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 is an important paradigm in writing applications for embedded heterogeneous multicore systems. The Multicore Association (MCA), an industry consortium promoting multicore technology, is working to standardize message passing into a single API, MCAPI, for bare metal implementation and portability across platforms. Correctness in such an API is difficult to reason about manually, and testing against reference solutions is equally difficult as reference solutions implement an unknown set of allowed behaviors, and programmers have no way to directly control API internals to expose or reproduce errors. This paper provides a way to encode an MCAPI execution as a Satisfiability Modulo Theories (SMT) problem, which if satisfiable, yields a feasible execution schedule on the same trace, such that it resolves non-determinism in the MCAPI runtime in a way that it now fails user provided assertions. The paper proves the problem is NP-complete. The encoding is useful for test, debug, and verification of MCAPI program execution. The novelty in the encoding is the direct use of match pairs (potential send and receive couplings). Match-pair encoding for MCAPI executions, when compared to other encoding strategies, is simpler to reason about, results in significantly fewer terms in the SMT problem, and captures feasible behaviors that are ignored in previously published techniques. Further, to our knowledge, this is the first SMT encoding that is able to run in *infinite-buffer* semantics, meaning the runtime has unlimited internal buffering as opposed to no internal buffering. Results demonstrate that the SMT encoding, restricted to zero-buffer semantics, uses fewer clauses when compared to another zero-buffer technique, and it runs faster and uses less memory. As a result the encoding scales well for programs with high levels of non-determinism in how sends and receives may potentially match.

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

[⋆] Special thanks to Christopher Fischer, formally at BYU and now at Amazon smaller and more modular. It is now becoming commonplace to find multiple processing units inside a single device. The Multicore Association (MCA) is an industry group that has formed to define specifications for low-level communication, resource management, and task management for embedded heterogeneous multicore devices [9].

One specification that the MCA has released is the Multicore Association Communications API (MCAPI) [10]. The specification defines types and functions for simple message passing operations between different computing entities within a device. Messages can be passed across persistent channels that force an ordering of the messages, or they can be passed to specific *endpoints* within the system. The specification places few ordering constraints on messages passed from one endpoint to another. This freedom introduces the possibility of a race between multiple messages to common endpoints thus giving rise to non-deterministic behavior in the runtime [14]. If an application has non-determinism, it is not possible to test and debug such an application without a way to directly (or indirectly) control the MCAPI runtime.

There are two ways to implement the MCAPI semantics: infinite-buffer semantics (the message is copied into a runtime buffer on the API call) and zero-buffer semantics (the message has no buffering) [18]. An infinite-buffer semantics provides more non-deterministic behaviors in matching send and receives because the runtime can arbitrarily delay a send to create interesting (and unexpected) send reorderings. The zero-buffer semantics follow intuitive message orderings as a send and receive essentially rendezvous.

Sharma et al. propose a method to indirectly control the MCAPI runtime to verify MCAPI programs under zero-buffer semantics [16]. As the work does not address infinite-buffer semantics, it is somewhat limited in its application. The work does provide a dynamic partial order reduction for the model checker, but such a reduction is not sufficient to control state space explosion in the presence of even moderate non-determinism between message sends and receives. A key insight from the approach is its direct use of match pairs–couplings for potential sends and receives.

Wang *et al.* propose an alternative method for resolving non-determinism for program verification using symbolic methods in the context of shared memory systems [19]. The work observes a program trace, builds a partial order from that trace called a concurrent trace program (CTP), and then creates an SMT problem from the CTP that if satisfied indicates a property violation.

Elwakil *et al.* extend the work of Wang *et al.* to message passing and claim the encoding supports both infinite and zero buffer semantics. A careful analysis of the encoding, however, shows it to not work under infinite-buffer semantics and to miss behaviors under zero-buffer semantics [6]. Interestingly, the encoding assumes the user provides a precise set of match pairs as input with the program trace, and it then uses those match pairs in a non-obvious way to constrain the happens-before relation in the encoding. The work does not discuss how to generate the match pairs, which is a non-trivial input to manually generate

Task 0	Task 1	Task 2
`00 initialize(NODE_0,&v,&s);`	`10 initialize(NODE_1,&v,&s);`	`20 initialize(NODE_2,&v,&s);`
`01 e0=create_endpoint(PORT_0,&s);`	`11 e1=create_endpoint(PORT_1,&s);`	`21 e2=create_endpoint(PORT_2,&s);`
	`12 e0=get_endpoint(NODE_0,PORT_0,&s);`	`22 e0=get_endpoint(NODE_0,PORT_0,&s);`
`02 msg_recv_i(e0,A,sizeof(A),&h1,&s);`		`23 e1=get_endpoint(NODE_1,PORT_1,&s);`
`03 wait(&h1,&size,&s,MCAPI_INF);`	`13 msg_recv_i(e1,C,sizeof(C),&h3,&s);`	
`04 a=atoi(A);`	`14 wait(&h3,&size,&s,MCAPI_INF);`	`24 msg_send_i(e2,e0,"4",2,N,&h5,&s);`
		`25 wait(&h5,&size,&s,MCAPI_INF);`
`05 msg_recv_i(e0,B,sizeof(B),&h2,&s);`	`15 msg_send_i(e1,e0,"1",2,N,&h4,&s);`	
`06 wait(&h2,&size,&s,MCAPI_INF);`	`16 wait(&h4,&size,&s,MCAPI_INF);`	`26 msg_send_i(e2,e1,"Go",3,N,&h6,&s);`
`07 b=atoi(B);`		`27 wait(&h6,&size,&s,MCAPI_INF);`
	`17 finalize(&s);`	
`08 if(b > 0);`		`28 finalize(&s);`
`09  assert(a == 4);`		
`0a finalize(&s);`		

Fig. 1. An MCAPI concurrent program execution

for large or complex program traces. An early proof claims that the problem of finding a precise set of match pairs given a program trace is NP-complete [15].

This paper presents a proof that resolving non-determinism in message passing programs in a way that meets all assertions is NP-complete. The paper then presents an SMT encoding for MCAPI program executions that works for both zero and infinite buffer semantics. The encoding does require an input set of match pairs as in prior work, but unlike prior work, the match-set can be over-approximated and the encoding is still sound and complete. The encoding requires fewer terms to capture all possible program behavior when compared to other proposed methods making it more efficient in the SMT solver. To address the problem of generating match pairs, an algorithm to generate the over-approximated set is given. To summarize, the main contributions in this paper are

1. a proof that the problem of matching sends to receives in a way that meets assertions is NP-complete;
2. a correct and efficient SMT encoding of an MCAPI program execution that detects all program errors under zero or infinite buffer semantics given the input set of potential match pairs contains at least the precise set of match pairs; and
3. an $O(N^2)$ algorithm to generate an over-approximation of possible match pairs, where N is the size of the execution trace in lines of code.

The rest of the paper is organized as follows: Section 2 presents an MCAPI program illustrating the non-determinism in the runtime specification. Section 3 is the proof that the problem is NP-complete. Section 4 defines the SMT encoding using potential math-pairs. The encoding is shown to be sound and complete even under an over-approximated set of match pairs. Section 5 provides a solution to the outstanding problem of generating feasible match pairs. Section 6 presents the experimental results that show the encoding to be efficient. Section 7 discusses related work. And Section 8 presents conclusions and future work.

2 Example

It is a challenge to explain intended behavior in simple scenarios consisting of a handful of calls when dealing with concurrency. Consider the MCAPI program execution in Figure 1 that includes three tasks that use send (`mcapi_msg_send_i`) and receive (`mcapi_msg_recv_i`) calls to communicate with each other. Line numbers appear in the first column for each task with the first digit being the task ID, and the declarations of the local variables are omitted for space.

Picking up the scenario just after the endpoints are defined, lines `02` and `05` receive two messages on the endpoint *e0* in variables A and B which are converted to integer values and stored in variables a and b on lines `04` and `07`; task 1 receives one message on endpoint *e1* in variable C on line `13` and then sends the message *"1"* on line `15` to *e0*; and finally, task 2 sends messages *"4"* and *"Go"* on lines `24` and `26` to endpoints *e0* and *e1* respectively. The additional code (lines `08 - 09`) asserts properties of the values in a and b. The `mcapi_wait` calls block until the associated send or receive buffer is able to be used. Given the scenario, a developer might ask the question: *"What are the possible values of* `a` *and* `b` *after the scenario completes?"*

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

---

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

---

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

---

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

---

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

---

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

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

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

From the trace, variable a should contain 4 and variable b should contain 1 since task 2 must first send message *"4"* to *e0* before it can send message

"Go" to *e1*; consequently, task 1 is then able to send message *"1"* to *e0*. The assume notation asserts the control flow taken by the program execution. In this example, the program takes the true branch of the condition on line `08`. At the end of execution the assertion on line `09` holds and no error is found.

```
24 S_{2,4}(0, &h5)
25 W(&h5)
26 S_{2,6}(1, &h6)
27 W(&h6)
--------------------
13 R_{1,3}(2, &h3)
14 W(&h3)
15 S_{1,5}(0, &h4)
16 W(&h4)
--------------------
02 R_{0,2}(1, &h1)
03 W(&h1)
04 a = atoi(A);
05 R_{0,5}(2, &h2)
06 W(&h2)
07 b = atoi(B);
08 assume(b > 0);
09 assert(a == 4);
--------------------
```

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

There is another feasible trace for the example shown in Figure 3 which is reachable under the infinite-buffer semantics. In this trace, the variable a contains 1 instead of 4, since the message *"1"* is sent to *e0* after sending the message *"Go"* to *e1* as it is possible for the send on line `24` to be buffered in transit. The MCAPI specification indicates that the wait on line `25` returns once the buffer is available. That only means the message is somewhere in the MCAPI runtime under the infinite-buffer semantics; it does not mean the message is delivered. As such, it is possible for the message to be buffered in transit allowing the send on line `15` to arrive at *e0* first and be received in variable *"a"*. Such a scenario is a program execution that results in an assertion failure at line `09`.

From the discussion above, it is important to consider non-determinism in the MCAPI runtime when testing or debugging an MCAPI program execution. The next section presents a proof that the problem of matching sends to receives in a way that meets all assertions is NP-complete. The proof justifies the encoding and SMT solver. Following the proof, the algorithm to generate the encoding is presented. It takes an MCAPI program execution with a set of possible send-receive match pairs and generates an SMT problem that if satisfied proves that non-determinism can be resolved in a way that violates a user provided assertion (the assertions are negated in the encoding) and if unsatisfiable proves the trace correct (meaning the user assertions hold on the given execution under all possible runtime behaviors). The encoding can be solved by an SMT solver such as Yices [4] or Z3 [12].

3 NP Completeness Proof

The complexity proof is inspired by the NP-completeness proof for memory coherence and consistency by Cantin *et al.* that uses a similar reduction from SAT only in the context of shared memory [2]. The complexity proof is on a new decision problem: *Verifying Assertions in Message Passing* (VAMP).

Definition 1. ***Verifying Assertions in Message Passing***.

INSTANCE: A set of constants D, a set of variables X, and a finite set H of task histories consisting of send, receive, and assert operations over X and D.

QUESTION: Is there a feasible schedule S for the operations of H that satisfy all the assertions?

The VAMP problem is NP-complete. The proof is a reduction from SAT. Given an instance Q of SAT consisting of a set of variables U and set of clauses C over U, an instance V of VAMP is constructed such that V has a feasible schedule S that meets all the assertions if and only if there is a satisfying truth assignment for Q. Feasible in this context means the schedule is allowed by the MCAPI semantics.

The reduction is illustrated in Figure 4. The figure elides the explicit calls to wait which directly follow each send and receive operation, and it elides the subscript notation as it is redundant in the figure. The figure also adds the value sent and the variable that receives the value to the notation as that information is pertinent to the reduction.

The reduction relies on non-determinism in the message passing to decide the value of each variable in U. The tasks h_{d_0} and h_{d_1} repeatedly send the constant value d_0 (false valuation) or d_1 (true valuation) to task h_C. The key intuition is that these tasks are synchronized with h_C so they essentially wait to send the value until asked.

The task h_C sequentially requests and receives d_0 and d_1 values for each variable in the SAT instance Q. It does not request values for a new variable until the current variable is resolved. As the values come from two separate tasks upon request, the messages race in the runtime and may arrive in either order at h_C. As a result, the value in each variable is non-deterministically d_0 or d_1.

After the value of each variable u_i is resolved, the h_C task asserts the truth of each clause in the problem instance. As the clauses are conjunctive, the assertions are sequentially evaluated. If a satisfying assignment exists for Q, then a feasible schedule exists that resolves the values of each variable in such a way that every assert holds.

Lemma 1. *S is a feasible schedule for H that satisfies all assertions if and only if Q is satisfiable.*

Proof. **Feasible schedule for V implies Q is satisfiable**: proof by contradiction. Assume that Q is unsatisfiable even though there is a feasible schedule S for V that meets all the assertions. The reduction in Figure 4 considers all

SAT: $U \equiv \{u_0, u_1, ..., u_m\}$ $C \equiv \{c_0, c_1, ..., c_n\}$ $Q \equiv \{c_0 \wedge c_1 \wedge ... \wedge c_n\}$		
VAMPI: $H \equiv \{h_{d_0}, h_{d_1}, h_C\}$ $X \equiv \{u_0, ..., u_m, g_0, g_1\}$ $D \equiv \{d_0, d_1\}$		
h_{d_0}	h_{d_1}	h_C
$R(g_0, *)$ $S(d_0, h_C)$	$R(g_1, *)$ $S(d_1, h_C)$	$S(d_0, h_{d_0})$ $S(d_0, h_{d_1})$ $R(u_0, *)$ $R(u_0, *)$
$R(g_0, *)$ $S(d_0, h_C)$	$R(g_1, *)$ $S(d_1, h_C)$	$S(d_0, h_{d_0})$ $S(d_0, h_{d_1})$ $R(u_1, *)$ $R(u_1, *)$
$\ldots$	$\ldots$	$\ldots$
		$assert(c_0)$ $assert(c_1)$ $\ldots$

Fig. 4. General SAT to VAMPI reduction

truth values of the variables in Q, over every combination, by virtue of the non-determinism, and then asserts the truth of each of the clauses in Q. The complete set of possibilities is realized by sending in parallel from h_{d_0} and h_{d_1} the two truth valuations for a given variable to h_C. As these messages may be received in any order, each variable may assume either truth value. Further, each variable resolved is an independent choice so all combinations of variable valuations must be considered. This fact is a contradiction to the assumption of Q being unsatisfiable as the same truth values that meet the assertions would be a satisfying assignment in Q.
Q is satisfiable implies feasible schedule for V: the proof is symmetric to the previous case and proceeds in a like manner.

Theorem 1 (NP-complete). *VAMP is NP-complete.*

Proof. **Membership in NP**: a certificate is a schedule matching send and receives in each of the histories. The schedule is linearly scanned with the histories and checked that it does not violate MCAPI semantics. The next section constructs an operational model of MCAPI semantics that does just such a check given a schedule. The complexity is linear in the size of the schedule.
NP-hard: polynomial reduction from SAT. The correctness of the reduction is established by Lemma 1. The remainder of the proof is the complexity of the reduction. There are two tasks to send values d_0 and d_1 upon request. For each variable $u_i \in U$, each of these tasks, d_0 and d_1, needs two operations: one to synchronize with h_C and another to send the value: $O(2 * 2 * |U|)$. The task h_C

must request values from h_{d_0} and h_{d_1} and then receive both those values; it must do this for each variable: $O(2 * 2 * |U|)$. Once all the values are collected, it must them assert each clause: $O(|C|)$. As every term is linear, the reduction is linear.

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

(a)

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

(b)

Fig. 5. 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

$$
\frac{\begin{array}{l}(\mathit{Pnd\_s}\ \mathit{Q\_s}\ m\ s)\ \to_q^*\ (\mathit{Pnd\_s}_{p0}\ \mathit{Q\_s}_{p0}\ m_p\ s_{p0}) \\ (h\ \eta\ A\ \mathit{Pnd\_s}_{p0}\ \mathit{Pnd\_r}\ \mathit{Q\_s}_{p0}\ c_0\ s_{p0}\ \mathbf{ret})\ \to_e^*\ (h_p\ \eta_p\ A_p\ \mathit{Pnd\_s}_{p1}\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}_{p1}\ e\ s_{p1}\ \mathbf{ret})\end{array}}{\begin{array}{l}(h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ (t_0\ \ldots\ ([\rho_0\ c_0]\ [\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ ([\rho_0\ m]\ \sigma_1\ \ldots)\ s) \to_m \\ (h_p\ \eta_p\ A_p\ \mathit{Pnd\_s}_{p1}\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}_{p1}\ (t_0\ \ldots\ ([\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ (\sigma_1\ \ldots)\ s_{p1})\end{array}}
$$

Fig. 6. Machine Reductions ($\to_m$)

4 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

Process Queue Movement

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

Fig. 7. Queue Reductions ($\to_q$)

Variable Lookup

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ x\ s\ k) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ h(\eta(x))\ s\ k)$$

Left Operand

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (op\ e_0\ e)\ s\ k) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e_0\ s\ (op\ *\ e \to k))$$

Right Operand

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (op\ *\ e \to k)) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (op\ v\ * \to k))$$

Binary Operation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v_r\ s\ (op\ v_l\ * \to k)) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ \mathrm{op}(v_l, v_r)\ s\ k)$$

Assume Expressions Evaluation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (assume\ e)\ s\ k) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (assume\ * \to k))$$

Assume Command

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (assume\ * \to k)) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ \mathrm{assume}(v, s)\ k)$$

Assert Expressions Evaluation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (assert\ e)\ s\ k) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (assert\ * \to k))$$

Assert Command

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (assert\ * \to k)) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ \mathrm{assert}(v, s)\ k)$$

Assign Expressions Evaluation

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (x\ :=\ e)\ s\ k) \to_e \\ (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ e\ s\ (x\ :=\ * \to k))$$

Assign Command

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ (x\ :=\ *\ \to k)) \to_e \\ ([h \mid \eta(x) \mapsto v]\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ v\ s\ k)$$

Sndi Command

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

Rcvi Command

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

Wait (sndi) Command

$$(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ (\mathbf{wait}\ a_s)\ s\ k) \to_e (h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ true\ s\ k)$$

Wait (rcvi) Command

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

Fig. 8. Expression Reductions ($\to_e$)

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 5 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.

4.1 Syntax

Figure 5(a) is the syntax for a trace language program. This presentation uses ellipses $(\ldots)$ 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 first pending send of the source end-point should be moved to the send queue of 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 $\mathbf{x}$ 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 either a queue movement list (m) or a symbol $(\perp)$. 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 20 of task 2. Notice that whenever the trace reaches a wait command on a receive action, the trace includes a queue movement list where the destination is listed followed by the source. Messages are delivered from the source queue to the destination queue. In other words, the trace resolves any non-determinism in scheduling or message buffering that is present in the CTP.

4.2 Operational Semantics

The operational semantics for the trace language are given by a term rewriting system using a *CESK* style machine [1] only the machine is augmented to include additional structure for modeling message passing. Figure 5(b) defines the machine state and other syntax relating to evaluation.

A machine step ($\rightarrow_m$) in Figure 6 moves a thread forward by a single command. The rules operate on a machine state tuple (*mstate*) defined in Figure 5(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 also a set of queues where each queue is uniquely identified by the destination end-point. The queue itself holds pairs consisting of an action ID a and either a value v or a symbol $\perp$. 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 6 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

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

reductions apply (all queue movements are processed in the trace entry first) and then applying the expression reduction relation until no more reductions apply (as indicated by the asterisk). Note that queue reductions perform the queue movement such that all send actions are moved to the destination send queues. After completing all queue movements, messages can be delivered in the process of expression reduction.

The queue reduction for each command of the queue movement list in the trace entry is given in Figure 7. The definition of the *qstate* four-tuple is presented in the evaluation syntax in Figure 5(b). The symbol $\perp$ in the queue movement m indicates that no more queue movement follows. 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 8 and are defined over the *estate* nine-tuple in the evaluation syntax of Figure 5(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 check 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:

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

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

Note that the status only moves monotonically up the lattice as mentioned previously. The notations $h(\eta(x))$ (Sndi Command), $\mathit{Pnd\_r}(\beta)$ (Rcvi Command), and $\mathit{Pnd\_s}(\beta)(\alpha)$ (Sndi Command) in Figure 8 are used for lookup. For example, $\mathit{Pnd\_s}(\beta)(\alpha)$ returns a list of pairs of action IDs and values as defined in Figure 5(b).

The *Sndi Command* and *Rcvi Command* in Figure 8 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*:

$$\begin{aligned} \mathit{Pnd\_r}_p &= [\mathit{Pnd\_r} \mid \beta \mapsto ([a_0\ x_0]\ [a_1\ x_1]\ \ldots)] \\ ([a_1\ x_1]\ \ldots) &= \mathit{Pnd\_r}(\beta) \end{aligned}$$

$\mathit{Pnd\_r}_p$ is a new set, just like the old set *Pnd_r*, only the new set maps the destination end-point β to its contents in the old set 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, $A = ([a_0\ \beta]\ [a\ \gamma]\ \ldots)$, to include the coupling between the action ID and its destination end-point.

The function of the *Wait(SNDI) Command* rule in Figure 8 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 following trace

$$(\ldots\ (rcvA\ \bot)\ (rcvB\ \bot)\ (wait(rcvB)\ \bot)\ (wait(rcvA)\ \bot)\ \ldots)$$

omitting the regular definition for the commands and the trace. 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. Consider the same trace above, we have the following structures of the *pending receive queue* and the *delivered queue* respectively,

$$Pnd\_r:\ (\ldots\ (\beta\ \rightarrow\ ([rcvA\ x_0]\ [rcvB\ x_1]\ \ldots))\ \ldots)$$

$$Q\_s:\ (\ldots\ (\beta\ \rightarrow\ ([snd1\ v_0]\ [snd2\ v_1]\ \ldots))\ \ldots)$$

Function getMarkRemove in the command $wait(rcvB)$ works as the following steps: first finding the pair $[rcvB\ x_1]$ in the pending receive queue; then getting the pair $[snd2\ v_1]$ in the delivered queue; then marking the pair $[snd1\ v_0]$ as first assigning v_0 to variable x_0 in the heap then updating the pair $[snd1\ v_0]$ to $[snd1\ \bot]$; finally removing the pair $[rcvB\ x_1]$ and $[snd2\ v_1]$ in Pnd_r and Q_s, respectively.

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 5.

Definition 2. *A machine state* $(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ \mathrm{ctp}\ trace\ s\ k)$ *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:*

$$\begin{array}{l}(h\ \eta\ A\ Pnd\_s\ Pnd\_r\ Q\_s\ \mathrm{ctp}\ trace\ s\ k) \rightarrow_m^* \\ (h_p\ \eta_p\ ()\ Pnd\_s_p\ Pnd\_r_p\ Q\_s_p\ ((\,)\ ...)\ ()\ s_p\ ret)\end{array}$$

such that

$$\begin{array}{l}\forall\beta,\forall\alpha, Pnd\_s_p(\beta)(\alpha) = () \land \forall\beta, Pnd\_r_p(\beta) = () \land \\ \forall\beta, Q\_s_p(\beta) = () \qquad\qquad \land\ s_p\ \prec \boldsymbol{error}\end{array}$$

For convenience, we define the function $\mathrm{status}(m)$ *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**.

$$\dfrac{\begin{array}{l}\textsc{Machine Step}\\ (\mathit{Pnd\_s}\ \mathit{Q\_s}\ m\ s)\ \to_q^*\ (\mathit{Pnd\_s}_{p0}\ \mathit{Q\_s}_{p0}\ m_p\ s_{p0})\\ (h\ \eta\ A\ \mathit{Pnd\_s}_{p0}\ \mathit{Pnd\_r}\ \mathit{Q\_s}_{p0}\ c_0\ s_{p0}\ \mathbf{ret}\ l\ \mathit{smt})\ \to_e^*\ (h_p\ \eta_p\ A_p\ \mathit{Pnd\_s}_{p1}\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}_{p1}\ e\ s_{p1}\ \mathbf{ret}\ l_p\ \mathit{smt}_{p0})\\ \mathit{smt}_{p1} = \mathrm{addHB}(\mathit{smt}_{p0}\ \rho_0\ \rho_1)\end{array}}{\begin{array}{l}(h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ (t_0\ \ldots\ ([\rho_0\ c_0]\ [\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ ([\rho_0\ m]\ \sigma_1\ \ldots)\ s\ l\ \mathit{smt}) \to_m\\ (h_p\ \eta_p\ A_p\ \mathit{Pnd\_s}_{p1}\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}_{p1}\ (t_0\ \ldots\ ([\rho_1\ c_1]\ [\rho_2\ c_2]\ \ldots)\ t_2\ \ldots)\ (\sigma_1\ \ldots)\ s_{p1}\ l_p\ \mathit{smt}_{p1})\end{array}}$$

Fig. 9. Machine Reductions to build the SMT model of a trace language program ($\to_{\mathrm{m-smt}}$)

$$\begin{array}{l}\textsc{Assume Expressions Evaluation}\\ (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ (\mathit{assume}\ e)\ s\ k\ l\ (\mathrm{defs}\ (\mathrm{any}\ \ldots))) \to_e (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ e\ s\ (\mathit{assume}\ * \to k)\ l\ (\mathrm{defs}\ (e\ \mathrm{any}\ \ldots)))\end{array}$$

$$\begin{array}{l}\textsc{Assert Expressions Evaluation}\\ (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ (\mathit{assert}\ e)\ s\ k\ l\ (\mathrm{defs}\ (\mathrm{any}\ \ldots))) \to_e (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ e\ s\ (\mathit{assert}\ * \to k)\ l\ (\mathrm{defs}\ ((\mathrm{not}\ e)\ \mathrm{any}\ \ldots)))\end{array}$$

$$\begin{array}{l}\textsc{Assign Expressions Evaluation}\\ (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ (x\ :=\ e)\ s\ k\ l\ (\mathrm{defs}\ (\mathrm{any}\ \ldots))) \to_e (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ e\ s\ (x\ :=\ * \to k)\ l\ (\mathrm{defs}\ ((=\ x\ e)\ \mathrm{any}\ \ldots)))\end{array}$$

$$\dfrac{\begin{array}{l}\textsc{Sndi Command}\\ \qquad ([a_1\ v_1]\ \ldots) = \mathit{Pnd\_s}(\beta)(\alpha)\\ \qquad \mathit{Pnd\_s}_p = [\mathit{Pnd\_s} \mid \beta \mapsto [\mathit{Pnd\_s}(\beta) \mid \alpha \mapsto ([a_0\ h(\eta(x))]\ [a_1\ v_1]\ \ldots)]]\\ \qquad \mathrm{any}_0 = (\mathrm{define}\ a\ ::\ \mathit{send})\\ \qquad \mathrm{any}_1 = (\mathrm{and}\ (=\ (\mathrm{select}\ a\ \mathit{ep})\ \beta)\ (=\ (\mathrm{select}\ a\ \mathit{value})\ x))\end{array}}{\begin{array}{l}(h\ \eta\ ([a\ \gamma]\ \ldots)\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ (\mathbf{sndi}\ a_0\ \alpha\ \beta\ x)\ s\ k\ l\ ((\mathrm{any}_d\ \ldots)\ (\mathrm{any}_a\ \ldots))) \to_e\\ (h\ \eta\ ([a_0\ \alpha]\ [a\ \gamma]\ \ldots)\ \mathit{Pnd\_s}_p\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ \mathbf{true}\ s\ k\ l\ ((\mathrm{any}_0\ \mathrm{any}_d\ \ldots)\ (\mathrm{any}_1\ \mathrm{any}_a\ \ldots)))\end{array}}$$

$$\dfrac{\begin{array}{l}\textsc{Rcvi Command}\\ \quad ([a_1\ x_1]\ \ldots) = \mathit{Pnd\_r}(\beta)\\ \quad \mathit{Pnd\_r}_p = [\mathit{Pnd\_r} \mid \beta \mapsto ([a_0\ x_0]\ [a_1\ x_1]\ \ldots)] \qquad \mathrm{any}_0 = (\mathrm{define}\ a\ ::\ \mathit{recv})\\ \quad \mathrm{any}_1 = (\mathrm{and}\ (=\ (\mathrm{select}\ a\ \mathit{ep})\ \beta)\ (=\ (\mathrm{select}\ a\ \mathit{var})\ x_0))\end{array}}{\begin{array}{l}(h\ \eta\ ([a\ \gamma]\ \ldots)\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ (\mathbf{rcvi}\ a_0\ \beta\ x_0)\ s\ k\ l\ ((\mathrm{any}_d\ \ldots)\ (\mathrm{any}_a\ \ldots))) \to_e\\ (h\ \eta\ ([a_0\ \beta]\ [a\ \gamma]\ \ldots)\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}\ \mathbf{true}\ s\ k\ l\ ((\mathrm{any}_0\ \mathrm{any}_d\ \ldots)\ (\mathrm{any}_1\ \mathrm{any}_a\ \ldots)))\end{array}}$$

$$\dfrac{\begin{array}{l}\textsc{Wait (rcvi) Command}\\ \qquad\qquad ([a_0\ \gamma_0]\ \ldots\ [a_s\ \alpha]\ \ldots\ [a_r\ \beta]\ \ldots\ [a_1\ \gamma_1]) = A\\ \qquad\qquad (h_p\ a_s\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}_p\ s_p) = \mathrm{getMarkRemove}(h, \eta, \mathit{Pnd\_r}, \mathit{Q\_s}, \beta, a_r, s)\\ \qquad\qquad A_p = ([a_0\ \gamma_0]\ \ldots\ [a_1\ \gamma_1]) \qquad (l_p\ \mathit{last\_a}_s) = \mathrm{getlastsend/replace}(l, a_s, \alpha, \beta)\end{array}}{\begin{array}{l}(h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ (\mathbf{wait}\ a_r)\ s\ k\ l\ (\mathrm{defs}\ (\mathrm{any}_a\ \ldots))) \to_e\\ (h_p\ \eta\ A_p\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}_p\ \mathit{Q\_s}_p\ \mathit{true}\ s_p\ k\ l_p\ (\mathrm{defs}\ ((\mathrm{HB}\ (\mathrm{select}\ \mathit{last\_a}_s)\ \mathit{MP})\ (\mathrm{select}\ a_s\ \mathit{MP}))\ (\mathrm{MATCH}\ a_r\ a_s)\ \mathrm{any}_a\ \ldots)))\end{array}}$$

Fig. 10. Expression Reductions to build the SMT model of a trace language program ($\to_{\mathrm{e-smt}}$)

5 SMT Encoding

The new SMT encoding is based on (1) a trace of events during an execution of an MCAPI program including control-flow assumptions and property assertions, such as Figure 2; and (2) a set of possible match pairs. A match pair is the coupling of a receive to a particular send. In the running example, the set admits, for example, that $\mathtt{R}_{0,2}$ can be matched with either $\mathtt{S}_{1,5}$ or $\mathtt{S}_{2,4}$. This direct use of match pairs, rather than a state-based or indirect use of match pairs in an order-based encoding, [6] and [5], is novel.

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

5.1 Definitions

The encoding needs to express the partial order imposed by the MCAPI semantics as SMT constraints. The partial order is based on a *Happens-Before* relation over operations such as send, receive, wait, or assert:

Definition 3 (Happens-Before). *The* Happens-Before (HB) *relation, denoted as $\prec_{\mathtt{HB}}$, is a partial order over operations.*

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

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

Definition 4 (Wait). *The occurrence of a wait operation,* W*, is captured by a single variable, $order_{\mathtt{W}}$, that constrains when the wait occurs.*

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

Definition 5 (Send). *A send operation* S*, is a four-tuple of variables:*
1. *$M_{\mathtt{S}}$, the order of the matching receive event;*
2. *$order_{\mathtt{S}}$, the order of the send;*
3. *$e_{\mathtt{S}}$, the endpoint; and,*
4. *$value_{\mathtt{S}}$, the transmitted value.*

The most complex operation in MCAPI is a receive. Since receives are inherently asynchronous, it is not possible to represent them atomically. Instead, we need to associate each receive with a wait that marks where in the program the receive operation is guaranteed to be complete. The MCAPI runtime semantics allow a single wait to witness the completion of many receives due to *the message non-overtaking property.* A wait that witnesses the completion of one or more receives is the *nearest-enclosing wait.*

Definition 6 (Nearest-Enclosing Wait). *A wait that witnesses the completion of a receive by indicating that the message is delivered and that all the previous receives in the same task issued earlier are complete as well.*

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

Fig. 11. Nearest-enclosing Wait example

Figure 11 shows that the wait $\mathtt{W}(\&\mathtt{h2})$ witnesses the completion of the receive $\mathtt{R}_{0,1}$ and $\mathtt{R}_{0,2}$ in task 0. Thus, $\mathtt{W}(\&\mathtt{h2})$ is their nearest-enclosing wait.

The encoding requires that every receive operation have a nearest-enclosing wait as it makes match pair decisions at the wait operation. The requirement is not a limitation of the encoding, as accessing a buffer from a receive that does not have a nearest-enclosing wait is an error. Rather, the wait is a convenience in the encoding to mark where a receive actually takes place. The same requirement can be made for sends for correctness but is not required for the encoding as send buffering is handled differently than receive buffering. The encoding effectively ignores wait operations for sends as will be seen.

Definition 7 (Receive). *A receive operation* $\mathtt{R}$ *is modeled by a five-tuple of variables:*

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

5.2 Assumptions, Assertions, and match pairs

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

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

Definition 8 (Assumption). *Every assumption* $\mathtt{A}$ *is added as an SMT assertion.*

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

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

Definition 9 (Property Assertion). *For every property assertion* $\mathtt{P}$, $\neg\mathtt{P}$ *is added as an SMT assertion.*

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

Definition 10 (Match Pair). *A match pair,* $\langle \mathtt{R}, \mathtt{S} \rangle$, *for a receive* $\mathtt{R}$ *and a send* $\mathtt{S}$ *corresponds to the constraints:*
1. $M_{\mathtt{R}} = order_{\mathtt{S}}$
2. $M_{\mathtt{S}} = order_{\mathtt{R}}$
3. $e_{\mathtt{R}} = e_{\mathtt{S}}$
4. $value_{\mathtt{R}} = value_{\mathtt{S}}$ *and*
5. $order_{\mathtt{S}} \prec_{\mathtt{HB}} nw_{\mathtt{R}}$

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

Definition 11 (Receive Matches). *For each receive* $\mathtt{R}$, *if* $\langle \mathtt{R}, \mathtt{S}_0 \rangle$ *through* $\langle \mathtt{R}, \mathtt{S}_n \rangle$ *are match pairs, then* $\bigvee_i^n \langle \mathtt{R}, \mathtt{S}_i \rangle$ *is used as an SMT constraint.*

This encoding of the input ensures that the SMT solver can only use compatible send/receive pairs and ensures that sends happen before nearest-enclosing waits on receives.

5.3 Program Order Constraints

The encoding thus far is missing additional constraints on the *Happens-Before* relation stemming from program order. These constraints are added in four steps: we must ensure that sends to common endpoints occur in program order in a single task (step 1); similarly for receives (step 2); receives occur before their nearest-enclosing wait (step 3); and, that sends are received in the order they are sent (step 4).

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

Step 2 For each task, if there are sequential receive operations, say $\mathtt{R}$ and $\mathtt{R}'$, in that task on a common endpoint, $e_{\mathtt{R}} = e_{\mathtt{R}'}$, then those receives must follow program order: $\mathit{order}_{\mathtt{R}} \prec_{\mathtt{HB}} \mathit{order}_{\mathtt{R}'}$.

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

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

For example, consider two tasks where task 0 sends two messages to task 1 as shown in Figure 12.

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

Fig. 12. Send Ordering Example

The $M_{\mathtt{S}}$ variables from the sends will be assigned to the orders for $\mathtt{R}_{1,1}$ and $\mathtt{R}_{1,2}$ by the match pairs selected by the SMT solver. The constraints added in this step force the send to be received in program order using the HB relation which for this example yields $M_{\mathtt{S}_{0,1}} \prec_{\mathtt{HB}} M_{\mathtt{S}_{0,2}}$.

5.4 Zero Buffer Semantics

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

First, for each task, if there are two sends $\mathtt{S}$ and $\mathtt{S}'$ such that $\mathit{order}_{\mathtt{S}} \prec_{\mathtt{HB}} \mathit{order}_{\mathtt{S}'}$, and $\mathtt{S}$ and $\mathtt{S}'$ can both match a receive $\mathtt{R}$, then we add the following constraint to the encoding: $\mathit{order}_{\mathtt{W}} \prec_{\mathtt{HB}} \mathit{order}_{\mathtt{S}'}$ where $\mathtt{W}$ is the nearest-enclosing wait that witnesses the completion of $\mathtt{R}$ in execution.

The second constraint relies on a dependence relation between two match pairs.

Definition 12. *To match pairs are dependent, denoted as* $\langle\mathtt{R}, \mathtt{S}\rangle \rightharpoonup \langle\mathtt{R}', \mathtt{S}'\rangle$, *if and only if*

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

Fig. 13. SMT Encoding

1. *the nearest-enclosing wait* W *of* R′ *issues before* S *on an identical endpoint; or*
2. $\exists\langle \mathtt{R}''\mathtt{S}''\rangle$ *such that* $\langle \mathtt{R},\mathtt{S}\rangle \rightharpoonup \langle \mathtt{R}'',\mathtt{S}''\rangle \wedge \langle \mathtt{R}'',\mathtt{S}''\rangle \rightharpoonup \langle \mathtt{R}',\mathtt{S}'\rangle$.

With the dependence relation, the second set of constraints for the zero-buffer semantics is give as: for each pair of sends S and S′ that can both match an receive R, if there is a send S″ issued after the issuing of S′ by an identical endpoint, and a receive R′ such that $\langle \mathtt{R},\mathtt{S}\rangle \rightharpoonup \langle \mathtt{R}',\mathtt{S}''\rangle$, then we add the following constraint to the encoding: $\mathit{order}_{\mathtt{W}} \prec_{\mathtt{HB}} \mathit{order}_{\mathtt{S}}$ where W is the nearest-enclosing wait that witnesses the completion of R.

5.5 Example

Figure 13 shows the encoding of Figure 1 as an SMT problem. We elide the basic definition of the variables discussed in Section 5.1. Lines 05 through 09 give the assumptions, assertions, and match pairs. The first four lines reflect the program order constraints: receives happen before corresponding wait operations and receives from a common endpoint follow program order. There are no constraints between sends because there are no sequential sends from a common endpoint to a common endpoint. To encode the zero-buffer semantics, the constraint $\mathit{order}_{\mathtt{W(\&h1)}} \prec_{\mathtt{HB}} \mathit{order}_{\mathtt{S}_{1,5}}$ would need to be added to force the receive to complete before another send is issued.

5.6 Correctness

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

Definition 13 (Encoder). *For all programs, p, and match pair sets m, let $\mathcal{SMT}(p, m)$ be our encoding as an SMT problem.*

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

Definition 14 (SMT Solver). *For all SMT problems, s, let $\mathcal{SOL}(s)$ be in $\sigma + \mathbb{UNSAT}$, where σ is a satisfying assignment of variables to values.*

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

We assume a semantics for traces that gives their behavior as either having an assertion violation or being correct. This formal semantics is presented in Section 4.

Definition 15 (Semantics). *For all programs, p, and traces t, $\mathcal{SEM}(p,t)$ is either $\mathbb{BAD}$ or $\mathbb{OK}$.*

Given this framework, our SMT encoding technique is sound if

Theorem 2 (Soundness). *For all programs, p, and match pair sets, m, $\mathcal{SOL}(\mathcal{SMT}(p,m)) = t \Rightarrow \mathcal{SEM}(p,t) = \mathbb{BAD}$.*

Our soundness proof relies on the following lemma:

Lemma 2. *Any match pair $\langle \mathtt{R}, \mathtt{S} \rangle$ used in a satisfying assignment of an SMT encoding is a valid match pair and reflects an actual possible MCAPI program execution.*

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

Since $\langle \mathtt{R}, \mathtt{S} \rangle$ is not a valid match pair, match R and S requires program order, message non-overtaking, or no-multiple match to be violated. In other words, the *Happens-Before* constraints encoded in the SMT problem are not satisfied.

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

The correctness of our technique relies on completeness:

Theorem 3 (Completeness). *For all programs, p, and traces, t, $\mathcal{SEM}(p,t) = \mathbb{BAD} \Rightarrow \exists m.\mathcal{SOL}(\mathcal{SMT}(p,m)) = t$.*

To completely prove completeness, we give the following modified framework 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

$$\begin{aligned}
\mathit{mstate} &::= (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ \mathit{ctp}\ \mathit{trace}\ s\ k\ l\ \mathit{smt}) \\
\mathit{estate} &::= (h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathit{Q\_s}\ e\ s\ k\ l\ \mathit{smt}) \\
\mathit{smt} &::= (\mathit{defs}\ \mathit{constraints}\ \mathit{ML}) \\
\mathit{defs} &::= (\mathit{any}\ \ldots) \\
\mathit{constraints} &::= (\mathit{any}\ \ldots) \\
\mathit{ML} &::= (\mathit{ML} = \emptyset \mid (\mathit{ML}\ [\mu \rightarrow [(\mu\ \nu)\ldots]]))
\end{aligned}$$

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, etc. as defined by the SMT machine reductions. 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. Note that we alpha renamed so that μ and ν are all unique labels. For convenience, we define function $\mathit{dom}(\mathit{ML})$ and $\mathit{range}(\mathit{ML})$ that return the set of receive action IDs in the set of match pairs of *ML* and the set itself, respectively. The symbol l is another member added to the *mstate* and *estate*. It records the last send operation that happens before the current one in the same thread. 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 9 and Figure 10. The support function $\mathrm{addHB}(((\mathit{any}_d\ldots)\ (\mathit{any}_c\ldots))\ \rho_0\ \rho_1)$ adds program location to the definition list and adds a happens before relation

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

The reduction rules in Figure 9 and Figure 10 add constraints to the SMT problem with respect to the semantic definition of the trace language. The Machine Step in Figure 9 adds the program order of actions in the same thread to the SMT problem. The evaluation of assume, assert and assign in Figure 10 adds assertions to the SMT problem. The *Sndi* and *Rcvi* commands define the send and receive actions in the SMT problem, and the $\mathit{Wait}(\mathit{Rcvi})$ reduction adds the "match" constraints to the SMT problem. Note that the function getlastsend/replace picks up the last send operation, $\mathit{last\_a}_s$, that happens before the current one in the same thread, and adds a HB relation between the *MP* of $\mathit{last\_a}_s$ and that of the current send operation. The added relation ensures the messages from a common endpoint are non-overtaking (i.e., FIFO ordered).

The following definition supports the rest of the section.

Definition 16. *A machine state* $(h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathrm{ctp}\ \mathit{trace}\ s\ k)$ *is smt-enabled if it is well formed. The SMT problem is taken from the final SMT reduced state:*

$$\begin{gathered}
(h\ \eta\ A\ \mathit{Pnd\_s}\ \mathit{Pnd\_r}\ \mathrm{ctp}\ \mathit{trace}\ s\ k\ (()())) \rightarrow^*_{\mathrm{m-smt}} \\
(h_p\ \eta_p\ A_p\ \mathit{Pnd\_s}_p\ \mathit{Pnd\_r}_p\ \mathrm{ctp}_p\ \mathit{trace}_p\ s_p\ k_p\ \mathrm{smt})
\end{gathered}$$

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

defs is not shown;

constraints;

00 $event_{R_{0,2}} \prec_{HB} event_{W(\&h1)}$
01 $event_{R_{0,5}} \prec_{HB} event_{W(\&h2)}$
02 $event_{R_{0,2}} \prec_{HB} event_{R_{0,5}}$
03 $event_{R_{1,3}} \prec_{HB} event_{W(\&h3)}$
04 $event_{S_{1,5}} \prec_{HB} event_{W(\&h4)}$
05 $event_{S_{2,4}} \prec_{HB} event_{W(\&h5)}$
06 $event_{S_{2,7}} \prec_{HB} event_{W(\&h6)}$
07 (*assert* (> b 0))
08 (*assert* (not (= a 4)))

match;

09 $\langle R_{0,2}, S_{2,4}\rangle$
10 $\langle R_{0,5}, S_{1,5}\rangle$
11 $\langle R_{1,3}, S_{2,7}\rangle$

(a)

defs is not shown;

constraints;

00 $event_{R_{0,2}} \prec_{HB} event_{W(\&h1)}$
01 $event_{R_{0,5}} \prec_{HB} event_{W(\&h2)}$
02 $event_{R_{0,2}} \prec_{HB} event_{R_{0,5}}$
03 $event_{R_{1,3}} \prec_{HB} event_{W(\&h3)}$
04 $event_{S_{1,5}} \prec_{HB} event_{W(\&h4)}$
05 $event_{S_{2,4}} \prec_{HB} event_{W(\&h5)}$
06 $event_{S_{2,7}} \prec_{HB} event_{W(\&h6)}$
07 (*assert* (> b 0))
08 (*assert* (not (= a 4)))

match;

09 $\langle R_{0,2}, S_{1,5}\rangle$
10 $\langle R_{0,5}, S_{2,4}\rangle$
11 $\langle R_{1,3}, S_{2,7}\rangle$

(b)

...
01 $order_{R_{0,2}} \prec_{HB} order_{W(\&h1)}$
02 $order_{R_{0,5}} \prec_{HB} order_{W(\&h2)}$
03 $order_{R_{0,2}} \prec_{HB} order_{R_{0,5}}$
04 $order_{R_{1,3}} \prec_{HB} order_{W(\&h3)}$
05 (*assert* (> b 0))
06 (*assert* (not (= a 4)))
07 $\langle R_{0,2}, S_{2,4}\rangle \vee \langle R_{0,2}, S_{1,5}\rangle$
08 $\langle R_{0,5}, S_{2,4}\rangle \vee \langle R_{0,5}, S_{1,5}\rangle$
09 $\langle R_{1,3}, S_{2,7}\rangle$

(c)

Fig. 14. 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 14 shows the SMT problem (minus the event definitions) generated from our new *CESK SMT* encoding machines for the program in Figure 1 on two different traces (parts (a) and (b)). Note that the encodings in Figure 14 (a) and (b) are identical except for the MATCH clauses, which are generated from the trace parts of the program. Furthermore, the encoding in Figure 14(a) violates the properties of interest encoded in the assert clauses, and the answer for the encoding in Figure 14 (a) is SAT. From this observation, we present the following lemma that indicates the equivalence between the status of our *CESK* machines and the satisfiability of the generated SMT encoding.

Lemma 3. *For a well-formed machine state m,*

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

Proof. The generated SMT problem in Figure 9 and Figure 10 captures the execution trace from the initial state up to the final state by looking at the following facts. First, the machine step in Figure 9 adds an HB relation for two consecutive program locations ρ_0 and ρ_1 in an identical thread, with respect to the program order constraint. Second, the assume, assert and assign expression evaluations in Figure 10 add assert constraints for variable x or expression e

with respect to the trace constraint. Note that the negated system is used in the assert evaluation when generating the SMT statement. Third, the sndi and rcvi commands in Figure 10 add any_0 as the definition of a receive or send operation, followed by any_1 as the initialization of operation fields. Finally, the wait(rcvi) command in Figure 10 adds a match pair for the receive operation a_r and the send operation a_s where a_s is obtained by applying the process we described in Section 3.2. Furthermore, an HB relation is added with respect to the program order, such that the messages from a common endpoint are non-overtaking. The observations described above either adds program order constraints for the execution trace modeled in the machine states, or adds constraints for variables or expressions that are also modeled in the machine state. In a word, the generated SMT problem captures the execution trace in the machine.

Notice the fact above, we 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 10, we know that expression "e" is negated and added to the SMT encoding for the second reduction rule. Since e is evaluated false, (*not* e) is evaluated true trivially in the generated SMT problem. Other statements are all evaluated true due to the trace capture fact that the program order is assigned correctly based on the execution trace modeled in the machine states, and the variables and expressions are also evaluated true in assume and assign commands. 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 10 add statements with respect to the transition flow 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 discussed early, Figure 14 (a) and (b) are generated from the same program with two different traces. The encoding in Figure 14 (c) combines part (a) and (b) into one, the answer of which implies the non-deterministic behavior of the program. By solving the problem in Figure 14 (c), we implicitly solve two problems in Figure 14 (a) and (b) respectively. The following definition defines the "combination" behavior of two SMT problems.

Definition 17. *A combination operator COMB for two SMT problems, represented as three-tuples smt_1 and smt_2, where $smt_1 = (defs\ constraints\ ML_1)$ and $smt_2 = (defs\ constraints\ ML_2)$, 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 $\forall \mu \in dom(ML_1) \cup dom(ML_2)$, $ML_{new}(\mu) = ML_1(\mu) \cdot ML_2(\mu)$. Note that two SMT problems are assumed to hold the same defs and constraints excluding the match constraints.

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

Lemma 4. *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* $\mathrm{T_n}$ *according to our trace machine, 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. The following theorem states the relation between the ability of finding violation and the content of the match-list *ML*.

Theorem 4. *For two SMT problems,* $smt_\phi = (defs\ constraints\ ML_\phi)$ *and* $smt = (defs\ constraints\ ML)$*, if* $range(ML_\phi) \subseteq range(ML)$*,*

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

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

These theorems above obscure an important problem: how do we know which match pair set to use? Soundness assumes we have one, while completeness merely asserts that one exists. Although Section 7 discusses our generation algorithm, we prove here an additional theorem that asserts that any conservative over-approximation of match pair sets is safe.

Theorem 5 (Approximation). *Give two match pair sets m and m', $m \subseteq m' \Rightarrow \mathcal{SOL}(\mathcal{SMT}(p, m)) \sqsubseteq \mathcal{SOL}(\mathcal{SMT}(p, m'))$, where $\mathbb{UNSAT} \sqsubseteq \sigma$.*

Informally, this is true because larger match pair sets only allow *more* behavior, which means that the SMT solver has more freedom to find violations, but that all prior violations are still present. However, because of soundness, it is not possible that using a larger match pair set will discover false violations.

6 Experiments and Results

To assess the new encoding in this paper, three experiments with results are presented: a comparison to prior SMT encodings on a zero-buffer semantics, a scalability study on the effects of non-determinism in the execution time on infinite buffer semantics, and an evaluation on typical benchmark programs again with infinite buffer semantics. All of the experiments use the Z3 SMT solver ([12]) and are measured on a 2.40 GHz Intel Quad Core processor with 8 GB memory running Windows 7.

The initial program trace for the experiments is generated using the MCA provided reference solution with fixed input. In other words, the only non-determinism in the programs is that allowed by the MCAPI specification. As such, the experiments only consider one path of control flow through the program. Complete coverage of the program for verification purposes would need to generate input to exercise different control flow paths. Where appropriate, the time to generate the match pair sets from the input trace is reported separately.

6.1 Comparison to Prior SMT Encoding

To our best knowledge, the current most effective SMT encoding for verification of message passing program traces is the order-based encoding that describes the happens-before relation directly in the encoding and is only functional for zero-buffer semantics in its current form [6]. Although the tool to generate the encoding is not publicly available, the authors of the order-based encoding graciously encoded several contrived benchmarks used for correctness testing. These benchmarks are best understood as *toy* examples that plumb the MCAPI semantics to clarify intuition on expected behavior.

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

6.2 Scalability Study

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

Table 1. Scaling as a function of non-determinism

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

The study takes an initial program of $N = 30$, so 31 threads, and varies N to see how the SMT solver scales. A small N is an easy program while a large N is a hard program. Table 1 shows how the new encoding scales with hardness. The first column is the number of messages, or N, and the second column is the number of feasible match pair subsets that correctly match every receive to a unique send. As expected, running time and memory consumption increase non-linearly with hardness.

The case where $N = 70$ represents having 70 concurrent messages in flight from 70 different threads of execution. Such a scenario is not entirely uncommon in a high performance computing application, and it appears the new encoding is able to reasonably scale to such a level of concurrency. The result provides a bound on expected cost for analysis given the message passing behavior in a program. It is expected that the analysis of any program with fewer than 70! possible choices of feasible match pair resolutions will complete in a reasonable amount of time. Regardless, such a high-level concurrency seems unlikely in the embedded space to which MCAPI is targeted.

6.3 Typical Benchmark Programs

The results in the prior section suggest that the number of messages is not the deciding factor in hardness for the new encoding; rather, hardness is measured by the number of feasible match pair sets. This section further explores the observation to show that some programs are easy, even if there are many messages, while other programs are hard, even though there are only a few messages.

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

– *LE* is the leader election problem and is common to benchmarking verification algorithms.
– *Router* is an algorithm to update routing tables. Each router node is in a ring and communicates only with immediate neighbors to update the tables. The program ends when all the routing tables are updated.
– *MultiM* is an extension to an program in the MCAPI library distribution and is similar to the program in Figure 16. The extension adds extra iterations to the original program execution to generate longer execution trace.
– *Pktuse* is a benchmark from the MPI test suite [13]. The program creates 5 tasks—each of which randomly sends several messages to the other tasks.

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

Table 2. Performance on selected benchmarks

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

Table 2 shows the results for the benchmark suite. Other than the metrics used in Table 1, the time of generating the encoding and the match pairs is included in the third and fourth columns respectively. Note that the time shown in the third column includes the time in the fourth column. As before, the running time tracks hardness and not the total number of messages. The table also shows the cost of match pair generation as it dominates the encoding time for the *Pktuse* program. Future work is to address the high-cost of match pair generation, which the authors believe to be NP-complete [15].

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

7 Generating Match Pairs

The exact set of match pairs can be generated by simulating the program trace and using a depth-first search to enumerate non-determinism arising from concurrent sends and receives. Such an effort, however, solves the entire problem at once because if you simulate the program trace exploring all non-determinism, then you may as well verify all runtime choices for property violations at the same time.

In this section, we present an algorithm that does not require an exhaustive enumeration of runtime behavior in simulation. Our algorithm over-approximates the match pairs such that match pairs that can exist in the runtime are all included and some bogus match pairs that cannot exist in the runtime may or may not be included.

The algorithm generates the over-approximated match pair set by matching each pair of the send and receive commands at common endpoints and then pruning obvious matches that cannot exist in any runtime implementation of the specification.

Figure 15 presents the major steps of the algorithm. The algorithm proceeds by first linearly traversing each task of the program storing each receive and send command into two distinct structured lists. The receive list, `list_r`, is structured as in (1) and the send list, `list_s`, is structured as in (2).

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

The list `list_r` groups receives by the issuing endpoint. The integer field merely records the order in which the receives are issued and increases by one on each

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

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

output match_set;
```

Fig. 15. Pseudocode for generating over-approximated match pairs

receive. Similarly, the list `list_s` groups sends first by the destination endpoint and then by the source endpoint. Like `list_r`, an index increases by one to track the issue order. As the input is a program execution trace, any sends or receives in loops already have unique identifiers.

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

Consider the program in Figure 16. The lists `list_r` and `list_s` for the program are

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

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

The sends $\mathtt{S}_{1,1}$, $\mathtt{S}_{1,3}$, and $\mathtt{S}_{2,1}$ have task 0 as an identical destination endpoint. The send $\mathtt{S}_{0,3}$ has task 1 as the destination endpoint. The list `list_s` in (4) reflects this partition. Receive $\mathtt{R}_{0,1}$ is the first receive operation in endpoint 0. This fact is again reflected in `list_r` in (3).

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

Fig. 16. Another MCAPI concurrent program

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

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

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

The generated set of match pairs for our example in Figure 16 is over-approximated by the algorithm because it includes pairs that cannot exist in any feasible execution. For example, the match pair ($\mathtt{S}_{2,1}$ $\mathtt{R}_{0,4}$) is not feasible because it is not possible to order $\mathtt{S}_{1,3}$ before $\mathtt{R}_{0,2}$ since $\mathtt{R}_{1,2}$ can only match with $\mathtt{S}_{0,3}$ that must occur after $\mathtt{R}_{0,2}$. Fortunately, a satisfying solution is only possible using feasible match pairs. Non-feasible match pairs merely result in extra clauses in the encoding and potentially slow down the SMT solver.

The complexity of the algorithm is quadratic. Traversing the tasks to initialize the lists is $O(N)$, where N is the total lines of code of the program. Traversing the list of receives and the list of sends takes $O(mn)$ to complete, where m is the total number of sends and n is the total number of receives. As $m + n \leq N$, the algorithm takes $O(N + mn) \leq O(N + N^2) \approx O(N^2)$ to complete.

8 Related Work

Morse *et al.* provided a formal modeling paradigm that is callable from the C language for the MCAPI interface [11]. This model correctly captures the behavior of the interface and can be applied to model checking C programs that use the API. The work is a direct application of model checking and directly enumerates the non-determinism in the runtime to construct an exhaustive proof. The SMT encoding in this paper pushes that complexity to the SMT solver and leverages recent advances in SMT technology to find a satisfying assignment.

Sharma *et al.* present an dynamic model checker for MCAPI programs built on top of the MCA provided MCAPI runtime [16]. MCC systematically enumerates all non-determinism in the MCAPI runtime under zero-buffer semantics. It employs a novel dynamic partial order reduction to avoid enumerating redundant message orders. This work claims SMT technology is more efficient in practice in resolving non-determinism in a away to violate correctness properties.

Wang *et al.* present an SMT encoding for shared memory semantics for a given input trace from a multi-threaded program [19]. As mentioned previously, the program is partitioned into several concurrent trace programs, and the encoding for each program is verified using SMT technology. Elwakil *et al.* extend the encoding to message passing programs using the MCAPI semantics [5, 6]. The comparison to the encoding in this work is already discussed previously.

An important body of work is being pursued for MPI program verification [17, **?**,**?**,**?**,**?**,18, **?**]. Highlights include an extension to the SPIN model checker for MPI programs, symbolic execution tools for MPI programs including new approaches to computing loop invariants, and various dynamic verification tools for MPI programs. Although MPI is more expressive than MCAPI, the correctness properties in MCAPI are similar to those in MPI. More importantly, the encoding in this work should be applicable to MPI programs that do not include collective operations. An important aspect of future work is to extend the encoding to collectives.

There is a rich body of literature for SMT/SAT based Bounded Model Checking. Burckhardt *et al.* exhaustively check all executions of a test program by translating the program implementation into SAT formulas [1]. The approach relies on counter-examples from the solvers the refine the encoding. The SMT encoding in this work is able to directly resolve the match-pair set over-approximation directly without needing to check a counter-example.

Dubrovin *et al.* give a method to translate an asynchronous system into a transition formula over three partial order semantics [3]. The encoding adds constraints to compress the search space and decrease the bound on the program unwinding. The encoding in this paper operates on a program execution and does not need to resolve a bound.

Kahlon *et al.* presented a partial order reduction, *MPOR*, that operates in the bounded model checking space [7]. It guarantees that exactly one execution is calculated per each Mazurkiewicz trace to reduce the search space. It would be interesting to see if MPOR is able to extend to message passing semantics.

Other work in bounded model checking explores heap-manipulating programs and challenges in sequential systems code [8, ?].

The application of static analysis is another interesting thread of research to test or debug message passing programs with some work in the MPI domain [20, ?,?]. The work is important as it lays the foundation for refining match-pair sets to only include those that cannot be statically pruned.

9 Conclusions and Future Work

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

The paper further defines an algorithm with $O(N^2)$ time complexity to over-approximate the true set of match pairs, where N is the total number of code lines of the program. A comparison to prior work, [6], for a set of "toy" examples under zero-buffer semanics shows the new encoding capable and efficient in capturing correct behaviors of an MCAPI program execution. Experiments further show that the encoding scales to programs with significant levels of non-determinism in how sends are match to receives.

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

Finally, given the importance of high performance computing, future work looks to extend the encoding to account for MPI collective operations. This direction is motivated by the results where the encoding seems to scale to significant levels of concurrency. It should be possible to express MPI collectives as additional constraints in the encoding and apply the technique to MPI programs directly.

References

1. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: Checking consistency of concurrent data types on relaxed memory models. In: ACM SIGPLAN PLDI. San Diego, California, USA (June 10–13, 2007)
2. Cantin, J.F., Lipasti, M.H., Smith, J.E.: The complexity of verifying memory coherence and consistency. IEEE Trans. Parallel Distrib. Syst. 16(7), 663–671 (Jul 2005), http://dx.doi.org/10.1109/TPDS.2005.86
3. Dubrovin, J., Junttila, T., Heljanko, K.: Exploiting step semantics for efficient bounded model checking of asynchronous systems. In: Science of Computer Programming. pp. 77(10–11):1095–1121 (2012)
4. Dutertre, B., de Moura Leonardo: A fast linear-arithmetic solver for DPLL(T). In: CAV. vol. 4144 of LNCS, pp. 81–94. Springer-Verlag (2006)
5. Elwakil, M., Yang, Z.: CRI: Symbolic debugger for MCAPI applications. In: Automated Technology for Verification and Analysis (2010)
6. Elwakil, M., Yang, Z.: Debugging support tool for mcapi applications. In: PADTAD '10: Proceedings of the 8th Workshop on Parallel and Distributed Systems (2010)
7. Kahlon, V., Wang, C., Gupta, A.: Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In: ACM CAV. pp. 398–413. Springer, Berlin/Heidelberg, Grenoble, France (June 26–July 02, 2009)
8. Lahiri, S., Qadeer, S.: Back to the future revisiting precise program verification using SMT solvers. In: POPL. ACM, San Francisco, California, USA (2008)
9. MCA: The multicore association, http://www.multicore-association.org
10. MCA: The multicore association resource management API, http://www.multicore-association.org/workgroup /mcapi.php
11. Morse, E., Vrvilo, N., Mercer, E., McCarthy, J.: Modeling asynchronous message passing for C program. In: Verification, Model Checking, and Abstract Interpretation. vol. 7148 of LNCS, pp. 332–347. Springer-Verlag (2012)
12. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
13. MPPTest: MPPTest benchmark, http://www.mcs.anl.gov /research/projects/mpi/mpptest/
14. Netzer, R., Brennan, T., Damodaran-Kamal, S.: Debugging race conditions in message-passing programs. In: ACM SIGMETRICS Symposium on Parallel and Distributed Tools. pp. 31–40. Philadelphia, PA, USA (1996)
15. Sharma, S.: Private conversation on active research.
16. Sharma, S., Gopalakrishanan, G., Mercer, E., Holt, J.: MCC - a runtime verification tool for MCAPI user applications. In: FMCAD (2009)
17. Siegel, S.F., Zirkel, T.K.: Loop invariant symbolic execution for parallel program. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012. Lecture Notes in Computer Science, vol. 7148, pp. 412–427. Springer (2012)
18. Vakkalanka, S., Vo, A., Gopalakrishnan, G., Kirby, R.: Reduced execution semantics of MPI: From theory to pratice. In: FM. pp. 724–740 (2009)
19. 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)
20. Zhang, Y., Evelyn, D.: Barrier matching for programs with textually unaligned barriers. In: PPoPP. pp. 194–204. ACM, San Jose, California, USA (2007)