# Proving MCAPI Executions are Correct
## Applying SMT Technology to Message Passing

Yu Huang, Er ic Mercer, and Jay McCarthy *

Brigham Young University
{yuHuang,egm,jay}@byu.edu

**Abstract.** Asynchronous message passing for C is an important paradigm in writing applications for embedded heterogeneous multicore systems. The Multicore Association (MCA), an industry consortium promoting multicore technology, is working to standardize message passing into a single API, MCAPI, for bare metal implementation and portability across platforms. Correctness in such an API is difficult to reason about manually, and testing against reference solutions is equally difficult as reference solutions implement an unknown set of allowed behaviors, and programmers have no way to directly control API internals to expose or reproduce errors. This paper encodes an MCAPI execution as an SMT problem, which if satisfiable, yields a feasible execution schedule on the same trace, such that it resolves non-determinism in the MCAPI runtime in such 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 use of match-pairs (potential send and receive couplings) to model the MCAPI execution in the SMT problem. 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.

**Keywords:** Abstraction, refinement, SMT, message passing

## 1 Introduction

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

One specification that the MCA has released is the Multicore Association Communications API (MCAPI). The specification defines basic data type, data structures, and functions that can be used to perform simple message passing

* Special thanks to Christopher Fischer, formally at BYU and now at Amazon

operation between different computing entities within a device. Messages can be passed across persistent channels that force an ordering of the messages, or they can be passed to specific *endpoints* within the system. The specification places few ordering constraints on messages passed from one endpoint to another. This freedom introduces into the system possibilities for non-deterministic behavior in the way the messages arrive at their destination.

This non-deterministic behavior in MCAPI programs is difficult for current tools to test or debug. Chao Wang provided a description of a verification method for multithreaded software that uses shared memory as a means of synchronization that provides a marked improvement over a comparable tool that uses concrete execution [4]. Unfortunately, Wang's method for shared memory system does not work on the message passing system. Sharma et al. created a method of using concrete execution to verify MCAPI programs, but it was later discovered that this method does not completely explore the entire execution space of certain kinds of MCAPI programs [3]. Elwakil et al. also defined a method of representing MCAPI program executions as satisfiability problems building on the method of [4] adapting it to message passing, but this method does not correctly model all kinds of MCAPI program executions [2].

Either the existing technique can not correctly encodes the program, or they can not provide efficient method for testing or debugging. Due to the difficulty and defect described above, we present an encoding for MCAPI program executions that uses match-pairs from [3] in a new SMT encoding that requires fewer terms than [2] but includes all possible program executions unlike [2]. In other words, the encoding itself provides efficiency. Most importantly, this method guarantees of finding all hidden errors by encoding all possible execution traces of a concurrent program into an SMT problem.

Our main contributions in this paper include:

1. A correct and efficient SMT encoding of an MCAPI program execution; and
2. An $O(n^2)$ algorithm to generate an over-approximation of possible match-pairs.

The rest of the paper is organized as follows: Section 2 presents an MCAPI program, where two outcomes of execution traces can be found. Section 3 defines an SMT encoding for an MCAPI program execution that captures non-determinism in message order by defining the set of possible match pairs between sends and receives on a common endpoint. Section 4 solves the outstanding problem in other encodings of generating feasible match-pairs to use in the encoding. We present a $O(n^2)$ algorithm that over-approximates the true match set. We then use our encoding in a framework that provides a *CEGAR*[1] loop to prune infeasible match pairs from the initial set until we witness a true error or conclude no error is possible. Section 5 presents the experimental results that show our encoding to be correct and efficient, as other existing encodings both omit critical program behaviors and require more SMT clauses. Section 6 proves that our

[1] The *CEGAR* loop is defined as a **C**ounter **E**xample **G**uided **A**bstraction **R**efinement.

encoding is correct by first giving formal semantics to an MCAPI program execution. Second, defining the encoding of a single program execution in an SMT program; and third, generalizing that encoding to set of match pairs rather than those generated in the specific program encoding. Section 7 discusses related work, and Section 8 presents our conclusions.

## 2 Example

It is a challenge to explain intended behavior in simple scenarios consisting of a handful of calls when dealing with concurrency. Consider the MCAPI program 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 - 10`) 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?"*

| Task 0 | Task 1 | Task 2 |
|---|---|---|
| 00 initialize(NODE_0,&v,&s);<br>01 e0 = create_endpoint(PORT_0,&s);<br><br>02 msg_recv_i(e0,A,sizeof(A),&rcvA,&s);<br>03 wait(&rcvA,&size,&s,MCAPI_INF);<br>04 a = atoi(A);<br><br>05 msg_recv_i(e0,B,sizeof(B),&rcvB,&s);<br>06 wait(&rcvB,&size,&status,MCAPI_INF);<br>07 b = atoi(B);<br><br>08 if(b > 0);<br>09 assert(a == 4);<br><br>10 finalize(&s); | 00 initialize(NODE_1,&v,&s);<br>01 e1 = create_endpoint(PORT_1,&s);<br>02 e0 = get_endpoint(NODE_0,PORT_0,&s);<br><br>03 msg_recv_i(e1,C,sizeof(C),&rcvC,&s);<br>04 wait(&rcvC,&size,&s,MCAPI_INF);<br><br>05 msg_send_i(e1,e0,"1",2,N,&snd3,&s);<br>06 wait(&snd3,&size,&s,MCAPI_INF);<br><br>07 finalize(&s); | 00 initialize(NODE_2,&v,&s);<br>01 t2 = create_endpoint(PORT_2,&s);<br>02 t0 = get_endpoint(NODE_0,PORT_0,&s);<br>03 t1 = get_endpoint(NODE_1,PORT_1,&s);<br><br>04 msg_send_i(e2,e0,"4",2,N,&snd1,&s);<br>05 wait(&snd1,&size,&status,MCAPI_INF);<br><br>06 msg_send_i(e2,e1,"Go",3,N,&snd2,&s);<br>07 wait(&snd2,&size,&status,MCAPI_INF);<br><br>08 finalize(&s); |

**Fig. 1.** An MCAPI concurrent program

The intuitive trace is presented in the first three columns of Figure 2. Note that the fist column contains the task number in Figure 1. The second column presents the commands with line number shown in Figure 1. Also, we use the shorthand for each command of send, receive or wait in the third column. The 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 refers to the

if statement on line 08 of task 0 in Figure 1, such that it is the "check" that the program follows the same control flow. At the end of execution the assertion on line 09 of task 0 holds and no error is found. Such intuition is a valid program execution. Consider another scenario in the fourth and fifth column of Figure 2. Note that we use the shorthand for each command of send, receive and wait in the fifth column because of space limit. The variable $a$ contains 1 instead of 4, since the message *"1"* is sent to *e0* after sending the message *"Go"* to *e1*. The send on line 04 of task 2 may be delayed 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; it does not mean the message is delivered. As such, it is possible for the message to be delayed 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 valid program execution that results in an assertion failure at line 09 in Figure 1. From the discussion above, a single valid sequential execution is not sufficient to prove correctness because of concurrency as other valid traces may exist. The next section presents our method of SMT encoding that follow the same control flow path through an MCAPI program. The encoding can be solved by an SMT solver such that the non-deterministic behavior of the program can be resolved.

| Trace 1 | | | Trace 2 | |
|---|---|---|---|---|
| Task | Command | Shorthand | Task | Command |
| 2 | 04 msg_send_i(e2, e0, "4", 2, N, &snd1, &s); | snd1 | 2 | 04 snd1 |
| 2 | 05 wait(&snd1, &size, &status, MCAPI_INF); | wait_snd1 | 2 | 05 wait_snd1 |
| 2 | 06 msg_send_i(e2, e1, "Go", 3, N, &snd2, &s); | snd2 | 2 | 06 snd2 |
| 2 | 07 wait(&snd2, &size, &status, MCAPI_INF); | wait_snd2 | 2 | 07 wait_snd2 |
| 0 | 02 msg_recv_i(e0, A, sizeof(A), &rcvA, &s); | rcvA | 1 | 03 rcvC |
| 0 | 03 wait(&rcvA, &size, &s, MCAPI_INF); | wait_rcvA | 1 | 04 wait_rcvC |
| 0 | 04 a = atoi(A); | | 1 | 05 snd3 |
| 1 | 03 msg_recv_i(e1, C, sizeof(C), &rcvC, &s); | rcvC | 1 | 06 wait_snd3 |
| 1 | 04 wait(&rcvC, &size, &s, MCAPI_INF); | wait_rcvC | 0 | 02 rcvA |
| 1 | 05 msg_send_i(e1, e0, "1", 2, N, &snd3, &s); | snd3 | 0 | 03 wait_rcvA |
| 1 | 06 wait(&snd3, &size, &status, MCAPI_INF); | wait_snd3 | 0 | 04 a = atoi(A); |
| 0 | 05 msg_recv_i(e0, B, sizeof(B), &rcvB, &s); | rcvB | 0 | 05 rcvB |
| 0 | 06 wait(&rcvB, &size, &status, MCAPI_INF); | wait_rcvB | 0 | 06 wait_rcvB |
| 0 | 07 b = atoi(B); | | 0 | 07 b = atoi(B); |
| 0 | 08 assume(b > 0); | | 0 | 08 assume(b > 0); |
| 0 | 09 assert(a == 4); | | 0 | 09 assert(a == 4); |

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

## 3 SMT Model

The novelty of the SMT encoding in this paper is its use of match pairs rather than the state-based or order-based encoding of prior work [2, 1]. A match pair is the coupling of a receive to a particular send. Figure 3 (a) is the set of possible match pairs for the program in Figure 1. Note that the definitions of `rcvA`, `rcvB`, etc. are not shown because they are are not novel to our solution, but they are consistent with the commands in Figure 1, which is shown in Figure 2. The set admits, for example, that `rcvA` can be matched with either `snd1` or `snd2`. The SMT encoding in this paper asks the SMT solver to resolve the match pairs for the system.

Other than the match pairs as input, our SMT encoding needs various data structures and support functions. Before presenting the SMT encoding for a specific example, we need to explain the data structures and support functions in detail. We give the definitions in the Yices input language.

$$\begin{array}{l}(\textbf{define}\ \text{HB} :: (\rightarrow\ a{::}int\ b{::}int \\ \quad (\textbf{subtype}\ (c{::}bool)\ (\text{ite}\ (<\ a\ b)\ (=\ c\ \text{true})\ (=\ c\ \text{false})))))\end{array} \tag{1}$$

The HB function in equation (1) represents a happens-before relationship between two events. It takes two events $a$ and $b$ as parameters, such that $a, b \in \mathbb{N}$. In the SMT encoding, every location is assigned an event, and HB orders those locations (i.e., program order, etc.). This function creates a constraint such that the first event must be less than the second event, indicating that the first event occurs before the second event. For example, action $a$ happens before $b$ if and only if $event_a$ for $a$ is greater than $event_b$ for $b$. This function is not only used to assert the program order of statements within the same thread, but also to assert, for any matched pair, that the send operation occurs before the receive.

$$\begin{array}{l}(\textbf{define-type}\ R\ (\textbf{record}\ M{::}int\ e{::}int\ var{::}int\ event{::}int)) \\ (\textbf{define-type}\ S\ (\textbf{record}\ MP{::}int\ ID{::}int\ e{::}int\ value{::}int\ event{::}int))\end{array} \tag{2}$$

We define two records in equation (2) that are essential to encoding the SMT problem. The first record $R$ defined above represents a receive action such that $M$, $e$, $var$, $event \in \mathbb{N}$. $M$ represents a free variable that is assigned the $event$ field of the matched record $S$. The endpoint $e$ is the end point of the receive operation, $var$ represents a variable which is assigned a value received by the receive operation, and $event$ is the event term which indicates the program order for the receive action associated with the record. The second record $S$ defined above is a send action such that $MP$, $ID$, $e$, $value$, $event \in \mathbb{N}$. The fields $e$ and $event$, as those in $R$, are the end point of the send operation and event term, respectively. The field $value$ is the value being sent. $MP$ is another free variable used to assign match pairs, and it is assigned an event term of a receive operation if the receive operation is to be matched to the send operation. $ID$ is a unique number that identifies each send operation. Note that the integer value assigned to $event$ indicates the program order. Also, we keep record of two

match variables, $M$ and $MP$, for different purposes. $M$ is used for preventing two receive operations from being matched with the same send operation, which is defined in function NE. $MP$ is used for ordering two receive operations with respect to the program order of two matched send operations. For resolving the SMT encoding by an SMT solver, the send definition has the match pair and event term unconstrained, and the receive definition leaves the event term, match pair and the receive variable unconstrained.

```
(define MATCH :: (→ r::R s::S (subtype (c::bool) (ite
  (and (= (select r e) (select s e))
    (= (select r var) (select s value))
    (HB (select s event) (select r event))
    (= (select r M) (select s ID))
    (= (select s MP) (select r event)))
  (= c true)
  (= c false)))))                                    (3)
```

The MATCH function in equation (3) takes tuple $R$ and $S$ as parameters. It creates a number of constraints that represent the pairing of the specified send and receive operations. This function asserts that the destination end point of the send operation is the same as the end point used by the receive operation, that the value sent by the send operation is the same value received by the receive operation, that the send operation occurs before the receive operation, that the "match" value in the receive tuple is equal to the "ID" value in the send tuple, and that the "MP" value in the send tuple is equal to the "event" value in the receive tuple.

```
(define NE :: (→ r1::R r2::R (subtype (c::bool) (ite
  (= (select r1 M) (select r2 M))
  (= c false)
  (= c true)))))                                     (4)
```

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

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

The SMT encoding as defined in the following equation,

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

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

Given an MCAPI program as input with a set of match pairs, we can now generate our encoding in the following steps. First, for each send or receive, add an $S$ or $R$ definition. Second, use an variable for such definition following by a string ".event" to represent the field event of the variable. Third, for each location represented by the field event in each thread, add an `HB` relation on the associated event for program order. Finally, for each match pair, add the `MATCH` relation and the associated `NE` relation.

```
(rcvA snd1)
(rcvA snd3)

(rcvB snd1)
(rcvB snd3)

(rcvC snd2)
```

(a)

```
defs is not shown;

constraints;
(HB rcvA.event wait_rcvA.event)
(HB wait_rcvA.event rcvB.event)
(HB rcvB.event wait_rcvB.event)
(HB wait_rcvB.event assume.event)
(HB assume.event assert.event)
(HB rcvC.event wait_rcvC.event)
(HB wait_rcvC.event snd3.event)
(HB snd3.event wait_snd3.event)
(HB snd1.event wait_snd1.event)
(HB wait_snd1.event snd2.event)
(HB snd2.event wait_snd2.event)
(assert (> b 0))
(assert (not (= a 4)))

match;
(or (MATCH rcvA snd1)
    (MATCH rcvA snd3))
(or (MATCH rcvB snd1)
    (MATCH rcvB snd3))
(MATCH rcvC snd2)
(NE rcvA rcvB)
```

(b)

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

Figure 3 (b) is a concrete example of the stylized SMT encoding for our running program in Figure 1. The fields *constraints* and *match* work as we defined above. In particular, by encoding function HB, `rcvA` happens before the

wait command `wait_rcvA` given the "event" for each command as parameters in Figure 3 (b). Any assignment to event variables by the SMT solver must comply with program order constraints. Also, the `MATCH` function encodes the relation such that `rcvA` is matched with `snd3` in Figure 3 (b). Note that the `MATCH` clauses sets the set of match pairs in Figure 3 (a) as input. The `NE` term precludes `rcvA` and `rcvB` from matching to the same send. Other than the support functions, the first assert works as the assume (line `08` at task 0) in Figure 1, such that it prevents the SMT solver from finding solutions that are not consistent with control flow, so any solution that has $b \leq 0$ is infeasible. Finally, the second assert is negated as the goal is to find schedules that violate the property. In our example, the interest is in solutions that resolve in $a \neq 4$.

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

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

to return the solution of an SMT problem, such that **SAT** represents an satisfiable solution that finds an invalid execution of the MCAPI program, and **UNSAT** represents an unsatisfiable solution that guarantees all program executions implied by the encoding are valid. Note that **UNSAT** and **SAT** are ordered such that **UNSAT** $<$ **SAT**.

The SMT encoding defined above is used to capture the non-deterministic behavior of an MCAPI program 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. The SMT encoding we present in Figure 3 (b) captures both executions, since the set of match pairs in Figure 3 (a) is a complete set where all matches that can occur in the real execution for our running example in Figure 1 are included, and all matches that can not occur in the real execution are not included. Given a different set of match pairs, however, the MCAPI program behavior will be captured differently through a new encoding. Furthermore, the following theorem states that a larger set of match pairs over-approximates the true behavior of the system. 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, then that error may or may not be a real error.

**Theorem 1.** *For two SMT problems,* $smt_\alpha = (defs\ constraints\ match_\alpha)$ *and* $smt_\beta = (defs\ constraints\ match_\beta)$*, if* $set(match_\alpha) \subseteq set(match_\beta)$*,*

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

*where* $set(match)$ *represents the input set of match pairs for the* `MATCH` *clauses in the field match.*

**Proof Sketch** Consider the MCAPI program in Figure 1 as an example. Figure 4 (a) and (b) are the stylized SMT encodings for our running example in Figure 1, such that Figure 4 (a) encodes trace 1 in Figure 2 and Figure 4 (b)

```
defs is not shown;

constraints;
(HB rcvA.event wait_rcvA.event)
(HB wait_rcvA.event rcvB.event)
(HB rcvB.event wait_rcvB.event)
(HB wait_rcvB.event assume.event)
(HB assume.event assert.event)
(HB rcvC.event wait_rcvC.event)
(HB wait_rcvC.event snd3.event)
(HB snd3.event wait_snd3.event)
(HB snd1.event wait_snd1.event)
(HB wait_snd1.event snd2.event)
(HB snd2.event wait_snd2.event)
(assert (> b 0))
(assert (not (= a 4)))

match;
(MATCH rcvA snd1)
(MATCH rcvB snd3)
(MATCH rcvC snd2)
```

(a)

```
defs is not shown;

constraints;
(HB rcvA.event wait_rcvA.event)
(HB wait_rcvA.event rcvB.event)
(HB rcvB.event wait_rcvB.event)
(HB wait_rcvB.event assume.event)
(HB assume.event assert.event)
(HB rcvC.event wait_rcvC.event)
(HB wait_rcvC.event snd3.event)
(HB snd3.event wait_snd3.event)
(HB snd1.event wait_snd1.event)
(HB wait_snd1.event snd2.event)
(HB snd2.event wait_snd2.event)
(assert (> b 0))
(assert (not (= a 4)))

match;
(MATCH rcvA snd3)
(MATCH rcvB snd1)
(MATCH rcvC snd2)
```

(b)

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

encodes trace 2 in Figure 2. By solving the encodings in Figure 4 (a) and (b) for trace 1 and 2 in Figure 2 respectively, we get an unsatisfiable solution for Figure 4 (a), and a satisfiable solution for Figure 4 (b). As we discussed in Section 2, trace 1 is a valid execution without failure of the assertion, and trace 2 is the one that fails the assertion. Compare both encodings with that in Figure 3 (b), we find that the fields *defs* and *constraints* are identical except for the `MATCH` clauses. In particular, the input set of match pairs of either Figure 4 (a) or (b) is the subset of that in Figure 3 (b). As discussed above, the encoding in Figure 3 (b) captures the non-deterministic behavior of our running program in Figure 1, which encodes trace 1 and 2 in Figure 2 into one single SMT problem. Thus, an SMT solver will return a satisfiable solution for the encoding in Figure 3 (b). Thus, the answer of the encoding in Figure 3 (b) is great or equal to that in either Figure 4 (a) or (b). The formal proof can be given similarly, which is shown in our long version of paperwork at (url??????????).

## 4 Over-approximated Match Pairs Generation

The exact set of match pairs can be generated by simulating program execution and using a depth-first search to enumerate non-determinism arising from concurrent sends and receives. Such an effort, however, solves the very problem we are trying to solve because if you simulate the entire program exploring all non-determinism, then you can also verify all program executions for property violations. In this section, we present an algorithm that does not require an

exhaustive enumeration of program behavior in simulation. Our algorithm over-approximates the set of match pair, such that match pairs that can exist in the real execution are all included, and some bogus match pairs that can not exist in the execution may or may not be included.

Other than simulating all possible executions of the program, the algorithm that generates the over-approximated match pairs is basically a strategy of matching each pair of the send and receive commands with common endpoint and pruning some bogus matches that can not exist in the real executions. Our algorithm uses two lists that keeps record for all send and receive commands.

$$\begin{array}{l}(e_0 \ \rightarrow \ ((0,\ rcv_{00}),\ (1,\ rcv_{01}),\ \ldots)) \\ (e_1 \ \rightarrow \ ((0,\ rcv_{10}),\ (1,\ rcv_{11}),\ \ldots)) \\ \ldots \\ (e_n \ \rightarrow \ ((0,\ rcv_{n0}),\ (1,\ rcv_{n1}),\ \ldots))\end{array} \tag{5}$$

The list in (5) keeps record of the receive commands uniquely identified by the endpoint $e_x$, where $x$ is the number of the endpoint. The integer in the first field of each pair represents the index of the receive command for each endpoint. We use $I_r$ to denote the index. Note that the receive command with smaller $I_r$ should be served first in the program runtime. The receive command $rcv_{xxx}$ in the second field of the list is defined as a record $R$ in equation 2 of the previous section. The number $xxx$ represents the command number.

$$\begin{array}{l}\text{“}dst\text{”} \quad \text{“}src\text{”} \qquad\qquad\qquad\qquad\qquad\qquad \text{“}src\text{”} \\ (e_0 \ \rightarrow \ ((e_1 \ \rightarrow \ ((0,\ snd_{100}),\ (1,\ snd_{101}),\ \ldots),\ (e_2 \ \rightarrow \ (\ldots),\ \ldots)))) \\ (e_1 \ \rightarrow \ ((e_0 \ \rightarrow \ ((0,\ snd_{010}),\ (1,\ snd_{011}),\ \ldots),\ (e_2 \ \rightarrow \ (\ldots),\ \ldots)))) \\ \ldots \\ (e_n \ \rightarrow \ ((e_0 \ \rightarrow \ ((0,\ snd_{0n0}),\ (1,\ snd_{0n1}),\ \ldots),\ (e_1 \ \rightarrow \ (\ldots),\ \ldots))))\end{array} \tag{6}$$

Also, we keep record of the send commands using the list in (6) that is uniquely identified by the destination endpoint $e_x$. Each sublist for the destination endpoint is uniquely identified by the source endpoint $e_y$. The integers $x$ and $y$ in the endpoints represent the number of the destination and source endpoints, respectively. The strings “dst” and “src” indicate the endpoint is either a destination or a source, respectively. Similarly, the integer in the first field of each pair represents the index of the send command. We use $I_s$ to denote the index. Note that $I_s$ is ordered within each sublist. The send command $snd_{xxx}$ in the second field of the list is defined as a record $S$ in equation 2 of the previous section. The number $xxx$ represents the command number.

Observe that the sends defined with an identical destination endpoint and an identical source endpoint are processed in a FIFO way in the program runtime. Also, the receive ordered first should be served first compared with those from an identical endpoint. Because of the fact of FIFO processing, a list of receives in (5) and a sublist of sends in (6) with an identical endpoint can be determined how many matches there are from the sends to the receives by applying the following criteria:

For each receive command `rcv` in the list of (5) with index $I_r$ and each send command `snd` in the sublist of (6) with index $I_s$, if `rcv` is matched with `snd`, we have,

1. the destination endpoint of `snd` is identical to the endpoint of `rcv`;
2. $I_r \geq I_s$;
3. and $I_r \leq I_s + (N_s - n_s)$.

where $N_s$ is the number of the sends in the program, and $n_s$ is the number of the sends in the sublist where `snd` is resident.

After given the criteria above, we can now provide our algorithm and analyze the time complexity. Intuitively, we use the criteria above to prune each "bogus" match pair from a send to a receive if at least one of the rules is not satisfied. Our method works as follows. First, we linearly traverse each task of the program and store each receive and send command to the lists in (5) and (6), respectively. This step takes $O(N)$ to complete, where $N$ is the number of commands in the program. Second, we pick up each list of receives in (5) and each sublist in (6), and then apply the criteria above to each send command `snd` and receive command `rcv`. If all rules of the criteria are satisfied for `snd` and `rcv`, we build a match pair (`snd`, `rcv`) in the result set. This step takes $O(N^2)$ to complete since each pair of a send and a receive should be determined by the criteria. Once step 2 is completed, we build a set of match pairs, where all match pairs that can exist in the program runtime are included, and those can not exist in the real execution may or may not be included. Totally, the algorithm takes $O(N + N^2) \approx O(N^2)$ to complete.

By setting the over-approximated set of match pairs as input to our solution, we then validate satisfying assignments from the SMT solver to see if they use only feasible match pairs. Such validation is accomplished by creating an "oracle" - a model of the MCAPI library with full operational semantics. The model is able to detect bad match pairs in a given program trace. The operational semantics of the model are presented in the long version of paperwork at (url??????????). The bad match pairs can be omitted from the SMT encoding and the SMT solver re-run to see if another solution exists. The framework provides a *CEGAR* loop for MCAPI trace verification.

## 5 Result

To evaluate the reliability and efficiency of our system, we compare our encoding with the work presented in [2]. The encoding in [2] is not correct and misses valid program executions allowed by the MCAPI runtime. For example, it only finds one of the two possible traces from the example program used in this paper because the encoding does not allow an non-blocking send operations from different end points to be reordered. As such, the comparison in this paper is not exactly valid. Nevertheless, the encoding in this paper not only captures all non-deterministic behavior in the runtime, it is also a more efficient encoding.

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

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

For the program in Figure 1, we encodes 17 clauses for modeling the problem, omitting the definition of variables at the beginning of the encoding. Based on the encoding rules in [2], however, there are 47 clauses for the same program, omitting the definitions as well. Table 1 shows the clause number for four parts of each encoding for the MCAPI program in Figure 1. In particular, the program order covers most parts of each encoding. The program order and matches of the encoding in [2] are 2-4 times larger than those in our encoding. Besides, the encoding in [2] needs some extra clauses, including the variable assignments and assertion integration, that expand the entire encoding.

| | Example in Figure 1 | small1 | small2 | small3 |
|---|---|---|---|---|
| Our Encoding | 17 | 8 | 4 | 11 |
| Encoding in [2] | 46 | 33 | 18 | 44 |

**Table 2.** Comparison of two encodings for four different MCAPI programs with clause number.

Other than the program in Figure 1, we manually generate three programs and provide the experimental result in Table 2. Small1 is a program with two tasks, where two sends are in the first task, and two receives are in the second task. Small2 is a program with only one task, where one send and one receive are in this task. Small3 is a program with three tasks, where three sends are in the first task, two receives are in the second task, and one receive is in the third task. The number of clauses follows the performance of the SMT solver with fewer clauses leading to better runtime. The result in Table 2 is impressive because the number of our encoding is only $1/5 \sim 1/3$ of that of the encoding in [2]. From those experimental results, an SMT solver can give a more efficient solution with our encoding for an identical MCAPI program.

## 6 Related Work

Chao Wang et al. provided a symbolic algorithm that detects errors in all feasible permutations of statements in an execution trace [4]. They partition a concurrent program into a set of concurrent trace program (CTP), which we used in our work. Sharma et al. provided a verification tool, MCC, for verifying the MCAPI programs, but it was later discovered that this method does not completely

explore the entire execution space of certain kinds of MCAPI programs [3]. Elwakil et al. created a similar work with ours [1, 2]. We compared two works in the previous section, where their encoding does not always model all feasible execution traces and it lacks the efficiency for solving the SMT problem.

## 7 Conclusions

We have presented a trace language that can describe a CTP with a single trace on the same CTP. By executing a CTP with the language syntax, it can enter the final state that indicates whether the execution is correct or not. Also, we have defined an SMT problem that encodes the specific CTP, in which the match pair is defined to describe the transition of send and receive actions. Then we have proved the consistency of a trace language semantic and an SMT problem for a single trace given the same CTP. By expanding the set of match pair, we have proved that SMT problem can further imply more execution traces for a CTP. Thus, given a precise set of match pair, we can model all possible traces through an SMT problem. Finally, we provide a method of over-approximating the set of match pair that can lead to a refinement loop for finding errors.

## 8 Future Work

The generation of set of match pair is essential to the efficiency of our refinement process. Currently we have an over-approximated generation method, but the efficiency is not impressive. New methods for generating a much more precise set of match pair are required.

## References

1. Elwakil, M., Yang, Z.: CRI: Symbolic debugger for mcapi applications. In: Automated Technology for Verification and Analysis (2010)
2. Elwakil, M., Yang, Z.: Debugging support tool for mcapi applications. In: PADTAD '10: Proceedings of the 8th Workshop on Parallel and Distributed Systems (2010)
3. Sharma, S., Gopalakrishanan, G., Mercer, E., Holt, J.: Mcc - a runtime verification tool for mcapi user applications. In: FMCAD (2009)
4. 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)