Proving MCAPI Executions are Correct
Applying SMT Technology to Message Passing

Yu Huang, Eric 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 only the new schedule 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 something else specifications 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 constrains 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. As an example we present an abstract of a small program that uses MCAPI in Figure 1. Each node represents a different thread. The *Send* function call parameters include a unique identifier, the source endpoint, the destination endpoint, and the message contents, respectively. The *Recv* function call parameters include a unique identifier, the endpoint that will receive the message, and a buffer in which the contents of the message will be placed. We see that the first call to *Send* in Node 2 sends the contents of Y from *ep3* to *ep1*. This message can be received into A by the first call to *Recv* in Node 0. If this were to occur we would call this a *match pair* of *send1* and *recvA*.

If this abstract program is examined carefully it will be observed that messages received by *recvA* and *recvB* will be determined non-deterministically. One possible execution of the program would have the messages sent from *send1* and *send2*. *recvC* would receive the message from *send2* and then the message from *send3* would be sent. Then *recvA* could receive the message from *send1*, and *recvB* could receive the message from *send3*.

Another possible execution would have the message from *send1* being sent but then delayed in the communications layer. This would allow *send2* to send its message to *recvC* which would then allow *send3* to be paired with *recvA*. The message that was originally sent from *send1* might then be unblocked and be received by *recvB*.

It is important to recognize that the thread schedules of these two executions are exactly the same, even though the resulting behavior is completely different. It is the match pairings of the send and receive operations that introduce nondeterminism into the program. We have defined a process that takes as input an execution of an MCAPI program and verifies it, according to provided correctness properties. Output is given in the form of a path of execution to an error state, or as a declaration of correctness for the provided trace.

With this method of verifying individual execution traces, we have now begun work on defining a more complete method for verifying entire MCAPI applications. This method includes a concrete execution engine that controls the production of execution traces to be verified. The method will also include a property-based state-reduction technique that will provide input to the execution engine, allowing the whole system to remain complete in its exploration, while safeguarding the system from suffering the performance pains of the state-explosion problem.

Chao Wang provided a description of a similar verification method for multithreaded software that uses shared memory as a means of synchronization. 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 ex-

plore the entire execution space of certain kinds of MCAPI programs. Elwakil et al. defined a method of representing MCAPI program executions as satisfiability problems, but this method does not correctly model all kinds of MCAPI program executions [2] .

In this paper we describe our method of modeling executions of MCAPI programs as satisfiability problems such that the size of the problem scales linearly based on the number of possible match pairs that are being modeled. Negated system properties are integrated in the satisfiability problem in such a manner that if a satisfiable assignment is discovered for the problem, then the assignment can be used to simply create an error trace that indicates how the program could reach an error state in the program under test.

We will also describe how to determine an over-approximated set of match pairs, and how this set of match pairs can be used to create a satisfiability problem to precisely model all possible behaviors of an execution trace of an MCAPI application. Finally we will describe how this verification technique may be used within a refinement loop to provide a more efficient means of verifying MCAPI applications.

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 the top of 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 second 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 *ep0* 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 *ep1* in variable C on line `03` and then sends the message *"1"* on line `05` to *ep0*; and finally, task 2 sends messages *"4"* and *"Go"* on lines `04` and `06` to endpoints *ep0* and *ep1* 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?"*

Intuitively, variable a should contain 4 and variable b should contain 1 since task 2 must first send message *"4"* to *ep0* before it can send message *"Go"* to *ep1*; consequently, task 1 is then able to send message *"1"* to *ep0*. At the end of execution the assertion on line `12` of task 0 holds and no error is found. Such intuition is a valid program execution, and the sequential trace (ignoring endpoint creation) is given in the first column for each task in Figure 1 starting on line `04` of task 2. This single sequential execution is not sufficient to prove correctness however because of concurrency as other valid traces may exist. How to test those other traces is the question answered in this paper.

A single execution through a concurrent program can be understood as a concurrent trace program (CTP) which is a trace divided into its concurrent components including the requisite control flow on which the trace is valid [4]. The CTP for the trace indicated in the first column of each task in Figure 1 is shown in the bottom of the same figure. The CTP is more abstract than the original C code, and line numbers are now grouped into a program location (where the number in front of "_" is the thread number and the number behind is the command line) in each command, but it retains sufficient details to reason about other possible executions. The assume statement on program location `0_5` of task 0 preserves the control flow of the original trace. In other words, any alternative trace on the CTP is only valid if the assume statement holds; otherwise, such an execution belongs to a different path through the program. An SMT solver is used to enumerate the possible valid executions in an effort to find one that violates the assertion on program location `0_5` of the CTP for our example while preserving the assumption on line `0_4`.

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 2(a) is the set of possible match pairs for the CTP in the bottom of Figure 1 for our running example. 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.

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.

Figure 2(b) is the stylized SMT encoding for the CTP from our running example. Not shown are the definitions for `rcvA`, `rcvB`, etc. that are not novel to our solution but are records that group the various parameters for the respective send and receive calls with an event term to encode order. In our simple example, the send definitions have the match-pair and event term unconstrained. Receive definitions leave the event term, match-pair, and final value of the receive variable unconstrained. All of these unconstrained variables must be resolved by the SMT solver. The variables ending in "_loc" represent the event variables for the corresponding MCAPI operations. The `HB` terms encode constraints on the event variables to enforce program order as it appears in each task of the CTP. For example, by program order, `rcvA` happens before the wait command `wait_rcvA` for `rcvA` as shown in Figure 2(b). Similarly, `wait_rcvA` happens before `rcvB`. Any assignment to event variables must comply with program order constraints.

The `MATCH` terms in Figure 2(b) add constraints requisite with the specified match pair: the send and receive endpoints must be the same, the send value must appear in the receive variable, the send and receive events must be ordered appropriately, and the two events must be matched together. For example, matching `rcvA` with `snd3` forces the variable a to have the value 1. The `MATCH` function encodes this relationship. The `ne` term prevents the SMT solver from finding solutions that use conflicting match pairs. In our example, the `ne` term precludes `rcvA` and `rcvB` from matching to the same send. The assume 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 assert is negated as the goal is to find schedules that violate the property. In our example, the interest is in solutions the resolve in $a \neq 4$.

The SMT encoding in Figure 2(b) can be given to any preferred SMT solver and a satisfying assignment is a feasible execution that violates an assertion. For the example in Figure 1 such an execution does exist. The MCAPI natural language description states the send operation *"returns once the buffer can be reused by the application."*. As such, the return of the send only implies a copy-out of the message buffer and not a delivery to the intended endpoint; thus, an additional program execution places the value 1 in variable a and the value 4 in variable b. In order to place the value of some send to a variable of a specific receive call, we apply the queue movement in the sequential execution, where the pending sends in the source task are moved to the send queue in the destination task. The first column of each task in the bottom of Figure 2 gives the sequential program order of each statement for the error trace, and the second column indicates the appropriate queue movements where the match pairs are implied in Figure 2. For example, step `09` of the trace (found in task 0) indicates that the first item of the pending sends in task 1 is moved to the send queue in task 0, and the wait command on `rcvA` in program location `0_1` matches with `snd3` (the first item in the send queue in task 0). Conversations with the MCAPI designers confirm the intended behavior of the API to include both program executions of the scenario. To date, there have been three published verification and analysis tools purpose-built for MCAPI that omit this less intuitive program execution [3, 2, 1].

Finally, since the computation of match-pairs is expensive to do precisely, our solution includes the ability to over-approximate the match-pair set, and then validate satisfying assignments from the SMT solver to see if they use only feasible match pairs. Such validation is accomplished by creating 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 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[1] for MCAPI trace verification. The remainder of this paper details our MCAPI trace model, encoding, and an algorithm to compute an over-approximated match pair set.

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

Task 0	Task 1	Task 2
00 initialize(NODE_0,&v,&s); 01 ep0 = create_endpoint(PORT_0,&s); 04 02 msg_recv_i(ep0,A,sizeof(A),&rcvA,&s); 05 03 wait(&rcvA,&size,&s,MCAPI_INF); 06 04 a = atoi(A); 11 05 msg_recv_i(ep0,B,sizeof(B),&rcvB,&s); 12 06 wait(&rcvB,&size,&status,MCAPI_INF); 13 07 b = atoi(B); 14 08 if (b > 0) 15 09 assert(a == 4); 10 finalize(&s);	00 initialize(NODE_1,&v,&s); 01 ep1 = create_endpoint(PORT_1,&s); 02 ep0 = get_endpoint(NODE_0,PORT_0,&s); 07 03 msg_recv_i(ep1,C,sizeof(C),&rcvC,&s); 08 04 wait(&rcvC,&size,&s,INF); 09 05 msg_send_i(ep1,ep0,"1",2,N,&snd3,&s); 10 06 wait(&snd3,&size,&s,MCAPI_INF); 07 finalize(&s);	00 initialize(NODE_2,&v,&s); 01 t2 = create_endpoint(PORT_2,&s); 02 t0 = get_endpoint(NODE_0,PORT_0,&s); 03 t1 = get_endpoint(NODE_1,PORT_1,&s); 00 04 msg_send_i(ep2,ep0,"4",2,N,&snd1,&s); 01 05 wait(&snd1,&size,&status,MCAPI_INF); 02 06 msg_send_i(ep2,ep1,"Go",3,N,&snd2,&s); 03 07 wait(&snd2,&size,&status,MCAPI_INF); 08 09 finalize(&s);
08 0_0 (0_0 (rcvi rcvA 0 a)) 09 (0_1 (0 1)) (0_1 (wait rcvA)) 10 0_2 (0_2 (rcvi rcvB 0 b)) 11 (0_3 (0 2)) (0_3 (wait rcvB)) 12 0_4 (0_4 (assume (> b 0))) 13 0_5 (0_5 (assert (= a 4)))	04 1_0 (1_0 (rcvi rcvC 1 c)) 05 (1_1 (1 2)) (1_1 (wait rcvC)) 06 1_2 (1_2 (sndi snd3 1 0 0x1)) 07 1_3 (1_3 (wait snd3))	00 2_0 (2_0 (sndi snd1 2 0 0x4)) 01 2_1 (2_1 (wait snd1)) 02 2_2 (2_2 (sndi snd2 2 1 0x476f00)) 03 2_3 (2_3 (wait snd2))

Fig. 1. Two versions and traces of the same concurrent system with the top using MCAPI and the bottom using the trace language abstraction derived from the execution order in the first column of the program in the top.

```
(rcvA snd1)
(rcvA snd3)

(rcvB snd1)
(rcvB snd3)

(rcvC snd2)
```

(a)

```
(HB rcvA_loc wait_rcvA_loc)
(HB wait_rcvA_loc rcvB_loc)
(HB rcvB_loc wait_rcvB_loc)
(HB wait_rcvB_loc assume_loc)
(HB assume_loc assert_loc)
(HB rcvC_loc wait_rcvC_loc)
(HB wait_rcvC_loc snd3_loc)
(HB snd3_loc wait_snd3_loc)
(HB snd1_loc wait_snd1_loc)
(HB wait_snd1_loc snd2_loc)
(HB snd2_loc wait_snd2_loc)

(or (MATCH rcvA snd3)
    (MATCH rcvA snd1))
(or (MATCH rcvB  snd1)
    (MATCH rcvB snd3))
(ne rcvA rcvB)

(MATCH rcvC snd2)

(assume (> b 0))
(assert (not (= a 4)))
```

(b)

Fig. 2. A match-pair set and SMT encoding of the system in bottom half of 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.

$$
\begin{array}{rcl}
ctp & ::= & (t\ \ldots) \\
t & ::= & ([\rho\ c]\ \ldots\ \bot) \\
c & ::= & (\textbf{assume}\ e) \\
 & | & (\textbf{assert}\ e) \\
 & | & (\mathbf{x} := e) \\
 & | & (\textbf{sndi}\ a\ \alpha\ \beta\ e\ \rho) \\
 & | & (\textbf{rcvi}\ a\ \beta\ \mathbf{x}\ \rho) \\
 & | & (\textbf{wait}\ a) \\
m & ::= & \bot \mid \delta \\
\delta & ::= & ((\beta\ \alpha)\ \ldots\ \bot) \\
trace & ::= & (\sigma\ \ldots) \\
\sigma & ::= & (\rho\ m) \\
e & ::= & (\textbf{op}\ e\ e) \\
 & | & c \\
 & | & \mathbf{x} \\
 & | & v \\
v & ::= & \textbf{number} \\
 & | & bool \\
bool & ::= & \textbf{true} \\
 & | & \textbf{false} \\
\alpha & ::= & \gamma \\
\beta & ::= & \gamma
\end{array}
$$

(a)

$$
\begin{array}{rcl}
mstate & ::= & (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ ctp\ trace\ s) \\
qstate & ::= & (Pnd\_s\ Q\_s\ m\ s) \\
estate & ::= & (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ k) \\
h & ::= & \emptyset \mid (h\ [l \rightarrow v]) \\
\eta & ::= & \emptyset \mid (\eta\ [\mathbf{x} \rightarrow l]) \\
A & ::= & ((a\ \gamma)\ \ldots) \\
Pnd\_s & ::= & \emptyset \mid (Pnd\_s\ [\beta \rightarrow frm]) \\
frm & ::= & \emptyset \mid (frm\ [\alpha \rightarrow snd]) \\
snd & ::= & ([a\ \mathbf{v}]\ \ldots) \\
Pnd\_r & ::= & \emptyset \mid (Pnd\_r\ [\beta \rightarrow rcv]) \\
rcv & ::= & ([a\ \mathbf{x}]\ \ldots) \\
Q\_s & ::= & \emptyset \mid (Q\_s\ [\beta \rightarrow q]) \\
q & ::= & ([a\ \boldsymbol{v}{-}\bot]\ \ldots) \\
v{-}\bot & ::= & \bot \mid \mathbf{v} \\
s & ::= & \textbf{success} \\
 & | & \textbf{failure} \\
 & | & \textbf{infeasible} \\
 & | & \textbf{error} \\
k & ::= & \textbf{ret} \\
 & | & (\textbf{assert}\ \textbf{*} \rightarrow k) \\
 & | & (\textbf{assume}\ \textbf{*} \rightarrow k) \\
 & | & (\mathbf{x} := \textbf{*} \rightarrow k) \\
 & | & (\textbf{op}\ \textbf{*}\ e \rightarrow k) \\
 & | & (\textbf{op}\ \mathbf{v}\ \textbf{*} \rightarrow k)
\end{array}
$$

(b)

Fig. 3. The trace language syntax with its evaluation syntax for the operational semantics–bold face indicates a terminal. (a) The input syntax with terminals $\mathbf{x}$, ρ (which is unique), and a defined as strings and γ as a number. (b) The evaluation syntax with terminal l defined as a number.

Machine Step

$$
\frac{\begin{array}{l}(Pnd\_s\ Q\_s\ m\ s) \rightarrow_q^* (Pnd\_s_{p0}\ Q\_s_{p0}\ m_p\ s_{p0}) \\ (h\ \eta\ A\ Pnd\_s_{p0}\ Pnd\_r\ Q\_s_{p0}\ c_0\ s_{p0}\ \textbf{ret}) \rightarrow_e^* (h_p\ \eta_p\ A_p\ Pnd\_s_{p1}\ Pnd\_r_p\ Q\_s_{p1}\ e\ s_{p1}\ \textbf{ret})\end{array}}{\begin{array}{l}(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (t_0\ \ldots\ ([\rho_0\ c_0]\ [\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ ([\rho_0\ m]\ \sigma_1\ \ldots)\ s) \rightarrow_m \\ (h_p\ \eta_p\ A_p\ Pnd\_s_{p1}\ Pnd\_r_p\ Q\_s_{p1}\ (t_0\ \ldots\ ([\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ (\sigma_1\ \ldots)\ s_{p1})\end{array}}
$$

Fig. 4. Machine Reductions ($\rightarrow_m$)

Process Queue Movement

$$
\frac{\begin{array}{l}([a_s\ v]\ [a_1\ v_1]\ \ldots) = Pnd\_s(\beta)(\alpha) \\ Pnd\_s_p = [Pnd\_s \mid \beta \mapsto [Pnd\_s(\beta) \mid \alpha \mapsto ([a_1\ v_1]\ \ldots)]] \qquad ([a_1\ v_1]\ \ldots) = Q\_s(\beta) \\ Q\_s_p = [Q\_s \mid \beta \mapsto ([a_s\ v]\ [a_1\ v_1]\ \ldots)] \qquad s_p = \begin{cases} s & \text{if } |Pnd\_s(\beta)(\alpha)| > 0 \\ error & \text{otherwise} \end{cases}\end{array}}{(Pnd\_s\ Q\_s\ ((\beta\ \alpha)\ (\beta_0\ \alpha_0)\ \ldots\ \bot)\ s) \rightarrow_q (Pnd\_s_p\ Q\_s_p\ ((\beta_0\ \alpha_0)\ \ldots\ \bot)\ s_p)}
$$

Fig. 5. Queue Reductions ($\rightarrow_q$)

Variable Lookup
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ x\ s\ k) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ h(\eta(x))\ s\ k)$

Left Operand
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (op\ e_0\ e)\ s\ k) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e_0\ s\ (op\ *\ e \rightarrow k))$

Right Operand
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (op\ *\ e \rightarrow k)) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (op\ v\ *\ \rightarrow k))$

Binary Operation
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v_r\ s\ (op\ v_l\ * \rightarrow k)) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ \mathrm{op}(v_l, v_r)\ s\ k)$

Assume Expressions Evaluation
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (assume\ e)\ s\ k) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (assume\ * \rightarrow k))$

Assume Command
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (assume\ * \rightarrow k)) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ \mathrm{assume}(v, s)\ k)$

Assert Expressions Evaluation
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (assert\ e)\ s\ k) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (assert\ * \rightarrow k))$

Assert Command
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (assert\ * \rightarrow k)) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ \mathrm{assert}(v, s)\ k)$

Assign Expressions Evaluation
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (x\ :=\ e)\ s\ k) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (x\ :=\ * \rightarrow k))$

Assign Command
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (x\ :=\ * \rightarrow k)) \rightarrow_e ([h \mid \eta(x) \mapsto v]\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ k)$

Sndi Command

$$\frac{\begin{array}{l}([a_1\ v_1]\ \ldots) = Pnd\_s(\beta)(\alpha) \\ Pnd\_s_p = [Pnd\_s \mid \beta \mapsto [Pnd\_s(\beta) \mid \alpha \mapsto ([a_0\ h(\eta(x))]\ [a_1\ v_1]\ \ldots)]]\end{array}}{(h\ \eta\ ([a\ \gamma]\ \ldots)\ Pnd\_s\ Pnd\_r\ Q\_s\ (\mathbf{sndi}\ a_0\ \alpha\ \beta\ x)\ s\ k) \rightarrow_e (h\ \eta\ ([a_0\ \alpha]\ [a\ \gamma]\ \ldots)\ Pnd\_s_p\ Pnd\_r\ Q\_s\ \mathbf{true}\ s\ k)}$$

Rcvi Command

$$\frac{([a_1\ x_1]\ \ldots) = Pnd\_r(\beta) \qquad Pnd\_r_p = [Pnd\_r \mid \beta \mapsto ([a_0\ x_0]\ [a_1\ x_1]\ \ldots)]}{(h\ \eta\ ([a\ \gamma]\ \ldots)\ Pnd\_s\ Pnd\_r\ Q\_s\ (\mathbf{rcvi}\ a_0\ \beta\ x_0)\ s\ k) \rightarrow_e (h\ \eta\ ([a_0\ \beta]\ [a\ \gamma]\ \ldots)\ Pnd\_s\ Pnd\_r_p\ Q\_s\ \mathbf{true}\ s\ k)}$$

Wait (sndi) Command
$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (\mathbf{wait}\ a_s)\ s\ k) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ true\ s\ k)$

Wait (rcvi) Command

$$\frac{\begin{array}{l}([a_0\ \gamma_0]\ \ldots\ [a_r\ \beta][a_1\ \gamma_1]) = A \qquad A_0 = ([a_0\ \gamma_0]\ \ldots\ [a_1\ \gamma_1]) \\ (h_p\ a_s\ Pnd\_r_p\ Q\_s_p\ s_p) = \mathrm{getMarkRemove}(h, \eta, Pnd\_r, Q\_s, \beta, a_r, s) \\ ([a_0\ \gamma_0]\ \ldots\ [a_s\ \alpha][a_1\ \gamma_1]) = A_0 \qquad A_p = ([a_0\ \gamma_0]\ \ldots\ [a_1\ \gamma_1])\end{array}}{(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (\mathbf{wait}\ a_r)\ s\ k) \rightarrow_e (h_p\ \eta\ A_p\ Pnd\_s\ Pnd\_r_p\ Q\_s_p\ true\ s_p\ k)}$$

Fig. 6. Expression Reductions ($\rightarrow_e$)

3 Trace Language

The trace language is the theoretical framework for the match-pair encoding. The language syntax describes a CTP with a single execution trace on the same CTP. The evaluation syntax with its operational semantics define how to execute the CTP, following the specified trace, and define when that execution is a success (causes no assertion violation), a failure (causes an assertion violation), infeasible (causes an assume to not hold), or an error (uses a bogus match pair). Section 4 defines the encoding of a trace language program as an SMT problem and extends that encoding to capture a set of possible traces using match-pairs.

3.1 Syntax

Figure 3(a) is the syntax for a trace language program. This presentation uses ellipses (...) to represent zero or more repetitions, bold-face to indicate terminals, and omits commas in tuples for cleanliness. A trace language program is a CTP with a trace defining a sequential run of the CTP. The language defines a CTP (*ctp*) as a list of threads. A thread (t) is a list of pairs with each pair being a program location and a command. For simplicity, commands (c) are restricted to assume, assert, assignment, non-blocking send (*snd*) and receive (*rcv*), and wait. We model a trace as an order of executed locations identified by (ρ) and a series of queue movements that move a message from a source queue to a destination queue. The queue movement (m) is either a special symbol indicating no movement or a list of move commands. The move list (δ) consists of several pairs of end-points where the first and second end-points are the destination and source end-points respectively. Each pair in the move list refers to a queue movement such that the head item of the pending sends in the source end-point should be moved to the send queue in the destination end-point. The non-terminal a in the grammar is a unique string identifier associated with a send or receive command referred to as an action ID in the text. The wait command takes a single action ID belonging to the associated send or receive action. The non-terminals α and β are source and destination end-points respectively. The non-terminal γ is a number. The terminal **x** is any string not mentioned in the grammar definition and represents a program variable. Expressions (e) are defined using prefix notation over binary operators. The bottom of Figure 1 is an example *ctp* in the trace language (omitting the first and second columns and trivially grouping each thread and its commands into an appropriate list using parenthesis).

A sequential trace of a CTP in the grammar is a list of trace entries (σ). A trace entry is a pair consisting of a program location (ρ) and queue movement list. An example of a trace can be seen in the bottom of Figure 1 in the second column by following the sequential order in the first column which starts on program location `2_0` of task 2. Notice that whenever the trace reaches a wait command on a receive action, the trace includes a queue movement list that delivers messages from a source queue to a destination queue. In other words, the trace resolves any non-determinism in scheduling or message buffering that is present in the CTP.

3.2 Operational Semantics

The operational semantics for the trace language are given by a term rewriting system using a *CESK* style machine [2] only the machine is augmented to include additional structure for modeling message passing. Figure 3(b) defines the machine state and other syntax relating to evaluation.

A machine step ($\rightarrow_m$) in Figure 4 moves a thread forward by a single command. The rules operate on a machine state tuple (*mstate*) defined in Figure 3(b). The tuple can be partitioned into members relating to the *CESK* machine, members relating to the message passing model, and the trace status. The CESK machine members are *ctp* (the list of thread command sequences), η (an environment mapping a variable x to a location l), h (a store mapping a location l to a value v), and k (a continuation). Among the members of the message passing model, A is a dictionary mapping an action ID a to an end-point γ. *Pnd_s* is a set of send queues where each queue is uniquely identified first by the source end-point and then by the destination end-point. The queue itself holds pairs consisting of an action ID and value (a,v). *Pnd_r* is a set of receive queues where each queue is uniquely identified by the destination end-point. The queue itself holds pairs consisting of an action ID and a variable (a,x). *Q_s*is the structure similar as *Pnd_r*, except that the queue itself in the set is (a store mapping any given destination end-point β to a list of pairs where each pair is an action IF a and a value v). Intuitively, *Pnd_s* *Q_s* and *Pnd_r* are end-point queues tracking actions with associated values (sends) or variables (receives). Both *Pnd_s* and *Q_s* store the sends. *Q_s* holds delivered messages that are ready to be received. A message moves from *Pnd_s* to *Q_s* through queue movement lists in the trace on the CTP.

The trace status in the *mstate* nine-tuple is given by s which ranges over a lattice:

$$\textbf{success} \prec \textbf{failure} \prec \textbf{infeasible} \prec \textbf{error}$$

The trace status only moves monotonically up the lattice starting from success. A success trace completes the entire trace, meets all the assume statements, and does not fail an assertion. A failure trace completes the entire trace, meets all the assume statements, but fails an assertion. An infeasible trace completes the entire trace but does not meet all the assume statements. An error is a trace that does not complete.

We have several CESK machines that handle different aspects of the CTP and trace. The machine step moves one step on the trace. In each machine step, the queue machine processes any queue movements in that step, and the expression machine handles any expressions in the command associated with the trace step.

The *Machine Step* inference in Figure 4 matches any *mstate* that has a thread whose first list entry matches the program location in the head of the trace. A match on the inference rewrites the *mstate* with new entries for each member of the nine-tuple by first applying the queue reduction relation until no more

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

reductions apply (no more queue movement are performed) and then applying the expression reduction relation until no more reductions apply (as indicated by the asterisk). Note the queue reductions perform the queue movement such that all send actions are moved to the destination send queues, and the matching can be processed in the expression reduction. At this point, send and receive actions are matched in the wait expression.

The queue reduction for each command of the queue movement list in the trace entry is given in Figure 5. The definition of the *qstate* four-tuple is presented in the evaluation syntax in Figure 3(b). The symbol $\perp$ in the queue movement m indicates that no more queue movement follows. *Pnd_s* represents pending sends that have yet to be delivered from the source end-points. *Q_s* represents delivered sends sitting in the destination end-point. A message moves from pending to delivered through queue movement lists. Each reduction step processes the first pair of the queue movement list and reduction steps follow until $\perp$ is shown.

Expression reductions for each command in the trace language are given in Figure 6 and are defined over the *estate* nine-tuple in the evaluation syntax of Figure 3(b) which includes a continuation k. The **ret** continuation indicates that nothing follows, and an asterisk in a continuation is a place holder indication where evaluation is taking place. For example, the *Assume Expressions Evaluation* creates a continuation indicating that it is first evaluating the expression in the assume command. Once that expression reduces to a value, then the *Assume Command* inference matches to validate the assumptions.

The expression reductions use several helper functions. The function $op(v_l, v_r)$ applies the "op" to the left and right operands. The function getMarkRemove is explained in the later paragraph. The other helper functions are defined below:

$$\text{assert}(v, s) = \begin{cases} \textbf{failure} \text{ if } s \prec \textbf{failure} \wedge \\ \qquad\qquad v = \textbf{false} \\ s \qquad \text{otherwise} \end{cases}$$

$$\text{assume}(v, s) = \begin{cases} \textbf{infeasible} \text{ if } s \prec \textbf{infeasible} \wedge \\ \qquad\qquad v = \textbf{false} \\ s \qquad \text{otherwise} \end{cases}$$

Note that the status only moves monotonically up the lattice as mentioned previously. The notations $h(\eta(x))$ (Sndi Command), $Pnd\_r(\beta)$ (Rcvi Command), and $Pnd\_s(\beta)(\alpha)$ (Sndi Command) in Figure 6 are used for lookup. For example, $Pnd\_s(\beta)(\alpha)$ returns a list of pairs of action IDs and values as defined in Figure 3(b).

The *Sndi Command* and *Rcvi Command* in Figure 6 update *Pnd_r* and *Pnd_s* respectively with information to complete a message send or receive at a wait command. Consider a portion of the *Rcvi Command*:

$$Pnd\_r_p = [Pnd\_r \mid \beta \mapsto ([a_0\ x_0]\ [a_1\ x_1]\ \ldots)]$$
$$([a_1\ x_1]\ \ldots) = Pnd\_r(\beta)$$

Pnd_r_p is a new store, just like the old store *Pnd_r*, only the new store maps the destination end-point β to its contents in the old store plus the added entry to

the front of the list of the action ID and variable for the receive command being evaluated. Considering the entire rule in the figure, it also updates the action ID map, A, to include the coupling between the action ID and its destination end-point.

The function of the *Wait (SNDI) Command* rule in Figure 6 is to consume the wait command only. The function of the other rule for the wait command is more involved. The *Wait(RCVI) Command* rule is complicated because a CTP can wait on receive actions out of program order. Consider the simple trace *rcvA* followed by *rcvB*. It is perfectly valid to now call *wait(rcvB)* even though it appeared after the *rcvA* action. The *Wait(RCVI) Command* handles this very situation in function getMarkRemove by first finding in the pending receive queue the sub queue from the head of the queue to the receive being waited on. The length of the delivered queue Q_s must be at least as long as this sub queue. The actual wait command then updates the indicated variable in every receive preceding the one indicated by the wait command to reflect the completion of those commands which must precede the one indicated in the wait command.

The syntax with the operational semantics, as presented, are implemented directly in PLT Redex. PLT Redex is a language for testing and debugging semantics using term rewriting and is part of the Racket runtime.

The following definition is important to the proof of the match pair encoding in Section 4.

Definition 1. *A machine state* $(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ \text{ctp}\ trace\ s\ k)$ *is well formed if and only if it reduces to a final state where the CTP and trace run to completion, having matched every send and receive call, and the status is either success, failure, or infeasible:*

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ \text{ctp}\ trace\ s\ k) \rightarrow_m^* (h_p\ \eta_p\ ()\ Pnd\_s_p\ Pnd\_r_p\ Q\_s_p\ (()\ ...)\ ()\ s_p\ ret)$$

such that

$$\forall\beta, \forall\alpha, Pnd\_s_p(\beta)(\alpha) = () \wedge \forall\beta, Pnd\_r_p(\beta) = () \wedge \forall\beta, Q\_s_p(\beta) = () \wedge s_p \prec \textbf{\textit{error}}$$

For convenience, we define the function $\text{status}(m)$ *to return the final status after reduction of a well formed machine state* m*.*

Intuitively, a well-formed machine state completes all the transitions of send and receive calls, meaning that there are no elements in A, Pnd_s, Pnd_r and Q_s, the commands in the CTP and the execution trace are correctly executed, and the status of state never enters **error**.

4 SMT Model

The next step, after formally defining the operational semantics of our trace language, is to formally define a translation from the trace language into an SMT problem that can correctly model the original execution trace.

Machine Step

$$\frac{\begin{array}{l}(Pnd\_s\ Q\_s\ m\ s) \rightarrow_q^* (Pnd\_s_{p0}\ Q\_s_{p0}\ m_p\ s_{p0}) \\ (h\ \eta\ A\ Pnd\_s_{p0}\ Pnd\_r\ Q\_s_{p0}\ c_0\ s_{p0}\ \textbf{ret}\ l\ smt) \rightarrow_e^* (h_p\ \eta_p\ A_p\ Pnd\_s_{p1}\ Pnd\_r_p\ Q\_s_{p1}\ e\ s_{p1}\ \textbf{ret}\ l_p\ smt_{p0}) \\ smt_{p1} = \text{addHB}(smt_{p0}\ \rho_0\ \rho_1)\end{array}}{\begin{array}{l}(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (t_0\ \ldots\ ([\rho_0\ c_0]\ [\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ ([\rho_0\ m]\ \sigma_1\ \ldots)\ s\ l\ smt) \rightarrow_m \\ (h_p\ \eta_p\ A_p\ Pnd\_s_{p1}\ Pnd\_r_p\ Q\_s_{p1}\ (t_0\ \ldots\ ([\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ (\sigma_1\ \ldots)\ s_{p1}\ l_p\ smt_{p1})\end{array}}$$

Fig. 7. Machine Reductions to build the SMT model of a trace language program $(\rightarrow_{\text{m-smt}})$

Assume Expressions Evaluation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ (assume\ e)\ s\ k\ l\ (\text{defs}\ (\text{any}\ \ldots))) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ e\ s\ (assume * \rightarrow k)\ l\ (\text{defs}\ (e\ \text{any}\ \ldots)))$$

Assert Expressions Evaluation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ (assert\ e)\ s\ k\ l\ (\text{defs}\ (\text{any}\ \ldots))) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ e\ s\ (assert * \rightarrow k)\ l\ (\text{defs}\ ((\text{not}\ e)\ \text{any}\ \ldots)))$$

Assign Expressions Evaluation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ (x\ :=\ e)\ s\ k\ l\ (\text{defs}\ (\text{any}\ \ldots))) \rightarrow_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ e\ s\ (x\ :=\ * \rightarrow k)\ l\ (\text{defs}\ ((=\ x\ e)\ \text{any}\ \ldots)))$$

Sndi Command

$$\frac{\begin{array}{l}([a_1\ v_1]\ \ldots) = Pnd\_s(\beta)(\alpha) \\ Pnd\_s_p = [Pnd\_s \mid \beta \mapsto [Pnd\_s(\beta) \mid \alpha \mapsto ([a_0\ h(\eta(x))]\ [a_1\ v_1]\ \ldots)]] \\ \text{any}_0 = (\text{define}\ a\ ::\ send) \\ \text{any}_1 = (\text{and}\ (=\ (\text{select}\ a\ ep)\ \beta)\ (=\ (\text{select}\ a\ value)\ x))\end{array}}{\begin{array}{l}(h\ \eta\ ([a\ \gamma]\ \ldots)\ Pnd\_s\ Pnd\_r\ Q\_s\ (\textbf{sndi}\ a_0\ \alpha\ \beta\ x)\ s\ k\ l\ ((\text{any}_d\ \ldots)\ (\text{any}_a\ \ldots))) \rightarrow_e \\ (h\ \eta\ ([a_0\ \alpha]\ [a\ \gamma]\ \ldots)\ Pnd\_s_p\ Pnd\_r\ Q\_s\ \textbf{true}\ s\ k\ l\ ((\text{any}_0\ \text{any}_d\ \ldots)\ (\text{any}_1\ \text{any}_a\ \ldots)))\end{array}}$$

Rcvi Command

$$\frac{\begin{array}{l}([a_1\ x_1]\ \ldots) = Pnd\_r(\beta) \\ Pnd\_r_p = [Pnd\_r \mid \beta \mapsto ([a_0\ x_0]\ [a_1\ x_1]\ \ldots)] \qquad \text{any}_0 = (\text{define}\ a\ ::\ recv) \\ \text{any}_1 = (\text{and}\ (=\ (\text{select}\ a\ ep)\ \beta)\ (=\ (\text{select}\ a\ var)\ x_0))\end{array}}{\begin{array}{l}(h\ \eta\ ([a\ \gamma]\ \ldots)\ Pnd\_s\ Pnd\_r\ Q\_s\ (\textbf{rcvi}\ a_0\ \beta\ x_0)\ s\ k\ l\ ((\text{any}_d\ \ldots)\ (\text{any}_a\ \ldots))) \rightarrow_e \\ (h\ \eta\ ([a_0\ \beta]\ [a\ \gamma]\ \ldots)\ Pnd\_s\ Pnd\_r_p\ Q\_s\ \textbf{true}\ s\ k\ l\ ((\text{any}_0\ \text{any}_d\ \ldots)\ (\text{any}_1\ \text{any}_a\ \ldots)))\end{array}}$$

Wait (rcvi) Command

$$\frac{\begin{array}{l}([a_0\ \gamma_0]\ \ldots\ [a_r\ \beta][a_1\ \gamma_1]) = A \qquad A_0 = ([a_0\ \gamma_0]\ \ldots\ [a_1\ \gamma_1]) \\ (h_p\ a_s\ Pnd\_r_p\ Q\_s_p\ s_p) = \text{getMarkRemove}(h, \eta, Pnd\_r, Q\_s, \beta, a_r, s) \\ ([a_0\ \gamma_0]\ \ldots\ [a_s\ \alpha][a_1\ \gamma_1]) = A_0 \\ A_p = ([a_0\ \gamma_0]\ \ldots\ [a_1\ \gamma_1]) \qquad (l_p\ last\_a_s) = \text{getlastsend/replace}(l, a_s, \alpha, \beta)\end{array}}{\begin{array}{l}(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (\textbf{wait}\ a_r)\ s\ k\ l\ (\text{defs}\ (\text{any}_a\ \ldots))) \rightarrow_e \\ (h_p\ \eta A_p\ Pnd\_s\ Pnd\_r_p\ Q\_s_p\ true\ s_p\ k\ l_p\ (\text{defs}\ ((\text{HB}\ (\text{select}\ last\_a_s)\ MP)\ (\text{select}\ a_s\ MP))\ (\text{MATCH}\ a_r\ a_s)\ \text{any}_a\ \ldots)))\end{array}}$$

Fig. 8. Expression Reductions to build the SMT model of a trace language program $(\rightarrow_{\text{e-smt}})$

An SMT problem that encodes a trace through CTP needs various data structure and support functions. We have intuitively introduced most of them in the example of Section 2. Before presenting the translation framework from trace language to the SMT problem, we need to explain them in detail. We give the definitions in the Yices input language.

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

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. This function is used to assert the program order of statements within the same thread, and also to assert that, for any matched pair, that the send operation occurs before the receive.

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

We define two records in equation (2) that are essential to encoding the SMT problem. The first tuple R defined above represents a receive action such that M, ep, var, $event \in \mathbb{N}$. M represents a free variable that matches R with a tuple S. ep 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. The second tuple S defined above is a send action such that MP, ID, ep, $value$, $event \in \mathbb{N}$. ep and $event$, as those in R, are the end point of the send operation and event term, respectively. $value$ is the value for sending. MP is another free variable for match pair and it is assigned an end point of a receive operation if such receive operation matches the send operation. ID is a unique number that identifies each send operation. Note that integer value assigned to $event$ indicates the program order. For example, action a happens before b if and only if $event_a$ for a is greater than $event_b$ for b. Also, we keep record of two match variables, M and MP, for different purposes. M is used for avoiding two receive operations 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.

$$\begin{array}{l}(\textbf{define}\ \text{MATCH} :: (-> \ r{::}R\ s{::}S\ (\textbf{subtype}\ (c{::}bool)\ (ite \\ \quad (\text{and}\ (=\ (\text{select}\ r\ ep)\ (\text{select}\ s\ ep)) \\ \quad\quad (=\ (\text{select}\ r\ var)\ (\text{select}\ s\ value)) \\ \quad\quad (\text{HB}\ (\text{select}\ s\ event)\ (\text{select}\ r\ event)) \\ \quad\quad (=\ (\text{select}\ r\ M)\ (\text{select}\ s\ ID)) \\ \quad\quad (=\ (\text{select}\ s\ MP)\ (\text{select}\ r\ event))) \\ \quad (=\ c\ \text{true}) \\ \quad (=\ c\ \text{false})))))\end{array} \tag{3}$$

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.

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

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

We modify our system on the same framework of the trace language that produces a correlating SMT problem for a given execution trace.

The evaluation syntax for the machine reductions to build the SMT model of a trace language program are largely those of the regular machine reductions with a few changes below

$$\begin{array}{rcl}\textit{mstate} & ::= & (\textit{h}\ \eta\ \textit{A}\ \textit{Pnd\_s}\ \textit{Pnd\_r}\ \textit{Q\_s}\ \textit{ctp}\ \textit{trace}\ \textit{s}\ \textit{k}\ \textit{l}\ \textit{smt})\\ \textit{estate} & ::= & (\textit{h}\ \eta\ \textit{A}\ \textit{Pnd\_s}\ \textit{Pnd\_r}\ \textit{Q\_s}\ \textit{e}\ \textit{s}\ \textit{k}\ \textit{l}\ \textit{smt})\\ \textit{smt} & ::= & (\textit{defs}\ \textit{constraints})\\ \textit{defs} & ::= & (\textit{any}\ \ldots)\\ \textit{constraints} & ::= & (\textit{any}\ \ldots)\end{array}$$

The new syntax adds the *smt* member to the *mstate* and *estate* where the "*any*" term in *defs* and *constraints* matches any structure. The lists will be filled with definitions, HB entries, MATCH entries, NE entries, etc. as defined by the SMT machine reductions. *l*, which records the last send operation happens before the current one in the same thread, is another member added to the *mstate* and *estate*. The *qstate* does not contain any SMT encodings in the reductions and it is not changed.

The changes for the reduction rules are presented in Figure 7 and Figure 8. The support function addHB($((\textit{any}_d\ldots)\ (\textit{any}_c\ldots))\ \rho_0\ \rho_1$) adds program location to the definition list and adds a happens before relation

$$\equiv (((\texttt{define}\ \rho_0 :: \texttt{int})\ \textit{any}_d\ldots)((\texttt{HB}\ \rho_0\ \rho_1)\ \textit{any}_c\ldots))$$

The reduction rules in Figure 7 and Figure 8 add constrains to the SMT problem with respect to the semantic definition of trace language. The Machine Step

in Figure 7 adds the program order of actions in the same thread to the SMT problem. The evaluation of assume, assert and assign in Figure 8 adds assertions to the SMT problem. The *Sndi* and *Rcvi* commands define the send and receive actions in the SMT problem, and the $Wait(Rcvi)$ reduction adds the "match" constrains to the SMT problem. Note that the function getlastsend/replace picks up the last send operation, $last\_a_s$, that happens before the current one in the same thread, and adds a HB relation between the *MP* of $last\_a_s$ and that of the current send operation. The added relation ensures the matched receive operations of two send operations are also program ordered properly.

```
(HB rcvA (wait rcvA))
(HB (wait rcvA) rcvB)
(HB rcvB (wait rcvB))
(HB (wait rcvB) assume)
(HB assume assert)
(HB rcvC (wait rcvC))
(HB (wait rcvC) snd3)
(HB snd3 (wait snd3))
(HB snd1 (wait snd1))
(HB (wait snd1) snd2)
(HB snd2 (wait snd2))

(MATCH rcvA snd1)
(MATCH rcvB snd3)
(MATCH rcvC snd2)

(assume (> b 0))
(assert (not (= a 4)))
```

(a)

```
(HB rcvA (wait rcvA))
(HB (wait rcvA) rcvB)
(HB rcvB (wait rcvB))
(HB (wait rcvB) assume)
(HB assume assert)
(HB rcvC (wait rcvC))
(HB (wait rcvC) snd3)
(HB snd3 (wait snd3))
(HB snd1 (wait snd1))
(HB (wait snd1) snd2)
(HB snd2 (wait snd2))

(MATCH rcvA snd3)
(MATCH rcvB snd1)
(MATCH rcvC snd2)

(assume (> b 0))
(assert (not (= a 4)))
```

(b)

```
(HB rcvA (wait rcvA))
(HB (wait rcvA) rcvB)
(HB rcvB (wait rcvB))
(HB (wait rcvB) assume)
(HB assume assert)
(HB rcvC (wait rcvC))
(HB (wait rcvC) snd3)
(HB snd3 (wait snd3))
(HB snd1 (wait snd1))
(HB (wait snd1) snd2)
(HB snd2 (wait snd2))

(or (MATCH rcvA snd1)
    (MATCH rcvA snd3))
(or (MATCH rcvB snd1)
    (MATCH rcvB snd3))
(MATCH rcvC snd2)

(NE rcvA rcvB)

(assume (> b 0))
(assert (not (= a 4)))
```

(c)

Fig. 9. SMT problems. (a) SMT problem based on the first trace. (b) SMT problem based on a second trace. (c) SMT problem built from the preceding two problems.

Figure 9 contains three SMT problems for the same CTP in the bottom of Figure 1. Figure 9(a) uses several functions defined above to encode a single trace through the CTP. Especially, the assume statement is created by asserting any branch conditions that we encountered in the trace. As was stated earlier, this prevents the SMT solver from finding any solutions that are not consistent with the program's control flow. The assert statement is created by negating any assertions that were made in the original program. This creates an SMT problem that will only be satisfiable if an assertion violation can be found.

So far, we have achieved two different methods that can both verify the execution of a single trace. Furthermore, the following lemma indicates the equivalence between the trace language and the SMT encoding. Before presenting the lemma, we have the following definition.

Definition 2. *A machine state* $(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ \text{ctp}\ trace\ s\ k)$ *is smt-enabled if it is well formed. The SMT problem is taken from the final SMT reduced state:*

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ \text{ctp}\ trace\ s\ k\ ((())())) \rightarrow^*_{\text{m-smt}} (h_p\ \eta_p\ A_p\ Pnd\_s_p\ Pnd\_r_p\ \text{ctp}_p\ trace_p\ s_p\ k_p\ \text{smt})$$

For convenience, we define the function $\text{getSMT}(m) \mapsto \text{smt}$ *to return the SMT problem in the final state, and* $\text{ANS}(\text{getSMT}(m)) \mapsto \{\boldsymbol{SAT}, \boldsymbol{UNSAT}\}$ *to return the status of the SMT problem in the final state of* m*. Also, we define the relation of the range of function* ANS *such that* $\boldsymbol{UNSAT} > \boldsymbol{SAT}$*.*

Lemma 1. *For a well-formed machine state* m*,*

$$\text{status}(m) = \boldsymbol{failure} \iff \text{ANS}(\text{getSMT}(m)) = \boldsymbol{SAT}$$

Proof. Suppose the status of the final machine state is **failure**. From the reduction rules, we know that there exists at least one assert action that is evaluated false. In Figure 8, we know that expression "e" is negated and added to the SMT encoding for the second reduction rule. By hypothesis, e is evaluated false, so clause $(not\ e)$ is evaluated satisfiable in the generated SMT problem. Since the SMT problem generated by applying the reduction rules only remains a set of free variables for program ordering unresolved (other free variables are assigned to some value after creating the problem), the SMT problem exactly captures the execution trace described in the final state. Thus, the SMT problem is evaluated **SAT**. On the other hand, suppose the SMT problem is evaluated **SAT**, indicating that all statements of the SMT problem are satisfiable. Since the reduction rules in Figure 8 add statements with respect to the transition of the trace, it implies that each transition of the trace is executed correctly except for some assert action. Because of the same reason above, we know that expression e for the assert action is evaluated false. Thus, the status of the final machine state is **failure**.

As we have discussed in the early section, the CTP can have other traces in execution. Figure 9(b) represents a SMT problem that encodes another trace. The structure of the second SMT problem is similar to the first one. The only difference is the predefined set of match pair. Because of that, we combine two SMT problems into one, which is presented in Figure 9(c). To formally define the "combination" of multiple single SMT problems, we have the following definitions.

Definition 3. *A match-list ML is a set of match pair lists uniquely identified by a receive action ID, and each list consists of a set of match pairs for this receive action ID and a send action ID:*

$$ML\ =\ \emptyset\ |\ (ML\ [\mu\ \rightarrow [(\mu\ \nu)\ldots]])$$

Note that we alpha renamed so that μ *and* ν *are all unique labels. For convenience, we define function* $dom(ML)$ *and rangeML that returns the set of receive action IDs in the set of match pairs of MLand the set itself, respectively.*

We extend the structure *ML* to the structure *smt* such that

$$smt = (defs\ constrains\ ML)$$

where the match constraints are separated from *constraints*. In addition, two SMT problems are assumed to hold the same *defs* and *constrains* excluding the match constraints if and only if they can be combined by applying a combination operator in Definition 4.

Definition 4. *A combination operator COMB for two SMT problems, represented as three-tuples smt_1 and smt_2, where $smt_1 = (defs\ constrains\ ML_1)$ and $smt_2 = (defs\ constrains\ ML_2)$, returns a new SMT tuple, such that,*

$$COMB(smt_1, smt_2)\ =\ (defs\ constraints\ ML_{new})$$

where ML_{new} is a new relation such that $\forall \mu \in dom(ML_1) \cup dom(ML_2)$, $ML_{new}(\mu) = ML_1(\mu) \cdot ML_2(\mu)$.

To further finding the correlation between the combined SMT problem and the single SMT problems, we get the following lemma and proof.

Lemma 2. *For a set of traces $\mathrm{T_n}$ for the same CTP, and a set of SMT problems $\mathrm{SMT_n} = \{smt_0, smt_1, \ldots, smt_n\}$, where each member in $\mathrm{SMT_n}$ encodes a trace in $\mathrm{T_n}$, there exists a new SMT problem smt_{total}, where*

$$smt_{total} = COMB(smt_n, COMB(smt_{n-1}, COMB(smt_{n-2}, COMB(\ldots))))$$

and

$$\mathrm{ANS}(smt_{total}) = \begin{cases} \boldsymbol{SAT} & \text{iff } \exists smt_i \in \mathrm{SMT_n},\ s.t.\ \mathrm{ANS}(smt_i) = \boldsymbol{SAT} \\ \boldsymbol{UNSAT} & \text{otherwise} \end{cases}$$

Proof. We Prove it by induction.

We consider the base case as a SMT problem that encodes a single trace and the answer can be trivially proved by definition.

Induction. Assume we have combined n SMT problems and the combined SMT problem smt_{total} is evaluated **UNSAT**. We combine an additional SMT problem smt_{n+1}, which is different from any existent SMT problems. If $\mathrm{ANS}(smt_{n+1})$ is equal to **SAT**, the newly combined SMT problem, smt'_{total}, is evaluated **SAT** because the match pairs defined in smt_{n+1} are combined into smt'_{total}. If $\mathrm{ANS}(smt_{n+1})$ is equal to **UNSAT**, on the other hand, smt'_{total} is then evaluated **UNSAT** because all the traces implied in smt'_{total} are evaluated **UNSAT**. Additionally, it can be trivially proved if smt_{total} is evaluated **SAT** then the newly added SMT problem smt_{n+1} do not change the answer.

From **Lemma 2**, the combined SMT problem is more powerful because it can find the violation of an assertion among several trace encodings. Moreover, because the function *COMB* only adds match pairs in *ML*, we can conclude that it is all about match pairs that strength the verification process. The following theorem states this conclusion.

Theorem 1. *For two SMT problems,* $smt_\phi = (defs\ constrains\ ML_\phi)$ *and* $smt = (defs\ constrains\ ML)$*, if* $range(ML_\phi) \subseteq range(ML)$*,*

$$\text{ANS}(smt_\phi) \geq \text{ANS}(smt)$$

Proof. Since the range of ML_ϕ is the sub set of the range of ML, we can obtain smt by combining smt_ϕ with other SMT problems. Suppose $\text{ANS}(smt_\phi)$ is equal to **SAT**, by **Lemma 2**, $\text{ANS}(smt)$ is equal to **SAT** as well. If $\text{ANS}(smt_\phi)$ is equal to **UNSAT**, $\text{ANS}(smt)$ is equal to either **UNSAT** or **SAT**, depending on the answers of other single SMT problems that combine smt. In either case, $\text{ANS}(smt_\phi) \geq \text{ANS}(smt)$.

5 Result

In Section 4, we know that a well defined SMT problem can encode all possible execution traces for a CTP given a precise set of match pair. A precise set of match pair contains all possible match pairs that can exist in the real execution, and all match pairs that can not exist in the execution are not included. In other words, every hidden error can be found if it exists in some trace. To evaluate the reliability and efficiency of our system, we compare our encoding with the work presented in [2].

Elwakil et al. defined a similar order-based SMT encoding for the same problem.

$$F_R = F_{order} \wedge F_{asgn} \wedge F_{recv} \wedge F_{prp} \tag{5}$$

The formula in Equation (5) represents the main construction of their encoding. Four sub-formulas mimic the transition relations of the CTP, where F_{order} ensures the ordering of events, F_{asgn} encodes the assignment actions, F_{recv} ensures the matching of receive and send actions, and F_{prp} encodes the assertion actions. Each sub-formula has some similar clause of our encoding. For example, F_{order} in [2] works like HB function in our encoding. At the mean time, however, their encoding is different from ours on some parts. For example, They keep a history of values of local variables while we do not.

Their encoding can handle most regular scenarios of CTP and some errors can be found if such traces are modeled. However, It can not always guarantee of finding all hidden errors since the modeling is sometimes incorrect for a specific CTP. We have presented an example in the previous section. By applying the technique in [2] to this example, we can not model two execution traces that we have done so in the previous section. Actually, we can only model a trace that assigns "4" to "a". The other trace can not be modeled, since the encoding in [2] requires $snd1$ happening before $rcvB$ and no other candidate send operations for $rcvB$ happens after $snd1$. If we change the assertion in line 13 of the CTP to $(assert\ (=\ a\ \text{"}Go\text{"}))$, we can not find the error in the second trace. In our encoding, however, we have made it flexible that a non-blocking send operation can match a receive action even if some other sends happen before the receive action. So, our encoding can correctly model different traces.

An important measurement of efficiency for solving an SMT problem is to measure the "size" of the SMT encoding, one way of which is to count the numbers of the clauses that construct the encoding. To count the clause number of our encoding, we have the following rules: 1) there are $N - M$ HB clauses where N is the total number of actions and M is the number of thread; 2) there are $R+1$ MATCH clauses (some of them is a disjunction of several match pairs), where R is the number of receive actions; 3) there are A assignment and assertion clauses, where A is the total number of assignment and assertion in the program. Compared with those in [2], our encoding has few clauses, and their encoding has more free variables and rules.

For the example in the previous section, we encodes 17 clauses for modeling the problem, omitting the definition of variables at the beginning of the encoding. Based on their encoding rules, however, there are 47 clauses for the same problem, omitting the definitions as well. Reducing the clause number means a large improvement of efficiency. Thus, our encoding can be executed faster given an SMT solver.

6 Over-approximated Match-pair Generation

In the previous sections, we know that a precise set of match pair and a CTP is required to verify all possible execution traces through a SMT problem. Generating a precise set of match pair is time consuming because an execution tree needs to be generated, which requires exponential time complexity to finish. It is impossible to verify a regular MCAPI program because of the state explosion. In this section, we present a method that 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 are also included and required to prune. We then input this over-approximated set of match pair to a CEGAR loop for further verification. Either error is found during one iteration and the process stops, or the current set of match pair is prune to a more precise one for the next iteration.

Other than simulating all possible executions of the CTP, the algorithm that generates the over-approximated match pairs is basically a strategy of matching and pruning among all possible Sndi and Rcvi commands with matched end points. We use the structure in Figure 10 to support the algorithm. We keep one copy of both lists and store the Rcvi and Sndi commands as follows: The Rcvi commands in all threads are scanned and stored in the list of Rcvi, where the index of a Rcvi command, r_i, is the actual order in the thread where this Rcvi command exists. For example, $Rcvi(t_0, i)$ represents the i_{th} Rcvi command in Thread t_0. Similarly, the Sndi commands in all threads are stored and ordered in the list of Sndi, where s_i is the index of a Sndi command in the sublist of Sndi commands with same from-end-point and to-end-point. For example, $Sndi(t_0, t_1, j)$ represents the j_{th} Sndi command sending from t_1 to t_0. In addition, we have the following lemma that supports the pruning strategy in the algorithm.

```
([0 Revi_00, Revi_01, ...]
 [1 Revi_10, Revi_11, ...]
 ...
 [n Revi_n0, revi_n1, ...])
```

(a)

```
((0 [1 Sendi_100, Sendi_101, ...], [2 Sendi_200, Sendi_201, ...], ...)
 (1 [0 Sendi_010, Sendi_011, ...], [2 Sendi_210, Sendi_211, ...], ...)
 ...
 (n [0 Sendi_0n0, Sendi_0n1, ...], [1 Sendi_1n0, Sendi_1n1, ...], ...))
```

(b)

Fig. 10. Structure of pending lists. (a) List of Rcvi commands. (b) List of Sndi commands.

Lemma 3. *For each Rcvi command with index r_i and all matched Sndi commands of this Rcvi command, if s_j is the index of any Sndi commands,*

1. $r_i \geq s_j$;
2. $r_i \leq s_j + (N - sendsize_j)$.

Where N is the number of Sndi commands in all threads, and $sendsize_j$ is the size of the sublist where this Sndi command exists.

The algorithm that generates the over-approximated set of match pairs works as follows:

1 For each Rcvi command $Rcvi(t_s, i)$, all Sndi commands in the entire set of sublists $SndiList(t_s)$ are selected as candidates for matching the Rcvi command.
2 For each candidate Sndi command $Sndi(t_s, t_d, j)$, check if Lemma 3 is satisfied. If it is satisfied, the pair $(Rcvi(t_s, i), Sndi(t_s, t_d, j))$ is added to the result set of match pairs. Otherwise, this pair is ignored, and other candidate Sndi commands for $Rcvi(t_s, i)$ are checked.
3 Repeat step 1 and 2 until each Rcvi command is checked with all its candidate Sndi commands.

The generated set of match pairs is an over-approximation of the precise set because it contains all legal match pairs. Furthermore, the pruning strategy restricts the size of the set because some match pairs that can illegally exist are removed.

Now, we analyze the time complexity of this algorithm, which requires several steps to complete. Organizing the contents of pending lists requires linearly scanning the CTP so that the time complexity for this step is $O(N)$. Matching the possible Sndi commands with the given Rcvi commands requires $N_0^2 + N_1^2 + ... + N_n^2 < N^2$ steps, where $N_0 + N_1 + ... + N_n = N$, so this step requires $O(N^2)$ time complexity. Thus, the whole process requires $O(N^2)$ time complexity totally.

7 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.

8 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.

9 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)