding

The new 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. In the running example, 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 direct use of match-pairs, rather than a state-based or indirect use of match-pairs in an 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 over events.*

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 relation is derived from the program source and potential match pairs. In order to specify the constraints from the program source, each program operation is mapped to a set of variables that can be manipulated by the SMT solver.

**Definition 2** (Wait)**.** *The occurrence of a wait operation,* W*, is captured by a single variable,* $order_{\mathtt{W}}$*, that constraints when the wait occurs.*

It is not enough to represent all events as simple numbers that will be ordered in this way. Such an encoding would not allow the solver to discover what values would flow across communication primitives. Instead, some events in the trace are modeled as a set of SMT variables that record the pertinent information about the event. For example,

**Definition 3** (Send)**.** *A send operation* S*, is a four-tuple of 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 that marks where in the program the receive operation is guaranteed to be complete. The MCAPI runtime semantics allow a single wait to witness the completion of of many receives due to *the message non-overtaking property*. A wait that witnesses the completion of one or more receives is 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.*

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

The encoding requires that every receive operation have an nearest-enclosing wait as it makes match-pair decisions at the wait operation. The requirement is not a limitation of the encoding, as accessing a buffer from a receive that does not

| 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

have a nearest-enclosing wait is an error. Rather, the wait is a convenience in the encoding to mark where a receive actually takes place. The same requirement can be made for sends for correctness, but is not required for the encoding as send buffering is handled differently than receive buffering. The encoding effectively ignores wait operations for sends as will be seen.

**Definition 5** (Receive)**.** *A receive operation* $\mathtt{R}$ *is modeled by a five-tuple of 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 value; and,*
*5)* $nw_{\mathtt{R}}$*, the order of the nearest enclosing wait.*

*B. Assumptions, Assertions, and Match-pairs*

The definitions so far merely establish the pertinent information about each event in the trace as SMT variables. It is necessary to now express constraints on those variables.

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

**Definition 6** (Assumption)**.** *Every assumption* $\mathtt{A}$ *is added as an SMT assertion.*

It may seem strange to turn *assumptions* into *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. The 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* $\mathtt{P}$*,* $\neg\mathtt{P}$ *is added as an SMT assertion.*

Finally, we must express the relation in a given 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* $\mathtt{R}$ *and a send* $\mathtt{S}$ *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)* $order_{\mathtt{S}} \prec_{\mathtt{HB}} nw_{\mathtt{R}}$

The encoding is given a set of potential match-pairs over all the sends and receives in the program trace. The constraints from these match pairs are not simply unioned. If we were to do that, then we would be constraining the system such that a single receive must be paired with all possible sends in a feasible execution rather than a single send. Therefore, we combine all the constraints for a given receive with all possible sends as specified by the input match pairs into a single disjunction:

**Definition 9** (Receive Matches)**.** *For each receive* $\mathtt{R}$*, 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" nearest-enclosing waits on receives.

*C. Program Order Constraints*

The encoding thus far is missing additional constraints on the *Happens-Before* relation stemming from program order. These constraints are added 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 task, if there are multiple send operations, say $\mathtt{S}$ and $\mathtt{S}'$, from that task 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 task, if there are multiple receive operations, say $\mathtt{R}$ and $\mathtt{R}'$, from that task 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 $\mathtt{R}$ and its nearest enclosing wait $\mathtt{W}$, $order_{\mathtt{R}} \prec_{\mathtt{HB}} order_{\mathtt{W}}$.

*Step 4:* For any pair of sends $\mathtt{S}$ and $\mathtt{S}'$ 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.

| 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

The $\mathtt{M}$ 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

...
01 $order_{\mathtt{R}_{0,2}} \prec_{\mathtt{HB}} order_{\mathtt{W(\&h1)}}$
02 $order_{\mathtt{R}_{0,5}} \prec_{\mathtt{HB}} order_{\mathtt{W(\&h2)}}$
03 $order_{\mathtt{R}_{0,2}} \prec_{\mathtt{HB}} order_{\mathtt{R}_{0,5}}$
04 $order_{\mathtt{R}_{1,3}} \prec_{\mathtt{HB}} order_{\mathtt{W(\&h3)}}$
05 $orders_{\mathtt{S}_{1,5}} \prec_{\mathtt{HB}} order_{\mathtt{W(\&h4)}}$
06 $orders_{\mathtt{S}_{2,4}} \prec_{\mathtt{HB}} order_{\mathtt{W(\&h5)}}$
07 $orders_{\mathtt{S}_{2,7}} \prec_{\mathtt{HB}} order_{\mathtt{W(\&h6)}}$
08 (*assert* (> b 0))
09 (*assert* (not (= a 4)))
10 $\langle \mathrm{R}_{0,2},\mathrm{S}_{2,4}\rangle \vee \langle \mathrm{R}_{0,2},\mathrm{S}_{1,5}\rangle$
11 $\langle \mathrm{R}_{0,5},\mathrm{S}_{2,4}\rangle \vee \langle \mathrm{R}_{0,5},\mathrm{S}_{1,5}\rangle$
12 $\langle \mathrm{R}_{1,3},\mathrm{S}_{2,7}\rangle$

Fig. 6. SMT Encoding

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 further restrict behavior and enforce a zero-buffer semantics. There are two kinds of such constraints.

First, for each task, 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: $order_{\mathtt{W(\&h)}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$ where W(&h) is the nearest-enclosing wait 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 task; 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: $order_{\mathtt{W(\&h)}} \prec_{\mathtt{HB}} order_{\mathtt{S}'}$ where W(&h) is the nearest-enclosing wait that witnesses the completion of R in execution.

### E. Example

Figure 6 shows the encoding of Figure 1 as an SMT problem. We elide the basic definition of the variables discussed in Section III-A. Lines 08 through 12 give the assumptions, assertions, and match-pairs. The first seven lines reflect the program order constraints: receives happen before corresponding wait operations, receives from a common endpoint follow program order, and sends happen before corresponding wait operations. To encode the zero-buffer semantics, the constraint $event_{\mathtt{W(\&h1)}} \prec_{\mathtt{HB}} event_{\mathtt{S}_{1,5}}$ is added because the received is forced to complete before another send is issued to the runtime.

### F. Correctness

Before we can state our correctness theorem, we must define a few terms. We define our encoder as a function from programs and match pair sets to SMT problems:

**Definition 10** (Encoder)**.** *For all programs, $p$, and match pair sets $m$, let $\mathcal{SMT}(p, m)$ be our encoding as an SMT problem.*

We assume that an SMT solver can be represented as a function that takes a problem and returns a satisfying assignment of variables or an unsatisfiable flag:

**Definition 11** (SMT Solver)**.** *For all SMT problems, $s$, let $\mathcal{SOL}(s)$ be in $\sigma + \mathbb{UNSAT}$, where $\sigma$ is a satisfying assignment of variables to values.*

We assume that from a satisfying assignment to one of our SMT problems, we can derive an execution trace by observing the values given to each of the $order_e$ variables. In other words, we can view the SMT solver as returning traces and not assignments.

We assume a semantics for traces that gives their behavior as either having an assertion violation or being correct [1]:

**Definition 12** (Semantics)**.** *For all programs, $p$, and traces $t$, $\mathcal{SEM}(p, t)$ is either $\mathbb{BAD}$ or $\mathbb{OK}$.*

Given this framework, our SMT encoding technique is sound if

**Theorem 1** (Soundness)**.** *For all programs, $p$, and match pair sets, $m$, $\mathcal{SOL}(\mathcal{SMT}(p, m)) = t \Rightarrow \mathcal{SEM}(p, t) = \mathbb{BAD}$.*

Our soundness proof relies on the following lemma:

**Lemma 1.** *Any match pair $\langle \mathtt{R}, \mathtt{S}\rangle$ used in a satisfying assignment of an SMT encoding is a valid match pair and reflects an actual possible MCAPI program execution.*

*Proof.* We prove this by contradiction. First, assume that $\langle \mathtt{R}, \mathtt{S}\rangle$ is an invalid match pair (i.e. one that is not valid in an actual MCAPI execution). Second, assume that the SMT solver finds a satisfying assignment.

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 *Happens-Before* constraints encoded in the SMT problem are not satisfied.

This is a contradiction: either the SMT solver would not return an assignment or the match pair was actually valid. □

The correctness of our technique relies on completeness:

**Theorem 2** (Completeness)**.** *For all programs, $p$, and traces, $t$, $\mathcal{SEM}(p, t) = \mathbb{BAD} \Rightarrow \exists m. \mathcal{SOL}(\mathcal{SMT}(p, m)) = t$.*

We prove completeness in our extended version [12] by designing our semantics, $\mathcal{SEM}$, such that it simulates the solving of the SMT problem during its operation to ensure that the two make identical conclusions.

However, these theorems obscure an important problem: how do we know which match pair set to use? Soundness assumes we have one, while completeness merely asserts that one exists. While Section IV discusses our generation algorithm, we prove here an additional theorem that asserts

[1] In fact, our extended technical report [12] gives such a semantics.

that any conservative over-approximation of match pair sets is safe.

**Theorem 3** (Approximation)**.** *Give two match-pair sets $m$ and $m'$, $m \subseteq m' \Rightarrow \mathcal{SOL}(\mathcal{SMT}(p, m)) \sqsubseteq \mathcal{SOL}(\mathcal{SMT}(p, m'))$, where $\mathbb{UNSAT} \sqsubseteq \sigma$.*

Informally, this is true because larger match pair sets only allow *more* behavior, which means that the SMT solver has more freedom to find violations, but that all prior violations are still present. However, because of soundness, it is not possible that using a larger match pair set will discover false violations. The formal proof, in our extended version [12], relies on a match set combination operator that we prove distributes over an essential part of the semantics.

## 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.

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

output match_set;
```

Fig. 7. Pseudocode for generating over-approximated match pairs

| 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. 8. Another MCAPI concurrent program

Figure 7 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).

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

The list `list_r` groups receives by the issuing endpoint. The integer field merely records the order in which the receives are issued and increases by one on each receive. Similarly, the list `list_s` groups sends first by the destination endpoint and then by the source endpoint. Like `list_r` an index increases by one to track the issue order. As the input is a program execution trace, any sends or receives in loops already have unique identifiers.

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

Consider the program in Figure 8. The lists `list_r` and `list_s` for the example are

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

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

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. The list `list_s` in (4) reflects this partition. Receive $\mathtt{R}_{0,1}$ is the first receive operation in endpoint 0. This fact is again reflected in `list_r` in (3).

The algorithm traverses the two lists in a nested loop to generate match pairs between send and receive commands. The function `index(r)` takes the endpoint of the receive and returns the issue order of that receive in the `list_r` structure. Similarly, the function `index(s)` takes the destination and source endpoints in the send and returns the issue order of that send in the `list_s` structure. These indexes essentially capture message non-overtaking.

The criteria to generate a match pair first requires the send and receive to be compatible (check 1), consistent with message non-overtaking (check 2), and message non-overtaking does preclude the match (check 3). A match is precluded message non-overtaking when a receive cannot possibly match a send because by the time the program issues the receive, the send must have already been matched somewhere else. The function `count` gives the number of sends to a specific destination or the number of sends to a specific source and destination. As long as a receive is issued early enough to still match the send given the message non-overtaking rule, then the match is possible.

In our concrete example, $\texttt{R}_{0,1}$ is matched with $\texttt{S}_{1,1}$ or $\texttt{S}_{2,1}$, but it cannot be matched with $\texttt{S}_{1,3}$ since the second rule is not satisfied such that the order of $\texttt{R}_{0,1}$ is less than the order of $\texttt{S}_{1,3}$ (i.e., $\texttt{S}_{1,3}$ would have to overtake $\texttt{S}_{1,1}$ to satisfy the rule). The match between $\texttt{R}_{0,4}$ and $\texttt{S}_{1,1}$ is also precluded by check 3 as $\texttt{S}_{1,1}$ must have already matched an earlier receive by message non-overtaking.

The generated set of match pairs for our example in Figure 8 is over-approximated by the algorithm because it includes pairs that cannot exist in any feasible execution. For example, the match pair ($\texttt{S}_{2,1}$ $\texttt{R}_{0,4}$) is not feasible because it is not possible to order $\texttt{S}_{1,3}$ before $\texttt{R}_{0,2}$ since $\texttt{R}_{1,2}$ can only match with $\texttt{S}_{0,3}$ that must occur after $\texttt{R}_{0,2}$. Fortunately, a satisfying solution is only possible using feasible match pairs. Non-feasible match-pairs merely result in extra clauses in the encoding and potentially slow down the SMT solver.

The complexity of the algorithm is quadratic. Traversing the tasks to initialize the lists is $O(N)$, 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. As $m+n \leq N$, 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 ([17]) 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 the MCA provided reference solution with fixed input. In other words, the only non-determinism in the programs is that allowed by the MCAPI specification. As such, 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 our best knowledge, the current most effective SMT encoding for verification of message passing program traces is the order-based encoding that describes the happens-before relation directly in the encoding and is only functional for zero-buffer semantics in its current form [9]. Although the tool to generate the encoding is not publicly available, the authors of the order-based encoding 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. The result provides a bound on expected cost for analysis given the the message passing behavior in a program. It is expected that the analysis of any program with fewer than 70! possible choices of feasible match-pair resolutions will complete in a reasonable amount of time.

### C. Typical Benchmark Programs

The results in the prior section suggest that 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-pair sets. This section further explores the observation to show that 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 8. 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 [20].

The benchmark suite demonstrates that a message passing program may have a large degree of non-determinism in the runtime that is prohibitive to verification approaches that directly enumerate 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 the C language for the MCAPI interface [16]. This model correctly captures the behavior of the interface and can be applied to model checking C programs that use the API. The work is a direct application of model checking and directly enumerates the non-determinism in the runtime to construct an exhaustive proof. The SMT encoding in this paper pushes that complexity to the SMT solver and leverages recent advances in SMT technology to find a satisfying assignment.

Vakkalanka *et al.* define POE, a dynamic partial order algorithm for MPI programs [22], [10]. As MPI is similar to MCAPI, though more expressive, the algorithm is relevant to MCAPI verification [18]. Intuitively, the POE algorithm, as a model checker, enumerates non-determinism in the MPI specification under zero-buffer semantics with the goal to detect deadlock. The algorithm is not suitable to infinite-buffer semantics as it misses behavior. The relatively efficiency, in practice, of the model checker enumerating non-determinism versus the SMT solver is an open question for the POE work.

Key to the POE algorithm is the notion of a *intra-happens-before-order*. The order is a partial order over MPI events that generalizes to any control flow path through the program. The SMT encoding in this builds the happens-before relation for a given control flow path and does not reason directly over all control flow paths in the program. A secondary analysis is needed to find program inputs sufficient to enumerate all reachable control flow paths in order to apply the SMT encoding to complete program verification.

Sharma *et al.* present an dynamic model checker for MCAPI programs built on top of the MCA provided MCAPI runtime [21]. MCC systematically enumerates all non-determinism in the MCAPI runtime under zero-buffer semantics. It employs a novel dynamic partial order reduction to avoid enumerating redundant message orders. This work claims SMT technology is more efficient in practice resolving non-determinism in a away to violate correctness properties.

Wang *et al.* present an SMT encoding for shared memory semantics for a given input trace from a multi-threaded program [23]. As mentioned previously, the program is partitioned into several concurrent trace programs, and the encoding for each program is verified using SMT technology. Elwakil *et al.* extend the encoding to message passing programs using the MCAPI semantics [8], [9]. The comparison to the encoding in this work is already discussed previously.

There is a rich body of literature for SMT/SAT based Bounded Model Checking. Burckhardt *et al.* exhaustively check all executions of a test program by translating the program implementation into SAT formulas [5]. The approach relies on counter-examples from the solvers the refine the encoding. The SMT encoding in this work is able to directly resolve the match-pair set over-approximation directly without needing to check a counter-example.

Dubrovin *et al.* give a method to translate an asynchronous system into a transition formula over three partial order semantics [6]. The encoding adds constraints to compress the search space and decrease the bound on the program unwinding. The encoding in this paper operates on a program execution and does not need to resolve a bound.

Kahlon *et al.* presented a partial order reduction, *MPOR*, that operates in the bounded model checking space [13]. It guarantees that exactly one execution is calculated per each Mazurkiewicz trace to reduce the search space. It would be interesting to see if MPOR is able to extend to message passing semantics. Other work in bounded model checking explores heap-manipulating programs and challenges in sequential systems code [15], [14].

The application of static analysis is another interesting thread of research to test or debug message passing programs with some work in the MPI domain [24], [4], [11]. The work is important as it lays the foundation for refining match-pair sets to only include those that cannot be statically pruned.

## VII. Conclusions and Future Work

The paper presents an SMT encoding of an MCAPI program execution that uses match pairs directly rather than the state-based or order-based encoding in the prior work. The encoding is generated from a given execution trace and a set of potential match pairs that can be over-approximated. The encoding takes extra care in the forming the SMT problem to preclude bogus match pairs in any over-approximation of the match pair input set. Critically, the encoding is the first to correctly capture the non-deterministic behaviors of an MCAPI program execution under infinite-buffer semantics.

The paper further defines an algorithm with $O(N^2)$ time complexity to over-approximate the true set of match pairs, where $N$ is the total number of code lines of the program. A comparison to prior work, [9], for a set of "toy" examples under zero-buffer semanics shows the new encoding capable and efficient in capturing correct behaviors of an MCAPI program execution. Experiments further show that the encoding scales to programs with significant levels of non-determinism in how sends are match to receives.

The results show that a large match-pair set does affect the runtime performance of the encoding in the SMT problem even if the encoding is sound under an over-approximation. Future work explores new methods for generating a much more precise set of match pairs. The encoding is dependent on an input execution trace of the program. Future work explores integrating the encoding into a model checker. The model checker generates a program trace that is encoded and verified. The result is then used to inform the model checker as to where it needs to backtrack to generate a new execution trace. The goal is to use the trace verification to construct a better partial order reduction in the model checker.

## 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] Huang, Y., Mercer, E., McCarthy, J.: Proving mcapi executions are correct using smt (extended), http://students.cs.byu.edu/ỹhuang2/downloads/paper.pdf

[13] 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)

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

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

[16] 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)

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

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

[19] 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)

[20] Sharma, S.: Private conversation on active research.

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

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

[23] 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)

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

## Appendix

**Definition 13.** ***VAMPI****: Verifying User Provided Assertion Violation in an MCAPI program.*
*INSTANCE: A set of constants D, a set of variables X, a finite set processes P, and a set of partial orders* $\prec$ *over send, receive and assert operations.*
*QUESTION: Is there a coherent schedule S for the operations of P that violates the user provided assertions?*

*Proof.* □

**SAT:** $U \equiv \{u_1, u_2, ..., u_m\}$
$C \equiv \{c_1, c_2, ..., c_n\}$
$Q \equiv \{c_1 \wedge c_2 \wedge ... \wedge c_n\}$

**VAMPI:** $P \equiv \{p_1, p_2, p_3, p_{u_1}, ..., p_{u_m}\}$
$X \equiv \{x_1, ..., x_m, x_{c_1+1}, ..., x_{c_1+m}, ..., x_{c_n+1}, ..., x_{c_n+m}\}$
$D \equiv \{d_{u_1}, ..., d_{u_m}, \bar{d}_{u_1}, ..., \bar{d}_{u_m}, d_{c_1}, ..., d_{c_n}, \bar{d}_{c_1}, ..., \bar{d}_{c_n}\}$
$D_l \equiv$ An ordered list of values $d_c$ for each literal $l$, such that $l \in c$
$\bar{D_l} \equiv$ An ordered list of values $\bar{d_c}$ for each literal $l$, such that $l \notin c$

| $p_1$ | $p_2$ | $p_{u_i}$ | $p_3$ |
|---|---|---|---|
| $S_{1,1}(d_{u_1}, p_{u_1}, r_1)$ | $S_{2,1}(\bar{d}_{u_1}, p_{u_1}, r'_1)$ | $R_{u_i,1}(x_i, *, r_i)$ | $R_{3,1}(x_{c_1+1}, *, r_1)$ |
| ... | ... | $R_{u_i,2}(x_i, *, r'_i)$ | ... |
| $S_{1,m}(d_{u_m}, p_{u_m}, r_m)$ | $S_{2,m}(\bar{d}_{u_m}, p_{u_m}, r'_m)$ | if($x_i = \bar{d}_{u_i}$){ | $R_{3,m}(x_{c_1+m}, *, r_m)$ |
| | | $S_{u_i,4}(D_i[1], p_3, r_{i_1})$ | ... |
| | | ... | $R_{3,(n-1)*m+1}(x_{c_n+1}, *, r_{(n-1)*m+1})$ |
| | | $S_{u_i,3+n}(D_i[n], p_3, r_{i_n})\}$ | ... |
| | | if($x_i = d_{u_i}$){ | $R_{3,n*m}(x_{c_n+m}, *, r_{n*m})$ |
| | | $S_{u_i,5+n}(\bar{D_i}[1], p_3, r'_{i_1})$ | $assert(\bigvee_{1 \leq i \leq m}(x_{c_1+i} = d_{c_1}))$ |
| | | ... | ... |
| | | $S_{u_i,4+2n}(\bar{D_i}[n], p_3, r'_{i_n})\}$ | $assert(\bigvee_{1 \leq i \leq m}(x_{c_n+i} = d_{c_n}))$ |

Fig. 9. General SAT to VAMPI reduction