Improving the Efficiency of Deadlock Detection in MPI Programs through Trace Compression

Yu Huang, Tao Wang, Zihui Yin, Eric Mercer, Benjamin Ogles

Abstract—This paper presents a static deadlock analysis for single-path MPI programs. Deadlock is when processes are block indefinitely by a circular communication dependency. A single path program is one that does not decode messages for control flow. The analysis records a program execution in the form of a trace and then determines from that trace whether there exists any feasible deadlocking schedules. The primary contribution is the combining of identical consecutive sends or receives into single macro actions. This simplified trace is analyzed for potential deadlock cycles. An abstract machine identifies infeasible cycles, and those not identified by the machine are encoded as satisfiability problems for an SMT solver to resolve. The action combination reduces the complexity of identifying and filtering cycles before needing the costly SMT solver. This paper shows the effectiveness of the action combination in experiments on a benchmark suite comparing to traces without action combination and other state-of-the-art deadlock analyses.

Index Terms—Static Analysis, MPI, SMT

1 Introduction

The *message passing interface* (MPI) is the *de facto* standard for communication and synchronization in high performance distributed programs. MPI programs contain a finite set of *endpoints* that send and receive messages concurrently. A *single-path* MPI program is one where the order of actions issued by each endpoint is deterministic for a given input. A common error in MPI programs, referred to as *deadlock*, occurs when one or more endpoints block indefinitely due to a circular communication dependency. Deadlock discovery for a *single-path* MPI program is proved to be an NP-Complete problem [1]. This paper presents a new static deadlock analysis that solves the problem in an efficient way.

The analysis is an optimization to the predictive analysis in [2], and as such, this paper follows that presentation making clear where it deviates from the original work. The work in [2] statically detects deadlock in *concurrent trace programs* (CTPs). A CTP is an observed execution of an MPI program that records the sequences of send, receive, and blocking actions in each endpoint.

The static deadlock analysis in [2] examines the CTP to identify potential *deadlock cycles* that might exist in other execution orders that are allowed by the MPI semantics. A deadlock cycle is a sequence of actions that form a circular communication dependency and thus never complete in runtime. The analysis identifies such candidate deadlock cycles by constructing and then searching a dependency graph from the CTP. Each candidate deadlock cycle is then used as a target state in an abstract machine that runs the CTP with abstract semantics. The machine takes linear time and space to identify those candidates that may exist in some feasible execution schedule in the original MPI programs. Besides the low time complexity, the machine is able to filter out most infeasible candidates, and thus acts as the most significant step for achieving better performance of the whole analysis. Potential feasible deadlock cycles are then proved to be reachable, or not, in the original single-path MPI program with a novel encoding as a satisfiability problem that can be dispatched to an SMT solver. The time cost of solving the satisfiability problem is dependent on the number of variables and clauses in the encoding, which, in this case, is about quadratic in the number of actions of the program.

The work in [2] includes a proof that the static deadlock analysis is sound: meaning that if no deadlock candidates are found, then no deadlock cycle exists in the original single-path MPI program. The work includes experiments in a benchmark set. The results of those experiments show that the static deadlock analysis is effective in only generating a small set of candidates, if any, that need to be verified with a satisfiability problem. It further shows that the static deadlock analysis is significantly faster, and more efficient, than existing state-of-the-art deadlock analysis tools. The experiments include a set of CTPs for which the static deadlock analysis completes while all the other tools timeout.

Although [2] is more efficient at deadlock detection than other state-of-the-art tools, it suffers from state explosion in the cycle analysis on the dependency graph from the CTP. The dependency graph constructed from a CTP creates bidirectional edges between send and receive pairs with common endpoints to represent viable pairings for message communication. *Wildcard* receives multiply such edges as these accept messages from any sending source endpoint to the destination endpoint. The final dependency graph thus over-approximates what is feasible in any MPI runtime and is thereby sound in the cycle analysis.

The time complexity of cycle analysis is $O((n+e)(c+1))$

- *Yu Huang, Tao Wang and Zihui Yin are with the School of Information and Engineering, Southwestern University of Finance and Economics, Chengdu, China. Email: {yuhuang, ccnuwt}@swufe.edu.cn, yinzihui98@gmail.com.*
- *Eric Mercer and Benjamin Ogles are with the Computer Science Department, Brigham Young University, Prove, USA. Email: egm@cs.byu.edu, benjaminogles@gmail.com.*

where n is the number of nodes in the graph, e is the number of edges, and c is the number of cycles. Every added edge increases the complexity of the algorithm and more so since adding one edge can create many new cycles. This problem is significant for a large program instance since c can be unexpectedly massive thus making the analysis halt in cycle detection.

1.1 Contributions

This work reduces the cost of cycle detection by reducing n, e, and c in the dependency graph by introducing a new macro action that combines a consecutive group of actions in one endpoint. These consecutive actions in one endpoint is able to send consecutive messages to the destination endpoint, or receive consecutive messages from the source endpoint, which is common in any typical communication pattern of MPI benchmarks. A dependency graph defined over CTPs with macro actions inherently has fewer nodes, edges, and thereby cycles when compared to dependency graphs from the original CTP. Cycle detection in the reduced dependency graph is shown to be sound with respect to finding deadlock cycles and significantly more efficient in finding those cycles when compared to its unsimplified counterpart.

Combining the actions in the original CTP into a set of macro actions can take extra pre-processing in analysis. The paper gives a linear algorithm for action combination, so the overhead is only a little bit while the CTP can be compressed to the utmost extent.

Further, the use of macro actions in the compressed CTP reduces the SMT encoding as the prefix matching of macro actions is sufficient to prove or disprove the deadlock state exists. This is done by letting the SMT encoding ignore many send and receive actions that are not in the prefixes of macro actions, and reduce the number of variables and clauses that were used for precisely matching original actions but are no longer needed for the new macro actions, leading to runtime improvements in the backend solver.

In order to test the scalability of our improved deadlock analysis, this paper expands the benchmarks from [2] to those with over hundreds of processes, and compares the performance with several state-of-the-art tools. The experiments show that our new analysis completes all tests shortly while the other tools time out. The effectiveness of the improved analysis is also evaluated by comparing with the old analysis.

In summary, the main contributions include,

1) the new macro actions that compress the CTP directly with fewer actions and thus reducing the dependency graph,
2) an efficient algorithm that combines eligible consecutive actions into fewer macro actions thus compressing the CTP to the utmost extent,
3) the new SMT encoding that reduces the cost of proving a candidate feasible by matching prefixes of macro actions to enter the deadlock state, and
4) the empirical tests showing our approach with action combination scales to hundreds of processes and outperforms other state-of-the-art tools in runtime.

$$
\begin{array}{lcl}
\mathrm{a} & ::= & \mathrm{nb} \mid \mathrm{bb} \\
\mathrm{nb} & ::= & (\mathbf{s}\ i\ p\ \ \mathtt{src}\ \ \mathtt{dst}\ \ \mathrm{n}) \\
 & \mid & (\mathbf{r}\ i\ p\ \ \mathtt{src}\ \ \mathtt{dst}\ \ \mathrm{n}) \\
\mathrm{bb} & ::= & (\mathbf{w}\ i\ p\ i) \\
 & \mid & (\mathbf{b}\ i\ p\ g) \\
\mathtt{src} & ::= & p \mid * \\
\mathtt{dst} & ::= & p
\end{array}
$$

Fig. 1. Types of Actions $\mathrm{a} \in \mathcal{A}$.

1.2 More Changes

The macro actions necessitated changes in the abstract machine that detects infeasible, or unreachable cycles. Those changes are somewhat direct and expected but merit proper documentation as given here. The proof that the abstract machine is sound is updated accordingly and stated in the full version of paper [3] because of page limit. The time and space complexity of the abstract machine does not change with the addition of macro actions.

1.3 Limitation

A threat to validity in this research and its results is that the benchmark suite is not representative of MPI program paradigms in general or that the restriction to single-path programs makes it hard to say anything about arbitrary input. There is no easy way to argue either side of these critiques. The authors have sought to include benchmark programs from all relevant related work and other programs from public repositories. The efficacy of the macro actions depends on size and frequency of consecutive identical actions. The more consecutive actions are combined, the more efficient the new analysis can be. Even for a program with less or no combined actions, the analysis is still more efficient than the existing state-of-the-art tools because of the new satisfiability encoding. Single-path programs represent behavior on a given input. Multiple inputs would need to be considered to cover all branching behavior in an MPI program which is a difficult problem in and of itself.

The formalism and notation from [2] is used for background and context here. As an aid to the reader, each section opens with an explanation of what is new. Additionally, quotation marks, "...", delineate text that is included verbatim with only minor edits from [2].

2 Concurrent Trace Program

This section specifies the syntax and semantics of concurrent trace program with the new macro actions. An example of CTP is shown for its behavior with/without compression and how to detect deadlock by our approach. Finally, a linear algorithm is given for combining the eligible consecutive actions in a program to the macro actions.

2.1 Syntax

We formalize an observed MPI program execution as a concurrent trace program (CTP). A CTP is a set of communication actions $A \subset \mathcal{A}$ where $\mathcal{A}$ is the set of all possible MPI actions. The syntax and structure of actions in $\mathcal{A}$ is shown in Fig. 1. It differs from [2] in the addition of counters for the send and receive actions to use in the trace compression.

Each action has a unique identifier i to index its position in the owning endpoint $p \in P(A)$ where $P(A)$ is the set of all endpoints in the CTP A. For a set of actions X and for each endpoint $p \in P(X)$, $X_p \subseteq X$ is the projection of the actions in X onto the endpoint p.

Non-blocking actions are used to asynchronously send (**s**) and receive (**r**) a number of messages between endpoints. Such actions can be singleton, one message, or a macro with n indicating the number of consecutively issued actions with the same source and destination endpoints. The source (src) and destination (dst) endpoints for a message are indicated by endpoint identifiers in the body of these actions where src can be $*$ to allow for *wildcard* receive. A wildcard receive accepts messages from any source endpoint.

Messages in an MPI program are held in buffers allocated by the user program and copied from source to destination buffers by the MPI runtime. The message buffers are not part of the CTP because the CTP is only the communication observed on a single-path of the original MPI program on some input. The analysis in this paper assumes that that observed communication is deterministic meaning that the same messages are sent and received whenever that same input is given. In this way the analysis considers different schedules on that communication to identify schedules that lead to deadlock.

Blocking actions are used to halt an endpoint until some condition becomes true, possibly synchronizing two or more endpoints. The wait (**w**) action blocks until the message buffer of a non-blocking action is available. Even though message buffers are not modeled in the CTP, the wait actions are needed to indicate the point where its corresponding actions are complete vis-a-vis the issuing endpoint. The barrier (**b**) action blocks until each endpoint in a group (indicated with a unique identifier g) has reached the same barrier. Blocking send and receive actions are not included in $\mathcal{A}$ because they are accurately modeled by placing a wait action directly after the non-blocking action.

For simplicity in the presentation of the analysis, we assume that the last action in each endpoint $p \in P(A)$ is a barrier action ($\mathbf{b}\ i_p\ p\ g$) for some group $g \in G(A)$ where $G(A)$ is the set of barrier groups in A. This assumption does not affect the programs we analyze as we can always extend the observed CTP with these actions. The added barrier action at the end of each endpoint is necessary to model all potential deadlocks in the dependency graph including those arising from wildcard receives.

The CTP can be constructed by the original send and receive actions with $\mathrm{n} = 1$, or it can be further compressed by combining these actions into new send and receive actions with $\mathrm{n} > 1$. The rules of action combination is given in Section 2.4. The analysis in this paper is more efficient and scalable for a compressed CTP that combines consecutive identical actions, but both the uncompressed and compressed CTPs can be correctly analyzed.

The disjoint sets of send, receive, wait, and barrier actions in A are denoted respectively by $S(A)$, $R(A)$, $W(A)$ and $B(A)$. For an action $a \in A$, $id(a)$ and $pid(a)$ denote the action and endpoint identifiers of a.

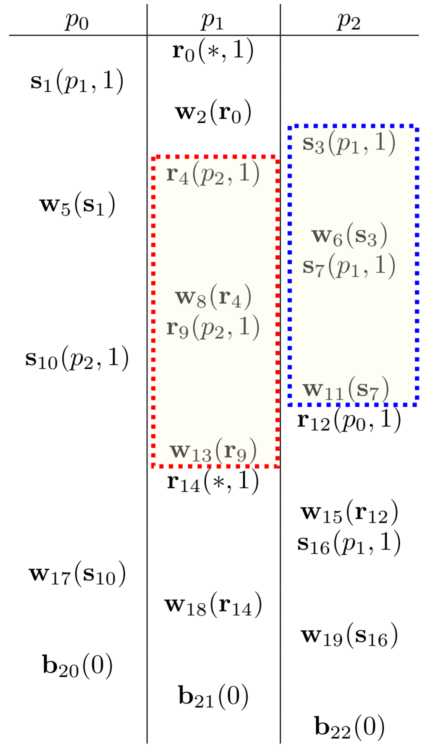


Fig. 2. Example CTP with hidden deadlock.

For an action $a \in S(A) \cup R(A)$, $src(a)$ and $dst(a)$ denote the source and destination endpoint of a. $src(a) = pid(a)$ and $dst(a) = pid(a)$ for sends and receives respectively. The notation $n(a)$ denotes the number of actions that are combined into a. For an action $a \in W(A)$, $req(a)$ is the non-blocking action that a waits for. For an action $a \in B(A)$, $g = grp(a)$ is the group identifier for a and $B_g(A)$ is the set of actions in that group.

2.2 Example

The example here is different from that in [2]. It uses a more clear and compact visualization of illustrative CTP programs as seen in Fig. 2. Specifically, the action IDs are subscripts on the action types while the issuing endpoint of all actions, source endpoint of a send action, and the destination endpoint of a receive actions are all conveyed visually through columniation of the endpoints. The vertical spacing in the figure represents the observed total order of actions in the CTP.

In the example shown in Fig. 2, the wildcard receive $\mathbf{r}_0$ can match with the sends $\mathbf{s}_1$ or $\mathbf{s}_3$. In the observed execution, $\mathbf{r}_0$ matches with $\mathbf{s}_1$, leaving the receives $\mathbf{r}_4$ and $\mathbf{r}_9$ to match with the sends $\mathbf{s}_3$ and $\mathbf{s}_7$ respectively. If however the message race is resolved by matching $\mathbf{r}_0$ with $\mathbf{s}_3$, then a deadlock occurs.

The improved analysis in this paper reduces the example CTP by combining eligible consecutive actions within a process. The reduced CTP in Fig. 3 replaces the four actions in p_1 with two actions, $\mathbf{r}_4(p_2, 2)$ and $\mathbf{w}_8(\mathbf{r}_4)$ and the four actions in p_2 with two actions, $\mathbf{s}_3(p_1, 2)$ and $\mathbf{w}_{11}(\mathbf{s}_3)$. It then builds a dependency graph for the reduced CTP to find deadlock candidates.

p_0	p_1	p_2
	$\mathbf{r}_0(*, 1)$	
$\mathbf{s}_1(p_1, 1)$		
	$\mathbf{w}_2(\mathbf{r}_0)$	
		$\mathbf{s}_3(p_1, 2)$
	$\mathbf{r}_4(p_2, 2)$	
$\mathbf{w}_5(\mathbf{s}_1)$		
$\mathbf{s}_{10}(p_2, 1)$		
		$\mathbf{w}_{11}(\mathbf{s}_3)$
		$\mathbf{r}_{12}(p_0, 1)$
	$\mathbf{w}_{13}(\mathbf{r}_4)$	
	$\mathbf{r}_{14}(*, 1)$	
		$\mathbf{w}_{15}(\mathbf{r}_{12})$
		$\mathbf{s}_{16}(p_1, 1)$
$\mathbf{w}_{17}(\mathbf{s}_{10})$		
	$\mathbf{w}_{18}(\mathbf{r}_{14})$	
		$\mathbf{w}_{19}(\mathbf{s}_{16})$
$\mathbf{b}_{20}(0)$		
	$\mathbf{b}_{21}(0)$	
		$\mathbf{b}_{22}(0)$

Fig. 3. Compressed CTP from Fig. 2.

$$\mathbf{s}_1 \rightarrow \mathbf{w}_5 \rightarrow \mathbf{s}_{10} \rightarrow \mathbf{r}_{12} \rightarrow \mathbf{w}_{15}$$
$$\rightarrow \mathbf{s}_{16} \rightarrow \mathbf{r}_4 \rightarrow \mathbf{w}_{13} \rightarrow \mathbf{r}_{14} \rightarrow \mathbf{s}_1$$

Fig. 4. Cycle for deadlock in Fig. 3.

The graph for Fig. 3 contains 22 nodes and 69 edges, versus the 26 nodes and 97 edges for that in Fig. 2. Our algorithm detects three deadlock candidates and filters away two that are provably infeasible, while there are eight candidates identified for the original CTP. Obviously, the reduced graph by action combination outputs fewer potential candidates after cycle detection. For example, the actions $\mathbf{s}_3$ and $\mathbf{s}_7$ are combined into one action and that one action appears in an infeasible deadlock candidate involving $\mathbf{b}_{21}$, whereas the original graph would generate two infeasible candidates with the same blocking action since it considers both send actions separately.

After filtering away infeasible candidates, our analysis gives the cycle in Fig. 4 as a potential deadlock. A deadlock candidate itself is a characterization of the deadlock involving only the blocking actions on the cycle: $\{\mathbf{w}_5, \mathbf{w}_{13}, \mathbf{w}_{15}\}$. This candidate is encoded as an SMT problem to see if it exists in some real execution.

p_0	p_1	p_2
		$\mathbf{s}_3(p_1, 2)$
	$\mathbf{r}_0(*, 1)$	
	$\mathbf{w}_2(\mathbf{r}_0)$	
	$\mathbf{r}_4(p_2, 2)$	
		$\mathbf{w}_{11}(\mathbf{s}_3)$
$\mathbf{s}_1(p_1, 1)$		
		$\mathbf{r}_{12}(p_0, 1)$
$\mathbf{w}_5(\mathbf{s}_1)$		
	$\mathbf{w}_{13}(\mathbf{r}_4)$	
		$\mathbf{w}_{15}(\mathbf{r}_{12})$

Fig. 5. Witness execution for deadlock in Fig. 4.

Issue-Send-Recv Transition

$$\frac{a \in S(A) \cup R(A) \setminus I \quad <_{po}^{-1}[a] \subseteq I \\ (<_{po}^{-1}[a] \cap (W(A) \cup B(A))) \subseteq M \quad a' = a[0]}{\langle A, I, M\rangle \rightarrow \langle A, I \cup \{a\}, M \cup \{a'\}\rangle}$$

Issue-Other Transition

$$\frac{a \in B(A) \cup W(A) \setminus I \quad <_{po}^{-1}[a] \subseteq I \\ (<_{po}^{-1}[a] \cap (W(A) \cup B(A))) \subseteq M}{\langle A, I, M\rangle \rightarrow \langle A, I \cup \{a\}, M\rangle}$$

Match-Send-Recv Transition

$$\frac{a, a' \in I \setminus M \quad a \in S(A) \quad a' \in R(A) \quad dst(a) = dst(a') \\ src(a') \in \{src(a), *\} \quad (<_{mo}^{-1}[a] \cup <_{mo}^{-1}[a']) \subseteq M \\ a_m, a'_m \in M \quad a_m[0] = a[0] \quad a'_m[0] = a'[0]}{\langle A, I, M\rangle \rightarrow \\ \langle A, I, M \setminus \{a_m, a'_m\} \cup \{a_m[n(a_m)+1], a'_m[n(a'_m)+1]\}\rangle}$$

Match-Wait Transition

$$\frac{a \in I \setminus M \quad a \in W(A) \quad <_{mo}^{-1}[a] \subseteq M}{\langle A, I, M\rangle \rightarrow \langle A, I, M \cup \{a\}\rangle}$$

Match-Barrier Transition

$$\frac{g \in G(A) \quad \forall a \in B_g(A), a \in I \wedge <_{mo}^{-1}[a] \subseteq M}{\langle A, I, M\rangle \rightarrow \langle A, I, M \cup B_g(A)\rangle}$$

Fig. 6. Transition Rules for Concrete Semantics.

The encoding for the candidate in Fig. 4 asks the SMT solver to find a state where $\mathbf{w}_5$, $\mathbf{w}_{13}$ and $\mathbf{w}_{15}$ are issued and where $\mathbf{s}_1$, $\mathbf{r}_4$ and $\mathbf{r}_{12}$ cannot be matched. Fig. 5 shows the witness execution from the satisfying assignment discovered by the solver. Here the send $\mathbf{s}_3$ is issued, and since there are two sends from $\mathbf{s}_3$, it matches the single receive of $\mathbf{r}_0$ and one of the two receives from $\mathbf{r}_4$. These can only match on messages from p_2. That leaves an outstanding receive from $\mathbf{r}_4$ still pending so the process is blocked on $\mathbf{w}_{13}$ waiting for that future send. The send $\mathbf{s}_1$ is issued and p_1 is then blocked on $\mathbf{w}_5$. The receive $\mathbf{r}_{12}$ is issued and then blocked on $\mathbf{w}_{15}$. No more actions can issue, and none of the pending send and receives in the runtime match. The CTP is deadlock.

2.3 Semantics

The semantics in [2] are extended here to define the behavior of the macro send and receive actions with the added counter in the syntax. The changes are localized in two of the semantic rules, **Issue-Send-Recv** and **Match-Send-Recv**, discussed later. The following background is verbatim from the prior work.

"The semantics of A is given by a finite state machine $\mathcal{F}(A) = \langle \mathcal{Q}, q_0, \rightarrow\rangle$ where $\mathcal{Q} \subseteq 2^{\mathcal{A}} \times 2^{\mathcal{A}} \times 2^{\mathcal{A}}$ is the set of states, $q_0 = \langle A, \emptyset, \emptyset\rangle$ is the start state and $\rightarrow \subseteq \mathcal{Q} \times \mathcal{Q}$ is the transition relation. In a state $q = \langle A, I, M\rangle$, A is the set of actions in the CTP, $I \subseteq A$ is the set of actions that have been issued to the runtime, and $M \subseteq \mathcal{A}$ is the set of actions that have been *matched*.

For an endpoint $p \in P(A)$, the actions owned by p are always issued sequentially. This constraint can be captured as a partial order over A if we assume that action identifiers were assigned in ascending order while observing the original execution.

Definition 1 (Endpoint order). Endpoint order is a partial order $(A, \leq_{po}^{A})$ where

$$\forall a, a' \in A : a \leq_{po}^{A} a' \iff pid(a) = pid(a') \wedge id(a) \leq id(a')$$

For Definition 1 and similarly for any other partial order defined as follows, we use $<_{po}^{A}$ to mean the partial order with reflexivity removed and we omit A from the notation when it is clear from context.

Send and receive actions are matched together by the runtime when their source and destination endpoints are compatible. Wait actions do not match other actions but are instead "matched" with themselves after their associated message request is matched. Finally, barrier actions match the other actions in their group when they have all been issued.

Some actions in the same endpoint may be matched in an order different from how they were issued while others must be matched in the order they were issued. This constraint is captured by two more partial orders.

Definition 2 (Queue order). Queue order is a partial order $(A, \leq_{qo}^{A})$ where for all actions $a, a' \in A$, $a \leq_{qo}^{A} a'$ if and only if $a \leq_{po}^{A} a'$ and one of the following is true:

1) $\{a, a'\} \subseteq S(A) \wedge dst(a) = dst(a')$
2) $\{a, a'\} \subseteq R(A) \wedge src(a) \in \{src(a'), *\}$

Definition 2 defines a first-in-first-out (FIFO) ordering over messages communicated on the same endpoint. With one exception, this order fully supports the "non-overtaking" property of ordered messages as defined in the MPI standard. The exception occurs when a deterministic receive is followed by a wildcard receive in the same endpoint.

In this case, FIFO ordering over the two actions is enforced only if they can both match the same send action. This condition is schedule-dependent as its value changes depending on whether a send action has been issued that can match the first action when the second action is issued. As in [1], we leave such receive actions unordered.

Definition 3 (Match order). Match order is a partial order $(A, \leq_{mo}^{A})$ where for all actions $a, a' \in A$, $a \leq_{mo}^{A} a'$ if and only if $a \leq_{po}^{A} a'$ and one of the following is true:

1) $a \leq_{qo}^{A} a'$
2) $a \in W(A) \cup B(A)$
3) $a \in S(A) \cup R(A) \wedge a' \in W(A) \wedge a = req(a')$

Definition 3 ensures that (1) queue order is preserved when messages are matched, (2) blocking actions are matched before subsequent actions in the same endpoint and (3) message requests are matched before their associated wait actions. With match order defined, we can define the transition relation $\rightarrow$ shown in Fig. 6. Given a relation Q over a set X and an element $x \in X$, $Q^{-1}[x] = \{y \in X : yQx\}$ is the preimage of x under Q."

The issue of a new action a is completed by the **Issue-Send-Recv** or **Issue-Other** transitions when all of the actions preceding a in the same endpoint have been issued and all of the blocking actions preceding a in the same endpoint have been matched. Each transition updates the state by putting a into the set I. If a is a send or receive action, the **Issue-Send-Recv** transition additionally creates an action a', which is identical to a except that n in its body is initialized to 0. This rule is different from the original semantics in [2], under which no extra action is created for a in either issuing or matching steps. The addition of a' makes sure that the semantics defined here can record how a is partially matched if a is a macro action. Note a macro action can not be completely matched in one step of matching. For that purpose, the action a' itself has no meaning to the CTP execution, but when it is updated in the set M by incrementing the number n iteratively, the action a can be matched gradually. To help presenting the step of matching send and receive actions, let the action $a[n]$ be defined as follows.

$$a[n] = \begin{cases} (\mathbf{s}\ id(a)\ pid(a)\ src(a)\ dst(a)\ n) & a \in S(A) \\ (\mathbf{r}\ id(a)\ pid(a)\ src(a)\ dst(a)\ n) & a \in R(A) \end{cases}$$

The **Match-Send-Recv** transition completes type compatible matches for the actions $a \in R(A)$ and $a' \in S(A)$ when their match order dependencies have been satisfied. Additionally, the associated actions a_m and a'_m are updated in the set M by incrementing the number n in each of their bodies by one at a time, indicating that a send and a receive combined in a and a' respectively are matched. Eventually, the action a is completely matched if the number n in a_m reaches $n(a)$ after all the **Match-Send-Recv** transitions that a takes part in are completed. In other words, the structure of action a_m is the same with a at this moment. Note matching a' needs similar iterations.

A complete match is made by the **Match-Wait** transition. The rule forces executions to follow a *rendevous protocol*. Rendevous protocol synchronizes the sender and receiver by blocking both until the message transfer is completed.

2.4 Compression by Action Combination

Action combination is only adaptable to a set of consecutive sends or a set of consecutive receives in an identical endpoint. More precisely,

Definition 4 (Action Combination). Two actions a and a' such that,

1) $pid(a) = pid(a') \wedge src(a) = src(a') \wedge dst(a) = dst(a')$, and
2) $id(a) < id(a') \wedge \forall a'' \in A : id(a) < id(a'') \wedge id(a'') < id(a') \Rightarrow a'' \in W(A)$,

can be combined, denoted as $Comb(a, a', A)$, to yield a new action

$$a_c = \begin{cases} (\mathbf{s}\ id(a)\ pid(a)\ src(a)\ dst(a)\ (n(a) + n(a'))) & a, a' \in S(A) \\ (\mathbf{r}\ id(a)\ pid(a)\ src(a)\ dst(a)\ (n(a) + n(a'))) & a, a' \in R(A) \end{cases}$$

and a new CTP

$$A' = A \cup \{a_c\} \cup \{\mathbf{w}_{(id(a_c)+1)}(pid(a_c))id(a_c)\} \setminus (\{a, a'\} \cup \{w \in W(a) : req(w) = a \vee req(w) = a'\})$$

The combination in Definition 4 works for any pair of two sends or two receives if their endpoints match and they are consecutively ordered in an identical endpoint. The new action a_c and its associated wait are created to replace the original actions and their waits, and consequently updates the CTP.

Given the basic steps of action combination in Definition 4, a CTP can be compressed by the operator $\xrightarrow{Comb}$.

Definition 5 (Compressing a CTP with Action Combination). The operator $\xrightarrow{Comb}$ is a relation on $2^{\mathcal{A}} \times 2^{\mathcal{A}}$ such that $\xrightarrow{Comb} = \{< A \subset \mathcal{A}, A' \subset \mathcal{A} >: \exists a, a' \in A, \exists a_c \in A', \{a_c, A'\} = Comb(a, a', A)\}$, where $\xrightarrow{Comb^*}$ denotes the transitive closure of $\xrightarrow{Comb}$.

This paper uses $A \xrightarrow{Comb} A'$ to represent $< A, A' > \in \xrightarrow{Comb}$. Additionally, any CTP A can be applied for an utmost compression in Definition 6 by multiple iterations of action combination.

Definition 6 (Utmost Compression). For any CTP A, there exists an utmost compressed CTP A_f such that

$$A \xrightarrow{Comb^*} A_f \wedge \nexists A'_f \subseteq \mathcal{A}, A_f \xrightarrow{Comb} A'_f.$$

The CTP A_f is constructed by combining the actions to the utmost extent, which stands for the largest simplification for the execution. The analysis in this paper intends to compress every CTP observed from the execution, A_0 to the CTP A_f, which directly optimizes the process of deadlock detection.

3 DEADLOCK

This section is unchanged from [2] and restated verbatim for convenience as it formalizes the problem statement for deadlock detection in A. The problem statement is defined over a generic transition relation $\delta \subseteq \mathcal{Q} \times \mathcal{Q}$ rather than that in the previous section because the statement is reused in the the proof for the abstract transition relation defined in the full version of paper [3].

"Let $\Sigma_\delta^q \subseteq \mathcal{Q}$ denote the reachable states of A from the state q with respect to a transition relation δ:

$$\Sigma_\delta^q = \{q' \in \mathcal{Q} : (q, q') \in \delta^*\}$$

where δ^* denotes the transitive closure of δ.

Definition 7 (Deadlock). A state $q = \langle A, I, M \rangle$ is deadlocked with respect to a transition relation δ if there are no enabled transitions and there are actions left to be issued or matched:

$$Dead_\delta(q) \iff (I \neq A \vee M \neq A) \wedge (\forall q' \in \mathcal{Q}, (q, q') \notin \delta)$$

The deadlock discovery problem, given in Definition 8, asks whether $\mathcal{F}(A)$ can reach a deadlocked state. This problem is NP-Complete and can be directly encoded as a propositional formula [1].

Definition 8 (Deadlock discovery problem).

$$\exists q \in \Sigma_\delta^{q_0}, Dead_\delta(q)$$

The search for an arbitrary feasible deadlock state can be extremely expensive for many programs. We can give the search a kind of head start by finding a simple way to characterize the types of states that may deadlock. A convenient way to describe a state is by its *control point*. The control point of a state is simply the set of last issued actions from each endpoint:

$$Ctrl(\langle A, I, M \rangle) = \{a \in I : \forall a' \in I_{pid(a)}, a' \leq_{po} a\}.$$

If we only provide the last issued action for a subset of endpoints, we obtain a partial control point that describes the collection of states which include it as a subset of their control points. Definition 9 augments the problem statement in Definition 8 to ask for a deadlock state that matches a partial program point D (also called a deadlock candidate)."

Definition 9 (Constrained deadlock discovery problem).

$$\exists q \in \Sigma_\delta^{q_0}, D \subseteq Ctrl(q) \wedge Dead_\delta(q)$$

4 DEADLOCK DETECTION

This section discusses the major steps of our static deadlock analysis, including the cycle detection from the dependency graph reduced by introducing the new macro actions, the abstract machine with the new semantics, and the SMT encoding that allows prefix matching of macro actions and has fewer variables and clauses leading to runtime improvements in the backend solver.

4.1 Cycle Detection in Dependency Graph

This sub-section defines how to compute a sound set of deadlock candidates $\mathbb{D}(A)$ for A. The dependency graph construction uses the same set of rules as those in [2]. New here is the definition of a deadlock path with macro actions, the set of potential matches for macro actions, and how that set is over-approximated to build the dependency graph. The algorithm for finding potential deadlock cycles is changed accordingly.

"The soundness property according to Definition 9 is formally stated in the following theorem.

Theorem 1 (Deadlock candidates sound). For all states $q \in \Sigma_\rightarrow^{q_0}$, If $Dead_\rightarrow(q)$, then $\exists d \in \mathbb{D}(A), d \subseteq Ctrl(q)$.

We generate $\mathbb{D}(A)$ by detecting cycles in a graph (N, E) where $N = A \cup \{\perp_p : p \in P(A)\}$ is the set of nodes and $E : N \times N$ is the set of edges. The node $\perp_p$ is used to explicitly represent the end of endpoint p in the graph. An edge $(a, a') \in E$ represents a potential communication dependency of a' on a in some execution of A. A proof that our technique satisfies Theorem 1, along with proofs for the other theorems stated in this paper, is given in the full paper [3].

Before presenting the dependency graph, we describe how one of its cycles can represent a deadlock. This not only motivates the rules for adding edges to the graph, but also leads to a precise understanding of the type of cycle the analysis must report and the types it can ignore. This is important because the graphs we build may contain a huge number of cycles which are expensive to enumerate. The more we can ignore, the more efficient our analysis will be.

Fix a deadlock state $q \in \Sigma_{\rightarrow}^{q_0}$ with control point $D = Ctrl(q)$. For each endpoint $p \in P(A)$, there is an action $a \in D$ with $p = pid(a)$. We will define a few new terms that allow us to talk about why a is blocking p from progressing in the state q.

First, we call a the *deadlock action* for p. Next, let a' be the earliest action in p that is issued in q but not matched with $a' \leq_{mo} a$. We call a' the *orphaned action* for p. Finally let a'' be an action that is not issued in q but would allow p to progress if it is matched with a'. We call a'' the *parent action*. If the parent action does not exist, it is represented in the graph by a $\perp$ node (discussed more below). We will use the following definition to translate these concepts to the context of a path of edges in E."

Definition 10 (Deadlock path). Let $a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_{m-1} \rightarrow a_m$ be a path of edges in E. This path is a *deadlock path* for the endpoint $p \in P(A)$ if

1) $pid(a_0) \neq p$ and
2) $pid(a_i) = p$ for all $i \in \{1 \ldots m\}$ and
3) if $m > 1$, $a_i \in W(A) \cup B(A)$ for some $i \in \{1 \ldots m-1\}$ and
4) if $m = 1$, $a_1 \in R(A)$ such that $n(a_1) > 1$ and $src(a_1) = *$.

Definition 10 specifies two typical instances of deadlock path. The first instance is constrained by the conditions (1) – (3), where a_0 and a_1 are interpreted as parent and orphan actions of the endpoint p for a deadlock path $a_0 \rightarrow a_1 \rightarrow \ldots \rightarrow a_{m-1} \rightarrow a_m$ for p. The edge connecting them represents the possibility that a_1 may depend on a_0 being issued and available to match to complete in some execution.

Specially, if $n(a_0) > 1$ or $n(a_1) > 1$, the edge implies more than one (probably a huge number of) possible communication dependencies between the actions combined in a_0 and a_1. The integration of these dependencies to a single edge is helpful to highly reduce the number of detected cycles, while still preserves all possible circular communication dependency between the actions.

The earliest blocking action issued by p and contained in the path is interpreted as the deadlock action of p. The path requires that the deadlock action not to be the last action in the path. This is because a deadlock cycle is constructed by composing the deadlock paths of two or more endpoints. In other words a_m in the deadlock path of p will be the parent action in a different deadlock path for some endpoint $p' \neq p$.

The second instance of deadlock path is enforced by the conditions (1), (2) and (4), where a_1 is the only action in p. Unlike a common understanding of deadlock path, the action a_1 takes part in two feasible connections $a_0 \rightarrow a_1$ and $a_1 \rightarrow a_2$ without inserting any deadlock action. The action a_2 is the orphaned action in some other deadlock path. In this case, the action a_1 can only be a wildcard receive with $n(a_1) > 1$. The actual deadlock action is the very next wait w for a_1 in $pid(a_1)$. w is outside the deadlock path and must exist according to action combination. To let the deadlock path contribute to a feasible deadlock cycle, a_1 has to be either unmatched or prefix matched, meaning not all actions in a_1 are matched, leaving some suffix action orphaned in the associated deadlock cycle. Note that a_1 can not be a send action even if $n(a_1) > 1$, because a send action can not connect with two different endpoints.

Definition 11 (Deadlock cycle). A cycle of edges in E is a *deadlock cycle* if it can be constructed from a set of deadlock paths, with each endpoint contributing at most one path, and the orphaned action of each path must not be a compatible match for the orphaned action of any path in the cycle.

For example, the deadlock cycle in Fig. 4 includes three deadlock paths:

1) $\mathbf{r}_{14} \rightarrow \mathbf{s}_1 \rightarrow \mathbf{w}_5 \rightarrow \mathbf{s}_{10}$,
2) $\mathbf{s}_{10} \rightarrow \mathbf{r}_{12} \rightarrow \mathbf{w}_{15} \rightarrow \mathbf{s}_{16}$, and
3) $\mathbf{s}_{16} \rightarrow \mathbf{r}_4 \rightarrow \mathbf{w}_{13} \rightarrow \mathbf{r}_{14}$,

each of which is contributed by a separate endpoint. The set of deadlock actions $\{\mathbf{w}_5, \mathbf{w}_{15}, \mathbf{w}_{13}\}$ is extracted as a deadlock candidate.

Deadlock cycles are detected by constructing the aforementioned graph and then enumerating the cycles in the graph. The nodes, N, being the actions, A, in the CTP with a special $\perp_p$ node for each endpoint. The edges between nodes are defined by actions that can match, a send with a receive for example, and the match order $\leq_{mo}^{A}$.

Definition 12 (Potential matches). For an action $a \in A$, $\mathbb{M}(a)$ denotes the set of actions where each can be matched with a in at least one transition between reachable states of A. More precisely, if $M(q)$ denotes the matched set in the state q, then

$$\begin{aligned}\mathbb{M}(a) = \{a' : \exists q, q' \in \Sigma_{\rightarrow}^{q_0}, q \rightarrow q' \\ \wedge \{a[m], a'[m']\} \subseteq M(q) \\ \wedge \{a[m+1], a'[m'+1]\} \subseteq M(q') \\ \wedge 0 \leq m < n(a) \wedge 0 \leq m' < n(a')\} \setminus \{a\}.\end{aligned}$$

Computing precisely $\mathbb{M}$ is as hard as the deadlock discovery problem itself. Potential matches are thus over approximated to ensure that the whole analysis is sound. The algorithm for approximating potential matches in [4] is extended here to work with macro send and receive actions. The extension generates a match set, $\mathbb{M}_0$, over the original, uncompressed actions,A_0, from the CTP. It then iteratively creates $\mathbb{M}$ with a mapping from the original actions in $\mathbb{M}_0$.

That mapping, $\phi : A \rightarrow 2^{A_0 \cup A}$, is defined such that for an action a, if it is a send or receive action, it returns the set of actions in A_0 that are combined by a; otherwise, it returns $\{a\}$. Given $\mathbb{M}_0$, computed by [4], and ϕ, then $\mathbb{M}(a) = \{a' \in A : \exists b, b' \in A_0, b' \in \mathbb{M}_0(b) \wedge b \in \phi(a) \wedge b' \in \phi(a')\}$. If two actions matched in $\mathbb{M}_0$, then the actions that cover those also match in $\mathbb{M}$.

Definition 13 (Edges). $\forall (a, a') \in N \times N$, $(a, a') \in E$ if and only if any one of the following holds:

1) $a \in A \wedge a' = \perp_{pid(a)}$
2) $a, a' \in A \wedge a <_{mo} a'$
3) $a, a' \in A \wedge a' \in \mathbb{M}(a)$
4) $a = \perp_{src(a')} \wedge a' \in R(A) \wedge \exists a_r \in R(A)(a_r <_{po} a' \wedge src(a_r) = * \wedge src(a') \neq *)$
5) $a' \in S(A) \wedge \exists a_r \in R(A)(a = \perp_{dst(a_r)} \wedge dst(a_r) = dst(a') \wedge src(a_r) = *)$

Rule one connects every action to the end of its owning endpoint. Rule two encodes the match order from Section 2.3. It connects deadlock actions to those in the tail of

Algorithm 1 Determine whether the edge (v, w) can possibly reach a deadlock cycle starting at s

1: **procedure** DEADLOCKEDGE(v, w, s)
2:     **if** $pid(v) = pid(w)$ **then**
3:         **return** $true$
4:     **if** `all_count`(`stack`, $pid(v)$) $> 1 \wedge$ `block_count`(`stack`, $pid(v)$) $= 0$ **then**
5:         **return** $false$
6:     **if** `block_count`(`stack`, $pid(v)$) $= 1 \wedge v \in W(A) \cup B(A)$ **then**
7:         **return** $false$
8:     **if** `all_count`(`stack`, $pid(v)$) $= 1 \wedge \neg(v \in R(A) \wedge src(v) = * \wedge n(v) > 1)$ **then**
9:         **return** $false$
10:     **if** $\exists a \in$ `orphaned`(`stack`), `can_match`(a, w) **then**
11:         **return** $false$
12:     **if** $w \neq s$ **then**
13:         **return** `all_count`(`stack`, $pid(w)$) $= 0$
14:     **return** $false$

a deadlock path, and it ensures that wait actions (often a deadlock action) have an incoming edge from their associated message request (often an orphaned action).

The rest of the rules ensure that parent actions are connected to orphaned actions in any scenario. The most obvious occurs in rule three when the actions can form a match. The nature of $\mathbb{M}$ adds an edge in each direction to reflect that either a send or a recieve can be the parent action in a deadlock.

Rules four and five encode *message starvation*. Message starvation is when a wildcard receive is matched with a send action that leaves some subsequent deterministic receive without any potential future matches. The parent action does not exist for this type of deadlock as the starved action is waiting for something that can never be issued. The added $\perp$ node at the end of every endpoint takes the place of the parent action in these cases.

As an aside, edges are not added from $\perp$ nodes to wildcard receive actions. Deadlocks arising from starved wildcard receives are deterministic; they manifest on any schedule of the program for the given input.

Definition 13 results in graphs with fewer nodes and edges for the CTP compressed by action combination. The number of cycles, therefore, can be noticeably reduced as any edge connecting two sends or receives with $n > 1$ actually implies more than one possible matches for the original actions, which could largely expand the graph if these actions were not combined. That said, many spurious, infeasible, cycles exist in the graph. For example, the third rule alone creates trivial cycles between matching actions.

Johnson's algorithm for enumerating the elementary cycles of a directed graph [5] is modified to only enumerate cycles that match Definition 11. The algorithm enumerates the strongly connected components in the graph finding the cycles in each one. The order in which it considers strongly connected components is set such that the component containing the next unvisited least node on some total order of $\leq_{po}$ is followed. Within that connected component, s, is the least node, and it is where the cycle search begins. This ordering ensures that s is always the orphaned action of the first deadlock path visited by the algorithm.

From here, the algorithm considers edges that can extend the current path being considered. Algorithm 1 is a boolean function that determines when a deadlock path can or cannot be extended given an edge (v, w) and the starting node s. The reference `stack` is the depth first search stack of visited nodes. The `all_count` and `block_count` functions respectively return the number of all actions and the number of blocking actions in a given endpoint that have been visited by the current stack. The `can_match` function determines whether two actions may form a compatible match based on their types and endpoints. Finally, the `orphaned` function returns the orphaned action from each deadlock path in the current stack.

Lines 2-3 of Algorithm 1 extend the tail of the current path along the same endpoint (rule one in Definition 10). If the path cannot be extended along the same endpoint, then v and w are the parent and orphaned actions for a new deadlock path. Lines 4-7 ensure that the current deadlock path contains a deadlock action that is not also a parent action for w (rules two and three in Definition 10). Lines 8-9 ensure that v has to be a wildcard receive with $n(v) > 1$ when the current deadlock path contains exactly one action for the endpoint $pid(v)$ (rule four in Definition 10). Lines 10-11 check that the cycle cannot unwind by matching some orphaned action (Definition 11). Lines 12-13 ensure that each endpoint only contributes one path to the cycle (also Definition 11). The final case occurs when w is equal to s and a cycle is formed.

As a final optimization not shown in Algorithm 1, duplicate candidates are not reported. The optimization tracks how actions are orphaned. If the tail of a path can reach an orphaned action in another endpoint from two different parents, then only the cycle from one such parent is needed.

4.2 Abstract Machine

This sub-section extends [2] to update the abstract machine semantics for the macro send and receive actions. The changes are made for the issuing and matching transitions.

"The abstract machine $\widehat{\mathcal{F}}(A) = \langle \mathcal{Q}, q_0, \rightarrow_{abs} \rangle$ augments the semantics of CTPs to efficiently filter away infeasible deadlock candidates. This filtering is an important stage in the analysis because it can drastically reduce the number of calls to the SMT solver. The abstract transition relation $\rightarrow_{abs}$ is shown in Fig. 7 with the barrier transition omitted as it is unchanged from $\rightarrow$. In this transition relation we create a dedicated *wildcard endpoint* for each source process. This eliminates the possibility of message starvation."

The **Issue-Send** transition generates a fresh *wildcard send* for the new endpoint and issue it alongside the original send action. Similar to the concrete semantics, the copies of both *wildcard send* and original send actions are added to M for matching. A wait on the send action is allowed to match in the **Match-Wait** transition if the action has been completely matched, indicating the accumulation of the n numbers for the two copies of sends in M reaches that of the original send. The wildcard send is only allowed to match wildcard receive actions issued by the destination process and the original send action is only allowed to match deterministic receive actions.

Issue-Send Transition

$$\frac{\begin{array}{c} a \in S(A) \setminus I \quad <_{po}^{-1}[a] \subseteq I \\ (<_{po}^{-1}[a] \cap (W(A) \cup B(A))) \subseteq M \\ s = (\mathbf{s}\ (id(a)-1)\ pid(a)\ *\ dst(a)\ n(a)) \quad s' = s[0] \\ a' = a[0] \end{array}}{\langle A, I, M\rangle \rightarrow_{abs} \langle A \cup \{s\}, I \cup \{s, a\}, M \cup \{s', a'\}\rangle}$$

Issue-Recv Transition

$$\frac{\begin{array}{c} a \in R(A) \setminus I \quad <_{po}^{-1}[a] \subseteq I \\ (<_{po}^{-1}[a] \cap (W(A) \cup B(A))) \subseteq M \quad a' = a[0] \end{array}}{\langle A, I, M\rangle \rightarrow \langle A, I \cup \{a\}, M \cup \{a'\}\rangle}$$

Issue-Other Transition

$$\frac{\begin{array}{c} a \in B(A) \cup W(A) \setminus I \quad <_{po}^{-1}[a] \subseteq I \\ (<_{po}^{-1}[a] \cap (W(A) \cup B(A))) \subseteq M \end{array}}{\langle A, I, M\rangle \rightarrow_{abs} \langle A, I \cup \{a\}, M\rangle}$$

Match-Send-Recv Transition

$$\frac{\begin{array}{c} a, a' \in I \setminus M \quad a \in S(A) \quad a' \in R(A) \quad dst(a) = dst(a') \\ src(a') = src(a) \quad (<_{mo}^{-1}[a] \cup <_{mo}^{-1}[a']) \subseteq M \\ a_m, a'_m \in M \quad a_m[0] = a[0] \quad a'_m[0] = a'[0] \end{array}}{\begin{array}{c} \langle A, I, M\rangle \rightarrow \\ \langle A, I, M \setminus \{a_m, a'_m\} \cup \{a_m[n(a_m)+1], a'_m[n(a'_m)+1]\}\rangle \end{array}}$$

Match-Wait Transition

$$\frac{\begin{array}{c} a \in I \setminus M \quad a \in W(A) \\ s = (\mathbf{s}\ (id(req(a))-1)\ pid(req(a))\ *\ dst(req(a))\ n(req(a))) \\ \exists a_s, s' \in M, (a_s[0] = req(a)[0]) \wedge (s'[0] = \\ s[0]) \wedge (n(req(a)) = n(a_s) + n(s')) \end{array}}{\langle A, I, M\rangle \rightarrow_{abs} \langle A, I, M \cup \{a\}\rangle}$$

Fig. 7. Transition Rules for Abstract Semantics.

"We filter a deadlock candidate D by deriving a CTP A_D that contains the actions in D, the actions process ordered before actions in D and all of the actions in other processes:

$$A_D = \bigcup_{a \in D} \leq_{po}^{-1}[a] \cup \bigcup_{p \notin P(D)} A_p$$

We then attempt to execute $\widehat{\mathcal{F}}(A_D)$ to determine whether

$$\exists q \in \Sigma^{q_0}_{\rightarrow_{abs}}, D \subseteq Ctrl(q) \wedge Dead_{\rightarrow_{abs}}(q)$$

Note that this is just Definition 9 with the abstract transition relation substituted in. If the abstract execution is able to issue every action in D, then the candidate may represent a real deadlock and it is added to the set of candidates to be encoded as an SMT formula. Otherwise, the candidate is infeasible and is discarded.

The abstract machine is sound if it never discards a reachable control point. Let the set of reachable control points from a state q and a transition relation δ be $\mathbb{C}^q_\delta$.

MATCH ORDER	$\bigwedge_{a \in A_D} \bigwedge_{a' \in <_{mo}^{-1}[a]} (c_a \implies c_{a'}) \wedge t_{a'}^{n(a')} < t_a^1$
	$\bigwedge_{a \in R(A_D) \wedge n(a) > 1} t_a^1 < t_a^{n(a)}$
	$\bigwedge_{a \in S(A_D) \wedge n(a) > 1} \bigwedge_{i \in \{1 \ldots n(a)-1\}} t_a^i < t_a^{i+1}$
QUEUE ORDER	$\bigwedge_{a \in S(A_D) \cup R(A_D)} \bigwedge_{a' \in <_{qo}^{-1}[a]} m_{a'}^{n(a')} < m_a^1$
	$\bigwedge_{a \in R(A_D) \wedge n(a) > 1} m_a^1 < m_a^{n(a)}$
	$\bigwedge_{a \in S(A_D) \wedge n(a) > 1} \bigwedge_{i \in \{1 \ldots n(a)-1\}} m_a^i < m_a^{i+1}$
BARRIERS	$\bigwedge_{a \in B(A_D)} \bigwedge_{a' \in B_{grp(a)}(A_D)} t_a = t_{a'}$
MATCH COUNT	$\bigwedge_{a \in S(A_D)} \bigwedge_{b \in \phi(a)} atm(1, \{\mathtt{M}(a', b) : b \in \mathbb{M}^*_D(a')\})$
	$\bigwedge_{a \in R(A_D)} atm(n(a), \{\mathtt{M}(a, a') : a' \in \mathbb{M}^*_D(a)\})$
MATCH CORRECT	$\bigwedge_{a \in S(A_D)} (c_a \iff \bigwedge_{b \in \phi(a')} exa(1, \{\mathtt{M}(a', b) : b \in \mathbb{M}^*_D(a')\})$
	$\bigwedge_{a \in R(A_D)} (c_a \iff exa(n(a), \{\mathtt{M}(a, a') : a' \in \mathbb{M}^*_D(a)\})$
REACH	$\bigwedge_{a \in D} \bigwedge_{a' \in <_{mo}^{-1}[a] \setminus O} c_{a'}$
DEADLOCK	$\bigwedge_{a \in D \cup O} \neg c_a$
NO MATCHES	$\bigwedge_{a \in O} \bigwedge_{a' \in \mathbb{M}^*_D(a) \setminus (O \cup D)} c_{a'}$

Fig. 8. Constraints in the formula F

$$\mathbb{C}^q_\delta = \{Ctrl(q') : q' \in \Sigma^q_\delta\}$$

Theorem 2 states that the reachable control points of the abstract machine subsumes the reachable control points of the concrete machine. Theorem 3 states that if the abstract machine cannot issue every action in the deadlock candidate in one execution, it will not be able to issue them all in any execution. Together these theorems prove that the candidate D can be filtered away in a single execution of the abstract machine when it fails to issue all of the actions in D. The complete proofs of Theorem 2 and Theorem 3 are given in the full paper [3].

Theorem 2 (Abstract candidate simulation). Let $q \in \Sigma^{q_0}_{\rightarrow}$ and $q' \in \Sigma^{q_0}_{\rightarrow_{abs}}$ be a concrete and abstract state reachable from the start state q_0. If $Ctrl(q) = Ctrl(q')$, then for all control points $D \in \mathbb{C}^q_{\rightarrow}$, it follows that $D \in \mathbb{C}^{q'}_{\rightarrow_{abs}}$.

Theorem 3 (Abstract deadlock deterministic). Let $q \in \Sigma^{q_0}_{\rightarrow_{abs}}$ be a reachable abstract state with $Dead_{\rightarrow_{abs}}(q)$. If $D \not\subseteq Ctrl(q)$, then for all $q' \in \Sigma^{q_0}_{\rightarrow_{abs}}$, $D \not\subseteq Ctrl(q')$.

"

4.3 SMT Encoding

The SMT encoding rules extend those in [2] to allow prefix matching for orphaned macro actions. These changes are isolated to the definitions and usage of two new sets of variables including timestamps and matches for each send

MATCH ORDER	$(c_{\mathbf{s}_1} \implies c_{\mathbf{w}_5}) \wedge t_{\mathbf{s}_1} < t_{\mathbf{w}_5} \bigwedge (c_{\mathbf{r}_0} \implies c_{\mathbf{w}_2}) \wedge t_{\mathbf{r}_0} < t_{\mathbf{w}_2} \bigwedge (c_{\mathbf{w}_2} \implies c_{\mathbf{r}_4}) \wedge t_{\mathbf{w}_2} < t^1_{\mathbf{r}_4} \bigwedge (c_{\mathbf{r}_4} \implies c_{\mathbf{w}_{13}}) \wedge t^2_{\mathbf{r}_4} < t_{\mathbf{w}_{13}}$
	$\bigwedge (c_{\mathbf{s}_3} \implies c_{\mathbf{w}_{11}}) \wedge t^2_{\mathbf{s}_3} < t_{\mathbf{w}_{11}} \bigwedge (c_{\mathbf{w}_{11}} \implies c_{\mathbf{r}_{12}}) \wedge t_{\mathbf{w}_{11}} < t_{\mathbf{r}_{12}} \bigwedge (c_{\mathbf{r}_{12}} \implies c_{\mathbf{w}_{15}}) \wedge t_{\mathbf{r}_{12}} < t_{\mathbf{w}_{15}} \bigwedge t^1_{\mathbf{s}_3} < t^2_{\mathbf{s}_3} \bigwedge t^1_{\mathbf{r}_4} < t^2_{\mathbf{r}_4}$
QUEUE ORDER	$m_{\mathbf{r}_0} < m^1_{\mathbf{r}_4} \bigwedge m^1_{\mathbf{r}_4} < m^2_{\mathbf{r}_4} \bigwedge m^1_{\mathbf{s}_3} < m^2_{\mathbf{s}_3}$
MATCH COUNT	$atm(1, \{\mathtt{M}(\mathbf{r}_0, \mathbf{s}^1_3), \mathtt{M}(\mathbf{r}_4, \mathbf{s}^1_3)\}) \wedge atm(1, \{\mathtt{M}(\mathbf{r}_4, \mathbf{s}^2_3)\})$
	$\bigwedge atm(1, \{\mathtt{M}(\mathbf{r}_0, \mathbf{s}_1)\}) \bigwedge atm(1, \{\mathtt{M}(\mathbf{r}_0, \mathbf{s}_1), \mathtt{M}(\mathbf{r}_0, \mathbf{s}^1_3)\}) \bigwedge atm(2, \{\mathtt{M}(\mathbf{r}_4, \mathbf{s}^1_3), \mathtt{M}(\mathbf{r}_4, \mathbf{s}^2_3)\})$
MATCH CORRECT	$(c_{\mathbf{s}_3} \iff exa(1, \{\mathtt{M}(\mathbf{r}_0, \mathbf{s}^1_3), \mathtt{M}(\mathbf{r}_4, \mathbf{s}^1_3)\}) \wedge exa(1, \{\mathtt{M}(\mathbf{r}_4, \mathbf{s}^2_3)\}))$
	$\bigwedge (c_{\mathbf{s}_1} \iff exa(1, \{\mathtt{M}(\mathbf{r}_0, \mathbf{s}_1)\})) \bigwedge (c_{\mathbf{r}_0} \iff exa(1, \{\mathtt{M}(\mathbf{r}_0, \mathbf{s}_1), \mathtt{M}(\mathbf{r}_0, \mathbf{s}^1_3)\})) \bigwedge (c_{\mathbf{r}_4} \iff exa(2, \{\mathtt{M}(\mathbf{r}_4, \mathbf{s}^1_3), \mathtt{M}(\mathbf{r}_4, \mathbf{s}^2_3)\}))$
REACH	$c_{\mathbf{r}_0} \bigwedge c_{\mathbf{w}_2} \bigwedge c_{\mathbf{s}_3} \bigwedge c_{\mathbf{w}_{11}}$
DEADLOCK	$\neg c_{\mathbf{s}_1} \bigwedge \neg c_{\mathbf{w}_5} \bigwedge \neg c_{\mathbf{r}_4} \bigwedge \neg c_{\mathbf{w}_{13}} \bigwedge \neg c_{\mathbf{r}_{12}} \bigwedge \neg c_{\mathbf{w}_{15}}$
NO MATCHES	$c_{\mathbf{r}_0} \bigwedge c_{\mathbf{s}_3}$

Fig. 9. SMT encoding for the witness execution in Fig. 5

or receive action, and the MATCH COUNT and MATCH CORRECT rules.

"If $\widehat{\mathcal{F}}(A_D)$ is able to issue each action in the candidate D, then it is used to construct an SMT formula F. A satisfying assignment for F can be used to construct a witness execution for the deadlock candidate. If F is unsatisfiable, then there is no feasible deadlock state that contains D as part of its control point."

To resolve the precise matching, either completely or partially, of each macro action a, the formula F expands a in structure. This is necessary because otherwise the precise count of actions needed for matching a cannot be resolved, or whether the actions in $\phi(a)$ are matched in a feasible order is never known.

The expansion includes two major changes compared to the original SMT encoding. First, instead of defining two variables, the timestamp and the timestamp of the matched action, for each send or receive action a, the formula here defines two new sets of variables: $\{t^i_a : i \in \{1 \dots n(a)\}\}$ and $\{m^i_a : i \in \{1 \dots n(a)\}\}$, where t^i_a and m^i_a respectively represent the timestamp of each action $b_i \in \phi(a)$ and timestamp of the action that matches b_i. For simplicity, if a is not a macro action with $n(a) = 1$, the only needed variables are t_a and m_a that represent t^1_a and m^1_a respectively.

A thing to mention is that the expansion of variables speeds up the process of encoding resolving, though more variables are defined for each macro action a. This is because a satisfying assignment of the encoding only needs to make sure the actions in $\phi(a)$ are matched correctly (usually satisfying the "non-overtaking" property). There is no need to resolve the exact action that matches some action in $\phi(a)$ (as the original encoding does), which takes much more computing time.

Therefore, the formula only preserves the variables t^1_a, $t^{n(a)}_a$, m^1_a and $m^{n(a)}_a$ if a is a macro receive action with $n(a) > 1$, which are sufficient to constrain the scope of its timestamps and the matched timestamps of the potential send actions. Then, the formula simply constrains that the timestamps of the potential send actions are issued and/or matched within the associated scopes (see the expanded rule of $\mathtt{M}(r, s_i)$ below).

The second change is the match set $\mathbb{M}^*_D$ expanded from $\mathbb{M}_D$, the match set $\mathbb{M}$ applied to the CTP A_D. More precisely, for an action $a \in A_D$,

$$\mathbb{M}^*_D(a) = \bigcup_{a' \in \mathbb{M}_D(a)} \phi(a'),$$

meaning all the actions combined in the set $\mathbb{M}_D(a)$ are included in $\mathbb{M}^*_D(a)$. The expansion of match set is straightforward as the macro actions are expanded and the encoding needs to figure out all the combined actions for potentially matching a macro action.

The formula uses w_a as another name for t_w where $w \in W(A_D)$ and $a = req(w)$. Additionally, it adds a boolean variable c_a for every action that is true if a must be issued and completed in the witness execution.

"The rules for the encoding are shown in Fig. 8. The MATCH ORDER and QUEUE ORDER constraints preserve the meaning of match order and queue order in the encoding. An action can only complete if its $<_{mo}$ predecessors have completed and the timestamps must reflect that. The timestamps of the first and last combined actions in a receive must be ordered. The timestamps of all combined actions in a send must follow the order by their indeices. Additionally, matches must conform to the non-overtaking guarantee of MPI executions. In all cases, constraints are omitted when they are obviously redundant with respect to existing constraints and the transitivity of $<$ and $=$ over the integers.

The BARRIERS constraint encodes the inter-process synchronization behavior of barrier actions by asserting that groups complete at the same time. All other timestamps are asserted to be distinct."

The MATCH COUNT constraint enforces the maximum count of matches for each send or receive action. The $atm(k, Z)$ and $exa(k, Z)$ constraints are inspired by the rules in [1], which are satisfied if and only if *at most* and *exactly* k many constraints from the set Z are true, respectively.

The MATCH CORRECT constraint allows two typical occasions that are both correct in a witness execution. First, a send or receive action, whether a macro action or not, is completed if and only if all the actions in $\phi(a)$ are matched. Second, an orphaned action or deadlock action in any deadlock path that is only allowed for prefix matching or never matching at all, is enforced by the false value of the variable c_a indicating not all the actions in $\phi(a)$ are matched.

Let O denote the set of orphaned actions for the candidate D.

$$\begin{aligned} O =& \{a \in D \cap R(A_D) : src(a) = * \wedge n(a) > 1\} \\ & \cup \{a \in A_D : \exists a' \in W(A_D) \cap D, a = req(a')\} \end{aligned}$$

In Section 2.3, a send and receive action were matched by incrementing their n numbers in the matched set, meaning that each time a combined action is matched. This semantic meaning is preserved in the encoding by the $\mathtt{M}(r, s_i)$ constraint where $r \in R(A_D)$ and $s_i \in \phi(s)$ for some $s \in S(A_D)$. Let t_s and m_s be the timestamps of s_i and the action that matches s_i, then $\mathtt{M}(r, s_i)$ expands to

$$m_r^1 \leq t_s \leq m_r^{n(r)} \wedge t_r^1 \leq m_s \leq t_r^{n(r)} \wedge t_s < w_r \wedge t_r^{n(r)} < w_a.$$

The constraint enforces the timestamp t_s to indicate that the send must complete between the completion time of the first and last matched sends for r. Similarly, the constraint enforces the timestamp m_s meaning the matched receive for s_i must complete between the completion time of the first and last receives combined in r. Finally, the timestamps of s_i and r are required to precede the timestamps of both wait actions. Note that in the infinite-buffer setting, the wait for the send action does not exist and so one of these constraints is omitted.

"The REACH constraint asserts that every predecessor of the actions in D is completed except the actions in O. In other words, it asserts that the deadlock D is reachable.

The DEADLOCK constraint simply asserts that the deadlock actions in D and the orphaned actions in O are not complete. The NO MATCHES constraint ensures that any issued actions that could untangle the deadlock are complete, thus forcing them to find matches that exclude the deadlock and orphaned actions. Given a satisfying assignment of the variables in F, the witness execution can be constructed from the t timestamp variables while the matches made can be recovered by consulting the m variables."

Consider the example CTP in Fig. 3. The formula resulting from the witness execution in Fig. 5 is given in Fig. 9. Here the expansion of $\mathtt{M}$ constraints are not interesting and thus omitted for space limit. The superscript i of an action a_j^i, if existing, represents the index that a_j^i is combined in a_j. To resolve the encoding, the REACH and DEADLOCK constraints enforce the true values of $c_{\mathbf{s}_3}$ and $c_{\mathbf{r}_0}$, and the false value of $c_{\mathbf{r}_4}$, which imply the only possible matches in the execution: $\{\mathbf{r}_4, \mathbf{s}_3^2\}$ and $\{\mathbf{r}_0, \mathbf{s}_3^1\}$.

5 EXPERIMENTS

The runtime trace for our approach is observed through code instrumentation. The MPICH library is used for the actual runtime [6]. Two CTPs for each benchmark are used in the experiments: the direct translation of an observed MPI execution, A_0, and the transformation of A_0 by action combination in Definition 6, A_f. The translation and transformation are largely trivial and proceeds as expected. Anything outside message passing as defined in this paper is ignored. Some care must be taken with collective operations but it is all mechanical. Any deadlock cycle is checked once it is detected. Once a cycle verified to be feasible, the test terminates.

The SMT solver Z3 is used by our approach for validation [7]. The experiments are run on an Intel i7-8700K 6-Core processor with 24 GB of memory running Ubuntu 18.04 LTS. A time limit of one hour is set for each test.

5.1 Comparison with Other Tools

The experiments are launched in two stages. First, the experiments compare the performance of our new approach with A_f and two state-of-the-art MPI verifiers MOPPER, a SAT based tool [8], [1], and Aislinn, a dynamic analyzer [9].

MOPPER is a trace based verifier that checks the same behaviors described in this paper. Therefore, the comparison to our approach is direct. The experiments compare the results of two versions of MOPPER, the original tool [8] and the optimized tool [1]. Aislinn is a dynamic verifier that covers two buffering choices for each send action in a program. In other words, Aislinn detects more behaviors than those in the infinite buffer semantics (buffering provided for all sends) and the zero buffer semantics (no buffer for any send). In order to compare to our approach, we set the buffering mode for Aislinn to "eager" (equivalent to the infinite buffer setting) and "rendezvous" (equivalent to the zero buffer setting).

The benchmarks are selected from multiple sources. Each tested program has some typical communication pattern that send and receive actions can be matched in many different ways. This evidence greatly increases the difficulty of deadlock detection, and thus is helpful to evaluate the scalability of the target tools. The benchmarks include,

- *Integrate* [10] implements the algorithm that computes an integral of the $\sin$ function by using a large number of wildcard receives to match the sends from multiple sources.
- *Diffusion2D* [10] has an interesting computation pattern that uses barriers to "partition" the message communication into several sections. A message from a send can be only received in a common section. Deadlock occurs under the zero buffer setting.
- *Floyd* [11] implements the shortest path algorithm for all the pairs of nodes. The message communication is only built between any two successive processes.
- *Heat* [12] implements the solution of the heat conduction equation. The communication pattern for this benchmark contains several message starvation cycles.
- *IS* [13] implements the solution of integer sorting. The message passing in this benchmark is deterministic.

The results of comparison are shown in Table 1. The column "B" represents the buffering setting in the runtime. The column "D" indicates the existence of deadlocks. The "Tm" column for A_fSOL is the time of action combination, executing the main analysis, and constraint solving on A_f. The "Tm" columns for MOPPER and optimized MOPPER are for constraint generation and solving. The "Tm" column for Aislinn is the running time of the tool for either "eager" mode or "rendezvous" mode. The notation "TO" means "time out" (exceeding the time limit). The notation "N/A" means "not available".

TABLE 1
Tests on Selected Benchmarks

Name	#Procs	#Calls	B	D	A_f SOL Tm	MOPPER Tm	MOPPER-o Tm	AISLINN Tm
Integrate	8	60	∞		0.001s	0.091s	0.078s	1.697s
	10	76	∞		0.001s	8.166s	0.089s	6.464s
	16	124	∞		0.003s	TO	1.255s	TO
	32	252	∞		0.015s	TO	1.615s	TO
	64	508	∞		0.165s	TO	3.998s	TO
	128	1020	∞		1.885s	TO	16.685s	TO
Diffusion2D	8	158	0	$\surd$	0.021s	N/A[a]	N/A[a]	79.878s
			∞		0.001s	TO	0.071s	42.725s
	16	318	0	$\surd$	0.047s	N/A[a]	N/A[a]	TO
			∞		0.003s	TO	1.753s	TO
	32	638	0	$\surd$	0.096s	N/A[a]	N/A[a]	TO
			∞		0.008s	TO	2.44s	TO
	64	1278	0	$\surd$	0.43s	N/A[a]	N/A[a]	TO
			∞		0.037s	TO	4.127s	TO
	128	2558	0	$\surd$	1.322s	N/A[a]	N/A[a]	TO
			∞		0.07s	TO	8.553s	TO
Floyd	8	156	∞		0.013s	22.732s	0.12s	TO
	16	316	∞		0.093s	TO	1.528s	TO
	32	636	∞		0.261s	TO	2.186s	TO
	64	1276	∞		2.978s	TO	3.379s	TO
Heat	8	228	0	$\surd$	0.037s	0.263s	0.206s	N/A[b]
	16	468	0	$\surd$	0.041s	2.926s	1.528s	N/A[b]
	32	948	0	$\surd$	0.151s	3.841s	2.144s	N/A[b]
	64	1908	0	$\surd$	0.368s	7.484s	3.362s	N/A[b]
IS	64	380	0		0.001s	0.137s	0.115s	N/A[b]
			∞		0.001s	0.146s	0.147s	N/A[b]
	128	764	0		0.002s	0.229s	0.253s	N/A[b]
			∞		0.005s	0.373s	0.26s	N/A[b]
	256	1532	0		0.006s	0.629s	0.538s	N/A[b]
			∞		0.012s	0.545s	0.504s	N/A[b]

[a] MOPPER is not launched because the process of trace generation finds a deadlock.
[b] Aislinn does not support some MPI calls in the program.

Overall, the results show that our approach with A_f is scalable to over a hundred processes and thousands of calls. Even for a much complex program such as *Diffusion 2D* with 128 processes, the time cost is less than two seconds. The scalability is achieved by the effectiveness of action combination, the abstract machine and the improved SMT encoding that highly reduce the time cost of analysis. In contrast, Aislinn has extremely high time costs for the tests, and even time out for some tests. This is expected because the standard partial order reduction Aislinn launches can generate a large number of interleavings of endpoints. Exhaustive enumeration of these interleavings is time consuming. MOPPER is much quicker but still time out when the size of benchmarks becomes larger (e.g., *Integrate* with more than 16 processes). The slowdown of MOPPER demonstrates that encoding a large program as a whole can make the process of encoding resolving unusually complicated by the backend solver. Our tool is more efficient because the program is only partially encoded when needed. The optimized MOPPER is much more efficient than the original MOPPER because it uses some pre-processing of combination. However, the optimized MOPPER runs over 8 seconds for some tests (*Integrate* with 128 processes and *Diffusion2D* with 128 processes). Unlike our approach of trace compression, the optimized MOPPER only combines the wildcard receives in an identical endpoint. Additionally, like the original MOPPER, the optimized MOPPER encodes all the actions in a program so the number of clauses reaches cubic the number of actions.

5.2 Effectiveness of Action Combination

The second stage of experiments tests the effectiveness of action combination by running the complete cycle detection independently on the CTPs A_f and A_0. The complete cycle detection may cost more time than that presented in Table 1 as the exhaustive enumeration of potential deadlock cycles does not terminate even if a feasible deadlock is detected. Fig. 10 shows the comparison of A_f and A_0 for the number of nodes and edges in the generated graphs and the corresponding time costs of complete cycle detection for several typical benchmarks.

The growths of time costs in Fig. 10 illustrates that for the benchmarks *Integrate*, *Diffusion 2D* and *Floyd*, the CTP A_f has an evident speedup on finishing the cycle detection over the CTP A_0. Among those benchmarks, *Integrate* has a long sequence of wildcard receives in the first process, which introduces many edges for message race. The action combination largely reduces the count of edges and so highly speeds the cycle detection, especially for the test of 128 processes.

Diffusion 2D has several sequences of wildcard receives in the first process and multiple pairs of consecutive send actions in each process, so the reduction of time cost is evident. Note that since the numbers of nodes and edges for *Diffusion 2D* are extremely large for over 64 processes, the time cost for zero buffer is reasonable. The time cost for infinite buffer, however, is under half a second for all the processes, because the nodes do not form valid deadlock paths, and our filtering algorithm in Algorithm 1 can effectively prune a large number of branches in depth first search.

Floyd has multiple consecutive wildcard receives in each process, whose count is the number of processes. The speedup for cycle detection is more obvious with the growth of #process, as more actions are combined.

Heat also has multiple actions that can be combined, but each combination can only happen between two actions, which does not merge many edges in the graph. Therefore, the improvement of cycle detection is not as evident as the tests of other benchmarks.

Overall, the action combination is able to largely help the analysis in deadlock detection. The level of action combination (how many actions can be combined to a single action) is more important than the count of combination (how many new actions are created) for the analysis to be efficient and scalable.

6 Related Work

Much of the literature on predictive program analysis focuses on detecting as many errors as possible from a single observed execution [14], [15]. As mentioned in Section 1, the analysis presented here is maximally predictive for single-path programs but cannot reason about messages that are issued in a schedule-dependent manner. Instead, this paper focuses on improving the efficiency of an SMT based analysis for single-path programs while leaving extensions to more general programs as future work.

The approach in this paper is inspired by several works. Trace compression is usually important because thread traces can be very large in size. Kini et al. proposed a data race detector that uses trace compression [16]. Unlike our approach that simply collapses the consecutive sends or receives in an identical process, the work represents a trace as a special program written by context-free grammars, which achieves a significant compression.

Joshi et al. proposed a method for multi-threaded Java programs by first detecting potential lock dependency cycles with a imprecise dynamic analyzer and then finding real deadlocks by a random thread scheduler with high probability [17]. The refining strategy of the work inspires the staged approach taken in this paper, but the tool presented in this paper is guaranteed to report a deadlock if it exists in the single-path program.

Sherlock is a tool that uses concolic execution for deadlock detection in Java programs [14]. The key idea is similar to our approach: finding potential deadlocks, and then searching for a feasible schedule that leads to the deadlock. The difference is that Sherlock repeatedly finds alternate schedules through solving constraints that describe new permutations of previously observed schedules rather than leveraging an abstract machine to filter out false deadlocks.

A precise SMT encoding technique was proposed by Huang et al. for verifying properties over MCAPI programs containing message race [18]. The encoding does not require a precise match set and was extended to checking zero buffer incompatibility for MPI programs [19]. This technique is adapted for validating deadlocks in this paper.

The POE approach is a dynamic partial order reduction solution [20] for MPI program verification [21], [22]. The approach was extended to POE_{MSE}, which first uses a precise happens-before relation to find the potential sends that may cause different behaviors based on the initial trace, then replays the execution at each potential send with a different choice, i.e. buffering the send instead of matching it [23].

MOPPER is an MPI deadlock detector based on boolean satisfiability encoding [1], [8]. While the solution is precise, the size of the encoding is cubic meaning it only scales to a low degree of message non-determinism.

CIVL is a model checker that uses symbolic execution to verify a number of safety properties of various types of concurrent programs including message passing programs [24], [25], [26]. The tool is outperformed by MOPPER.

An extension to the model checker SPIN [27], is MPI-SPIN that is specific to verifying MPI programs [28], [29]. Since a massive number of states are explored, the work is not scalable.

Böhm et al. provide an approach that aims to find deadlocks for an MPI program under both environments of synchronization and no synchronization [9]. The approach first uses standard partial order reduction to find deadlocks assuming the environment has no synchronization. It then uses an algorithm to search missed deadlocks by enforcing synchronization in the basic operations such as send and collective operations.

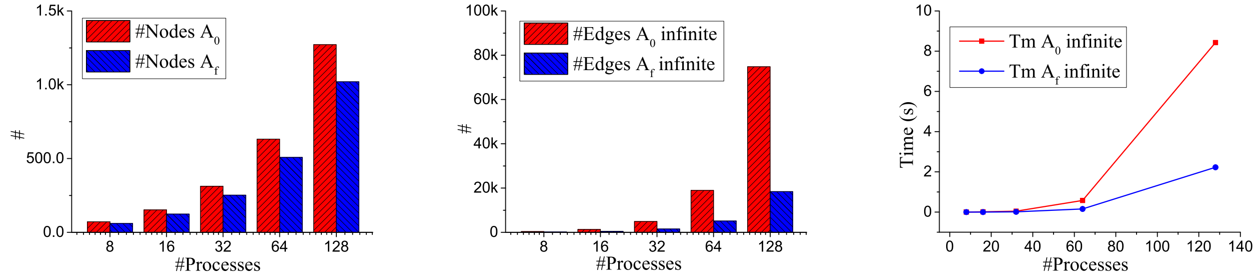
MPI-Checker is a static analyzer based on abstract syntax tree of the source code of MPI programs [30]. The tool is able to check many errors in a program, However, it is limited to check deadlocks caused by complicated semantics of communication.

ParTypes is a type-based approach for verifying MPI programs by developing a protocol language for a type system [31]. Since the approach is able to avoid traversing the state space, the analysis is scalable for large programs.

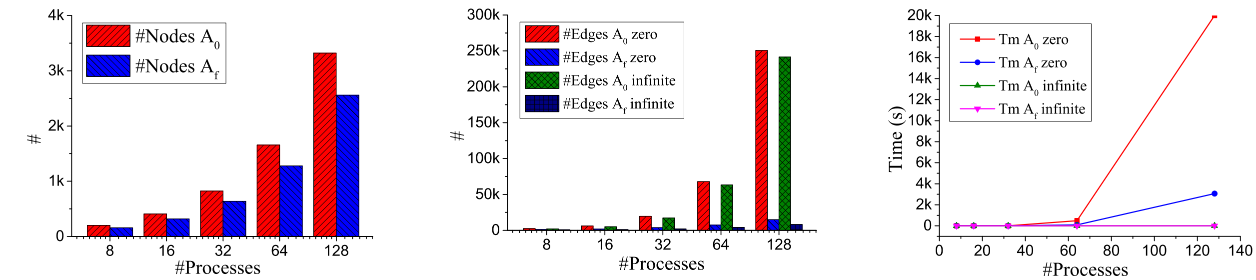
Umpire is an approach of runtime verification for checking multiple MPI errors such as deadlock and resource tracking [32]. The error checking is taken by spawning one manager thread and several outfielder threads in the execution of an MPI program. An extension to Umpire is Marmot [33]. The work uses a centralized server instead of multiple threads for error checking. Another extension to Umpire is MUST [34], [35]. The structure of MUST allows the users to execute the error checking either in an application process itself or in extra processes that are used to offload these analyses. However, just like Umpire and Marmot, the approach is neither sound nor complete for deadlock detection.

7 Conclusion

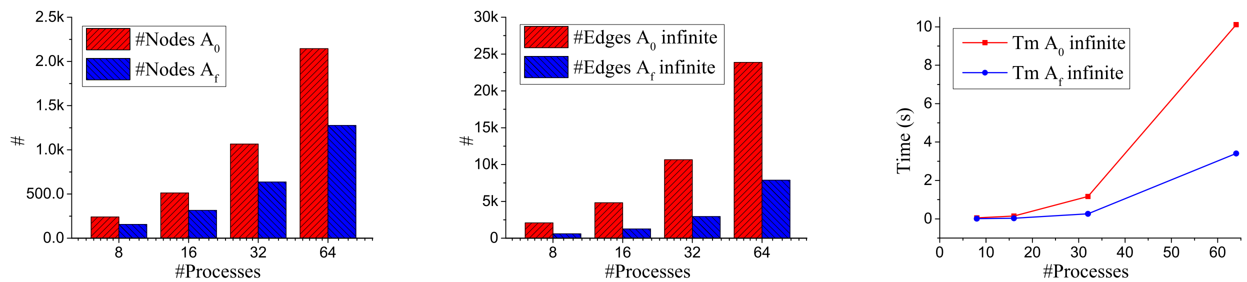
This paper presents a new approach that automatically detects deadlocks in single-path MPI programs after observing a single execution. The actions in the execution, if consecutively sending or consecutively receiving messages in an identical endpoint, can be combined to simplify the presentation of execution, which directly reduces the complication of the mapped dependency graph. The



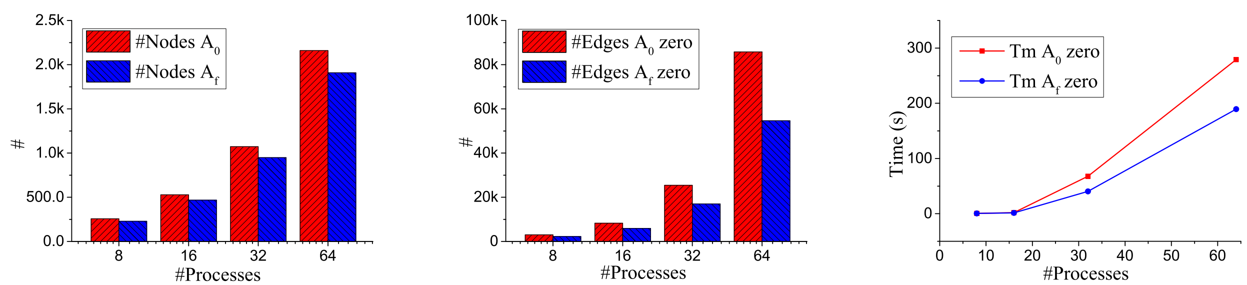
(a) *Integrate*



(b) *Diffusion 2D*



(c) Floyd



(d) *Heat*

Fig. 10. The #nodes, #edges and time costs of complete cycle detection for the benchmarks (a) *Integrate*, (b) *Diffusion 2D*, (c) *Floyd* and (d) *Heat* under zero and/or infinite buffer settings.

approach leverages a simple characterization of deadlock to efficiently detect deadlock candidates in the dependency graph. An abstract machine is used to quickly disregard many infeasible candidates while the remaining candidates are precisely validated by an SMT solver with an efficient encoding for deadlock. The approach is sound and complete for deadlock detection in any single-path MPI program on a given input. Experiments show that the new approach with combined actions performs much more efficient for typical benchmarks, comparing to the approach without combination of actions, and the other state-of-the-art MPI verifiers. Future work considers more filtering techniques and extending the approach to support multiple-path MPI programs with more complicated structures.

8 Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 62102325, and in part by the Ministry of Education of Humanities and Social Science Project with Grant 20YJC630146.

References

[1] V. Forejt, S. Joshi, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 4, pp. 15:1–15:27, 2017. [Online]. Available: http://doi.acm.org/10.1145/3095075

[2] Y. Huang, B. Ogles, and E. Mercer, "A predictive analysis for detecting deadlock in MPI programs," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 18–28. [Online]. Available: https://doi.org/10.1145/3324884.3416588

[3] The full version of paper. [Online]. Available: https://github.com/yuhuang/MPIDeadlockExtFullPaper

[4] Y. Huang, K. Gong, and E. Mercer, "An efficient algorithm for match pair approximation in message passing," *Parallel Computing*, vol. 91, p. 102585, 2020.

[5] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM J. Comput.*, vol. 4, no. 1, pp. 77–84, 1975. [Online]. Available: http://dx.doi.org/10.1137/0204007

[6] MPICH, "High-Performance Portable MPI," *http://www.mpich.org*.

[7] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, vol. 4963. Springer, Heidelberg, 2008, pp. 337–340.

[8] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma, "Precise predictive analysis for discovering communication deadlocks in MPI programs," in *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, 2014, pp. 263–278. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06410-9_19

[9] S. Böhm, O. Meca, and P. Jancar, "State-space reduction of non-deterministically synchronizing systems applicable to deadlock detection in MPI," in *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*, 2016, pp. 102–118. [Online]. Available: https://doi.org/10.1007/978-3-319-48989-6_7

[10] "FEVS benchmark," *http://vsl.cis.udel.edu/fevs/index.html*.

[11] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. M. Voelker, "MPIWiz: subgroup reproducible replay of mpi applications," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, 2009, pp. 251–260.

[12] M. S. Mueller, G. Gopalakrishnan, B. R. de Supinski, D. Lecomber, and T. Hilbrich, "Dealing with MPI Bugs at Scale: Best Practices," in *Automatic Detection, Debugging, and Formal Verification*, 2011.

[13] D. Bailey, E. Barszcz, B. J.T, B. D.S, C. R.L, D. D, F. R.A, P. Frederickson, L. T.A, R. Schreiber, H. Simon, V. Venkatakrishnan, and W. K, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, pp. 63–73, 09 1991.

[14] M. Eslamimehr and J. Palsberg, "Sherlock: scalable deadlock detection for concurrent programs," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 353–365. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635918

[15] C. G. Kalhauge and J. Palsberg, "Sound deadlock prediction," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[16] D. Kini, U. Mathur, and M. Viswanathan, "Data race detection on compressed traces," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 26–37. [Online]. Available: https://doi.org/10.1145/3236024.3236025

[17] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *PLDI*, 2009, pp. 110–120.

[18] Y. Huang, E. Mercer, and J. McCarthy, "Proving MCAPI executions are correct using SMT," in *ASE*, 2013, pp. 26–36.

[19] Y. Huang and E. Mercer, "Detecting MPI zero buffer incompatibility by SMT encoding," *NFM*, 2015.

[20] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *POPL*, 2005, pp. 110–121.

[21] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, and R. M. Kirby, "ISP: a tool for model checking MPI programs," in *PPOPP*, 2008, pp. 285–286.

[22] S. Sharma, G. Gopalakrishnan, and G. Bronevetsky, "A sound reduction of persistent-sets for deadlock detection in MPI applications," in *Formal Methods: Foundations and Applications - 15th Brazilian Symposium, SBMF 2012, Natal, Brazil, September 23-28, 2012. Proceedings*, 2012, pp. 194–209. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33296-8_15

[23] S. S. Vakkalanka, A. Vo, G. Gopalakrishnan, and R. M. Kirby, "Precise dynamic analysis for slack elasticity: Adding buffering without adding bugs," in *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*, 2010, pp. 152–159. [Online]. Available: https://doi.org/10.1007/978-3-642-15646-5_16

[24] S. F. Siegel, M. B. Dwyer, G. Gopalakrishnan, Z. Luo, Z. Rakamaric, R. Thakur, M. Zheng, and T. K. Zirkel, "Civl: The concurrency intermediate verification language," Department of Computer and Information Sciences, University of Delaware, Tech. Rep. UD-CIS-2014/001, 2014.

[25] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel, "CIVL: formal verification of parallel programs," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 830–835. [Online]. Available: https://doi.org/10.1109/ASE.2015.99

[26] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "CIVL: the concurrency intermediate verification language," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*, 2015, pp. 61:1–61:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807635

[27] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[28] S. F. Siegel, "Model checking nonblocking MPI programs," in *VMCAI*, 2007, pp. 44–58.

[29] S. Siegel, "Verifying parallel programs with MPI-Spin," in *PVM/MPI*, 2007, pp. 13–14.

[30] A. Droste, M. Kuhn, and T. Ludwig, "Mpi-checker: static analysis for MPI," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM 2015, Austin, Texas, USA, November 15, 2015*, 2015, pp. 3:1–3:10. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833159

[31] H. A. López, E. R. B. Marques, F. Martins, N. Ng, C. Santos, V. T. Vasconcelos, and N. Yoshida, "Protocol-based verification of message-passing parallel programs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA,*

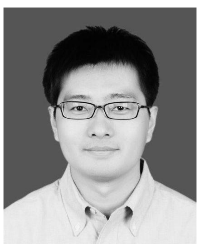
October 25-30, 2015, 2015, pp. 280–298. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814302

[32] J. S. Vetter and B. R. de Supinski, "Dynamic software testing of MPI applications with umpire," in *Proceedings Supercomputing 2000, November 4-10, 2000, Dallas, Texas, USA. IEEE Computer Society, CD-ROM*, 2000, p. 51. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SC.2000.10055

[33] B. Krammer, K. Bidmon, M. S. Müller, and M. M. Resch, "MARMOT: an MPI analysis and checking tool," in *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, 2003, pp. 493–500.

[34] T. Hilbrich, M. Schulz, B. R. de Supinski, and M. S. Müller, "MUST: A scalable approach to runtime error detection in MPI programs," in *Tools for High Performance Computing 2009 - Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, September 2009, ZIH, Dresden*, 2009, pp. 53–66. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11261-4_5

[35] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, "MPI runtime error detection with MUST: advances in deadlock detection," in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*, 2012, p. 30. [Online]. Available: https://doi.org/10.1109/SC.2012.79



Yu Huang received the PhD degree in computer science from Brigham Young University. He worked as a research engineer at Fortify Inc. (owned by Micro Focus). Currently, he is an associate professor at School of Information and Engineering, Southwestern University of Finance and Economics, China. His research interests include parallel and distributed system, software verification, model checking and big data analysis.

Tao Wang received the PhD in management information systems from Chungnam National University, South Korea. Currently, he is a professor at Southwestern University of Finance and Economics, China. His research interests include the adoption and implementation of information systems innovations. He has published in Information Systems Journal, Online Information Review, Electronic Commerce Research and Applications, Behavior & Information Technology, Industrial Management & Data Systems, Journal of Business Research, and other venues.

Zihui Yin received the bachelor's degree in computer science from Southwestern University of Finance and Economics, China. Currently, she is a 2-year master student in computer software and theory at Southwestern University of Finance and Economics, China. During her graduate studies, she was awarded several academic scholarships. She will work in ByteDance after graduation. Her research interests include formal verification and MPI distributed programs.

Eric Mercer received the PhD degree in Electrical Engineering from the University of Utah. Currently, he is an associate professor at Brigham Young University, USA. His research interests include software verification, model checking, static analysis, and programming languages.

Benjamin Olges received the bachelor's degree in computer engineering from Brigham Young University. Currently, he is a software engineer at 3dB Labs, Inc.. His research interests include software verification, parallel algorithms, hardware acceleration, distributed systems and signal processing.

9 APPENDIX

9.1 Proof of Theorem 1

First we prove a few useful lemmas. Fix a reachable deadlock $q = \langle A, I, M \rangle$ and its full control point $D = Ctrl(q)$.

Lemma 1. [deadlock-actions-blocking]

$$D \subseteq W(A) \cup B(A)$$

Proof: Proof by contradiction. Assume that $\exists a \in D, a \notin W(A) \cup B(A)$.

Then $a \in S(A) \cup R(A)$. Let a' be the very next action after a in $A_{pid(a)}$. a' must exist since a is a send or receive action and the last action in every process is a barrier action.

By definition D contains the last issued action of every process in $P(A)$. It follows that $a \in I$ and $a' \notin I$. Furthermore, by the definition of deadlock, there is no transition enabled in q that can issue q'.

The definition of $\rightarrow$ requires that every action preceding a in $A_{pid(a)}$ must be issued and every blocking action preceding a must be matched in q. Because $a \in I$ and $a \in S(A) \cup R(A)$, these conditions also hold for a'. Therefore a' can be issued. This contradicts the deadlock of q. □

Lemma 2. [wildcard-causes-send-starvation] Let $a \in D \cap W(A)$ be a wait action participating in the deadlock at q such that $req(a) \in S(A)$. If there are no receive actions in $(R(A) \setminus I)_{dst(req(a))}$ (no unissued potential matches), then there must exist an issued wildcard receive action $a' \in (R(A) \cap I)_{dst(req(a))}, src(a') = *$ in the destination process.

Proof: Proof by contradiction. Assume that there is no receive $a' \in R(A) \cap I$ such that $pid(a') = dst(req(a))$ and $src(a') = *$. Then the number of send and receive match pairs between $src(req(a))$ and $dst(req(a))$ is fixed in every execution of A. Since q is deadlocked and there are no unissued potential matches for $req(a)$, it follows that $\sum_{a_s \in (S(A) \cap I)_{src(req(a))}} n(a_s) > \sum_{a_r \in R(A)_{dst(req(a))}} n(a_r)$. Therefore, A will deadlock deterministically at a. This contradicts the fact that A was obtained by observing a successful execution of an MPI program. □

Lemma 3. [wildcard-causes-recv-starvation] Let $a \in D \cap W(A)$ be a wait action participating in the deadlock at q such that $req(a) \in R(A) \wedge src(req(a)) \neq *$. If there are no send actions in $(S(A) \setminus I)_{src(req(a))}$ (no unissued potential matches), then there must exist an issued wildcard receive action $a' \in (R(A) \cap I)_{pid(a)}, src(a') = *$ in the destination process.

Proof: Proof by contradiction. Assume that there is no receive $a' \in R(A) \cap I$ such that $a' <_{po} req(a)$ and $src(a') = *$. Then the number of send and receive match pairs between $src(req(a))$ and $dst(req(a))$ is fixed at least until a (where after a there may be more receives but no more sends). Since q is deadlocked and there are no unissued potential matches for $req(a)$, it follows that $\sum_{a_r \in (R(A) \cap I)_{dst(req(a))}} n(a_r) > \sum_{a_s \in S(A)_{src(req(a))}} n(a_s)$. Therefore, A will deadlock deterministically at a. This contradicts the fact that A was obtained by observing a successful execution of an MPI program. □

Lemma 4. [parent-action-exists]

$$\forall a \in D, (\exists a' \in A \setminus I, pid(a') \neq pid(a) \wedge (a', a) \in E^*)$$

Proof: Proof by case analysis on the action type of a. By Lemma 1, $a \in W(A) \cup B(A)$.

In the first case, $a \in W(A) \wedge req(a) \in S(A)$ If there is an unissued potentially matching receive action a' in $(A \setminus I)_{dst(req(a))}$, then rule two will ensure that $(a', req(a)) \in E$. If there is no such matching receive, then $a' = \perp_{dst(req(a))}$. By Lemma 2, there must be a wildcard receive in $I_{dst(req(a))}$. Therefore, rule four will again ensure that $(a', req(a)) \in E$. In either case, because $req(a) <_{mo} a$, it follows that $(req(a), a) \in E$ and thus $(a', a) \in E^*$.

In the second case, $a \in W(A) \wedge req(a) \in R(A)$ If there is an unissued potentially matching send action a' in $(A \setminus I)_{src(req(a))}$, then rule two will ensure that $(a', req(a)) \in E$. If there is no such matching send, then $a' = \perp_{src(req(a))}$. By Lemma 3, there must be a wildcard receive in $I_{pid(a)}$. Therefore, rule three will again ensure that $(a', req(a)) \in E$. In either case, because $req(a) <_{mo} a$, it follows that $(req(a), a) \in E$ and thus $(a', a) \in E^*$.

In the final case, $a \in B(A)$ Then $a' \in B(A) \setminus I$ and rule two ensures that $(a', a) \in E$. Such an a' must exist, otherwise a could be completed and q would not be a deadlock.

In the first two cases, $req(a)$ is the orphaned action, in the last case, a itself is the orphaned action. □

Lemma 5. [actions-preceding-deadlock-actions-connected]

$$\forall a \in D, (\exists a' \in D, a_o \in A_{pid}(a), a'_o \in A_{pid}(a'),$$
$$a_o \leq_{po} a \wedge a'_o \leq_{po} a' \wedge (a'_o, a_o) \in E^*)$$

Proof: Direct proof. By Lemma 4, there exists some action $a'' \in A \setminus I$ such that $pid(a'') \neq pid(a)$ and $(a'', a) \in E^*$. Let a_o be an action in $pid(a)$ such that $(a'', a_o), (a_o, a) \in E^*$. By definition of deadlock path, a_o must exist.

By definition, D contains an action $a' \in I_{pid(a'')}$. In the first case of deadlock path, it follows that $a' \leq_{po} a''$. If $a' <_{po} a''$, then $a' <_{mo} a''$ since a' is a blocking action by Lemma 1. In this case, rule one ensures that $(a', a'') \in E^*$. Let a'_o be the action $req(a')$ if $a' \in W(A)$, or equal to a' if $a' \in B(A)$. Then, $a'_o \leq_{mo} a'$ by the definition of match order, and rule one ensures $(a'_o, a') \in E^*$. By the transivity of E^*, $(a'_o, a_o) \in E^*$.

In the second case of deadlock path, it follows that $a'' \leq_{po} a'$. Let a'_o be equal to a''. Then, $(a'_o, a_o) \in E^*$ is evident.

If $a' = a'' = \perp_{pid(a')}$, then this is a contradiction since $a' \in I$ by the definition of D and $a'' \notin I$ by Lemma 4. □

Now we can prove Theorem 1. By Lemma 5, there exist some actions a_o and a'_o preceding the actions a and a' in $pid(a)$ and $pid(a')$ respectively such that $a, a' \in D$, $a \neq a'$ and $(a'_o, a_o) \in E^*$. By the same argument, there exists an action a''_o preceding the action a'' in $pid(a'')$ such that $a'' \in D$, $a' \neq a''$ and $(a''_o, a'_o) \in E^*$. If $a_o = a''_o$, then we have a cycle $C = \{(a_o, a'_o), \ldots, (a'_o, a_o)\}$. Otherwise, we can continue applying Lemma 5 until a cycle is created between the actions in D.

Let d be the partial control point or candidate extracted from C. Particularly, the action $a_d \in d$ can be the earliest blocking action in C from process $pid(a_d)$ if there are more than one action for process $pid(a_d)$ in C. The action a_d can also be the very next wait action for a wildcard receive $req(a_d)$ with $n(req(a_d)) > 1$ if the receive is the only action for process $pid(a_d)$ in C. In the second case, a_d is outside C but can be trivially obtained. The actions that make up edges in C are completely enumerated in Lemmas 4 and 5. C is entirely made up of actions in D, their message requests (which are non-blocking), unissued matches and final barrier actions. Therefore, $d \subseteq D$ holds by construction.

9.2 Proof of Theorem 2

Proof by contradiction. Without loss of generality, assume that no match transitions are enabled from q and q' in $\rightarrow$ and $\rightarrow_{abs}$ (match transitions do not modify the control points). Now assume that some issue transition $q \rightarrow q''$ is enabled such that $Ctrl(q'') \notin \mathbb{C}^{q'}_{\rightarrow_{abs}}$. Then, by the definition of $Ctrl(q'')$, there is some action $a \in A$ that can be issued by $\rightarrow$ but not by $\rightarrow_{abs}$ in the states q and q' repsectively. Let $q = \langle A, I, M \rangle$ and $q' = \langle A', I', M' \rangle$.

A few useful facts follow from the definitions of the two transition systems and the assumptions above.

1) $A \subseteq A'$ and $A' \setminus A$ consists solely of all wildcard send actions generated by the abstract machine when issuing send actions.
2) $I \subseteq I'$ and $I' \setminus I$ consists solely of all wildcard send actions generated by the abstract machine when issuing send actions.
3) There exists some earliest action $a' <^{A'}_{po} a$ that is preventing a from being issued by $\rightarrow_{abs}$ in the state q' because it is not issued or matched as required by the issue transition rule.
4) a' did not prevent $\rightarrow$ from issuing a in state q and therefore a' must be issued and/or matched as required in that state.

Fact three leads to a few cases. In the first case, $a' \notin I'$, thereby blocking a from being issued in the state q'.

If $a' \notin A$, then it is one of the wildcard send actions generated by the abstract machine. But by the definition of $\rightarrow_{abs}$, every wildcard send in A' is also in I' which is a contradiction.

So it must be that $a' \in A$ is not a wildcard send action. But then facts two and four imply that $a' \in I'$, again a contradiction.

In the second case, $a' \in W(A') \cup B(A') \wedge a' \notin M'$. If $a' \in B(A')$, then facts two and four imply that $B_{A'}(grp(a')) \subseteq I'$. And because there are no match transitions enabled from state q' in $\rightarrow_{abs}$, it follows that $B_{A'}(grp(a')) \subseteq M'$ which is a contradiction.

If a' is not a barrier action, then $a' \in W(A')$ must be an incomplete wait action. And because there are no match transitions enabled from state q' in $\rightarrow_{abs}$, it follows that $req(a')$ has no potential match in I'.

In contrast, fact four implies that $req(a')$ has matched with some action in the state q. Furthermore, fact two guarantees that the same number of send and receive actions from A have been issued between $src(req(a'))$ and $dst(req(a'))$ in the states q and q'. And because the abstract machine issues a wildcard send for every deterministic send, there must exist an enabled match for $req(a')$ in I'. This contradicts a previous assumption.

Because all cases lead to a contradiction, it must be that $Ctrl(q'') \in \mathbb{C}^{q'}_{\rightarrow_{abs}}$. $\square$

9.3 Proof of Theorem 3

Proof by contradiction. Let $t = q_0 \rightarrow_{abs} q_1 \ldots \rightarrow_{abs} q$ be a transition sequence in $\Sigma^{q_0}_{\rightarrow_{abs}}$ that reaches the deadlock state q. Now assume that there exists another state $q' \in \Sigma^{q_0}_{\rightarrow_{abs}}$ such that $Dead_{\rightarrow_{abs}}(q')$ and $D \subseteq Ctrl(q')$.

Then there must be some earliest q_i in t such that $q_i \rightarrow^*_{abs} q'$ and $q_{i+1} \not\rightarrow^*_{abs} q'$. In other words a scheduling choice causes the full control point subsuming D to become reachable. Because the abstract machine isolates wildcard actions on their own endpoints, the only scheduling choice that can affect which blocking actions are able to be matched is the choice of which wildcard send actions are matched with wildcard receive actions.

Therefore the transition $q_i \rightarrow_{abs} q_{i+1}$ must have matched a wildcard send action s with a wildcard receive action r and there must be some other send action s' that can be matched with r along the sequence $q_i \rightarrow^*_{abs} q'_{i+1}$. Furthermore, the matching of s and r must have prevented some blocking action in a process with an action in D to block indefinitely. However the definition of the **Issue-Send** transition in $\rightarrow_{abs}$, s and s' can be matched with the exact same set of actions and are entirely interchangeable. Therefore, if $q'_{i+1} \rightarrow^*_{abs} q'$, then $q_{i+1} \rightarrow^*_{abs} q'$. This is a contradiction.