Natural proofs for Asynchronous Programs using Almost-synchronous Invariants

Ankush Desai
University of California, Berkeley
ankush@eecs.berkeley.edu

Pranav Garg
University of Illinois at Urbana-Champaign
garg11@illinois.edu

P. Madhusudan
University of Illinois at Urbana-Champaign
madhu@illinois.edu

Abstract

We consider the problem of provably verifying that an asynchronous message-passing system satisfies its local assertions. We present a novel reduction scheme for asynchronous event-driven programs that finds *almost-synchronous invariants*— invariants consisting of global states where message buffers are close to empty. The reduction finds almost-synchronous invariants and simultaneously argues that they cover all local states. We show that asynchronous programs often have almost-synchronous invariants and that we can exploit this to build natural proofs that they are correct. We implement our reduction strategy, which is sound and complete, and show that it is more effective in proving programs correct as well as more efficient in finding bugs in several programs, compared to current search strategies which almost always diverge. The high point of our experiments is that our technique can prove the Windows Phone USB Driver written in P [9] correct for the receptiveness property, which was hitherto not provable using state-of-the-art model-checkers.

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

Writing correct asynchronous event-driven programs, which involve concurrently evolving components communicating using messages and reacting to input events, is difficult. These programs typically have layers of design, where the higher layers reason with how the various components (or machines) interact and the protocol they follow, and where lower layers manage more data-intensive computations, controlling local devices, etc. However, the programs often get written in traditional languages that offer no mechanisms to capture these abstractions, and hence over time leads to code where the individual layers are no longer discernible. High level protocols, though often first designed on paper using clean graphical state-machine abstractions, eventually get lost in code. Model-checking tools for protocols typically are designed to work on the abstract protocol layer, and hence verification tools for such programs face the daunting task of extracting these models from the programs.

The natural solution to the above problem is to build a programming language for asynchronous event-driven programs that preserves the protocol abstractions in code. Apart from the difficulty in designing such a language, this problem is plagued by the reluctance of programmers to adopt a new language of programming and the discipline that it brings. However, this precise solution was pioneered in a new project at Microsoft Research recently, where, during the development of Windows 8, the team building the USB driver stack decided to use a domain-specific language for asynchronous event-driven programs called P [9]. Programs written in P capture the high-level protocol using a collection of interacting state machines that communicate with each other by exchanging messages. The machines, internally, also have to do complex tasks such as process data and perform low level control of devices, reading sensors or controlling devices, etc., and these are modeled using external foreign functions written in C.

The salient aspect of P is that it is a programming paradigm where the protocol model and the lower level data and control are *simultaneously* expressed in the same language. P programs can be compiled to native code for execution, while the protocol model itself can be extracted cleanly from the code in order to help perform analysis, especially those relevant to finding errors in the protocol. Writing code in P gives immediate access to designers to correct errors found by analysis tools during the design phase itself, and significantly contributed to building a more reliable USB stack [9]. Maintenance of the code in P automatically keeps these models up to date, enabling verification mechanisms to keep up with evolving code.

The primary specification that P programs are required to satisfy in [9] is *responsiveness*. Each state declares the precise set of messages a machine can handle and the precise set of messages it will *defer*, implicitly asserting that all other messages are not expected by the designer to arrive when in this state. Receiving a message outside these sets hence signals an error, and in device drivers, often leads drivers to crash. The work reported in [9] includes a *systematic testing* tool for the models using model-checking, where the system is explored for hundreds of thousands of states to check for errors. However, such model-checking seldom succeeds in proving the program correct, since there are many sources of infinity, including message buffer sizes.

Verifying asynchronous event-driven programs: In this paper, we wish to build techniques that provably verify asynchronous event-driven programs against local assertions (including receptiveness, which can be modeled as a local assertion). There are many sources of infinity in verification of event-driven programs— the local data, the message buffer sizes, and the *number* of spawned machines being the primary ones. Our primary concern in this paper is to tackle the *asynchrony* of message passing which causes unbounded message buffers. Our goal is to effectively and efficiently *prove* (as opposed to systematically test) event-driven programs correct, when the number of processes and the local data are bounded, but when message buffers are unbounded.

Consider the simple scenario where a machine p sends a machine q unboundedly many messages, like in a producer-consumer setting. Even in this simple scenario, systematic model-checkers would fail to terminate checking local assertions, even when the local data stored at p and q is finite, since message buffers get unbounded. For example, the Zing model-checker [4] used to systematically test P programs in [9] will fail to finish, since the message buffers are unbounded and are part of the global state that's explored.

Almost-synchronous Invariants: Our primary thesis is that *almost-synchronous invariants* often suffice to prove asynchronous event-driven programs correct, and furthermore, a search for these invariants is also more effective in finding bugs. Intuitively, almost-synchronous states are those where the message buffers are close to empty, and almost-synchronous invariants are collections of such states that ensure that all local states have been discovered. For instance, in the producer-consumer example above, exploring the sends of p immediately followed by the receive in q discovers an almost-synchronous invariant where message buffers are bounded by 1, though blindly exploring the state-space would never lead to termination.

The primary contribution of this paper is a sound and complete reduction scheme that discovers almost-synchronous invariants using model-checking. The key idea is to explore interleavings that keep the message buffers small, while at the same time finding a closure argument that argues that all local states have been discovered, at which point we can terminate. Intuitively, for any partial order described by the system, we aim to "cover" this using a linearization that has small buffer sizes. The reduction scheme is quite involved, and even subverts the semantics of the underlying system, for instance throwing away messages into ether, to achieve small buffer sizes.

Natural Proofs: The technique set forth in this paper is a method involving *natural proofs*. Intuitively, the idea behind natural proofs is to find some *simplicity* of real-world instances and exploit them to find a simple proof of correctness, even when the general verification problem may be undecidable. The problem of checking whether an asynchronous program is correct, even when the number of machines and local data are bounded, is an undecidable problem [8]. Our thesis is that asynchronous systems often have a reasonably small set of almost-synchronous global states that can be used as an invariant to prove the program correct. For instance, when a process p sends a message to q, a global state would capture all possible states q could be in at that time; however, the designer of the program would actually be concerned with and argue about the states q could be in *when it receives the message currently being sent.* Almost-synchronous invariants are states that capture these kinds of global states, where message-buffers are close to empty. Finding these almost-synchronous global states often suffices in capturing the dynamics of the communication protocol and proving that it satisfies its specification.

Our solution strategy is hence to find a set of almost-synchronous invariants, prove that they are sufficient to cover all local states, and that they verify the local assertions. We discover almost-synchronous invariants in this paper using state-space exploration and model-checking.

Natural proofs have been studied earlier in the entirely different domain of logic-based verification of programs manipulating dynamic data-structures [28, 33, 38].

Implementation and Evaluation: We have implemented our reduction mechanism technique for discovering almost-synchronous invariants (ASI) of P programs. Our invariant synthesis is built over the Zing model-checker [4], adapting it to explore the state-space of P programs using our reduction strategy. The existing systematic model-checker for P programs (also implemented in Zing) [9] almost never terminates, and can finish exhaustive state-space exploration only when message buffers are bounded in some fashion. We show however that our reduction can handle such P programs *without* bounding buffers. We show two classes of results over a class of P programs analyzed for the receptiveness property. The first class of results show that our reductions can prove P programs correct, for arbitrary message buffers. This analysis works *faster*, despite handling unbounded buffers, than the naive exploration does on bounded buffers. The second class of results show that our reductions also help in finding bugs in incorrect P programs, exploring less states and performing faster than the existing exhaustive search techniques. The high point of our experiments is the complete verification of the USB Windows Phone Driver, which our tool can prove receptive with no bound on message buffers, a proof that has hitherto been impossible to achieve using current model-checkers.

2. Motivation

The key idea of this paper is that almost-synchronous invariants suffice to find proofs of local assertions in event-driven asynchronous programs. Given an asynchronous program with local assertions, we would like to explore a set of reachable global states that covers all reachable local states. However, this set of global states need not be the set of all reachable global states (partial-order reduction [12, 14] also works this way; all global states are not explored, but all local states are covered).

Synchronous states, intuitively, is a set of global states where message buffers are empty. From the perspective of rely-guarantee reasoning [20], when a machine p sends a message to machine q, p is not quite concerned with what the state of q is when the send-event happens, but rather is concerned with the state of q when it receives the message it sends, which is essentially what synchronous states capture. However, synchronous invariants (invariants containing synchronous states) may themselves not suffice to prove a system correct for two reasons: (a) in order to ensure that all synchronous states have been explored, we may need to explore asynchronous states (where message buffers are not empty), and (b) certain local states may manifest themselves only in asynchronous states. *Almost-synchronous invariants* are invariants of the system expressed using global states where message buffers are close to empty, but for which inductiveness of the invariant is provable and which covers all local states. The primary thesis of this paper is that almost-synchronous invariants (ASI) often exist for event-driven asynchronous programs, and natural proofs that target finding such invariants can prove their correctness efficiently.

We will present, in Section 4, a reduction scheme (called *almost-synchronous reduction*) that will explore a selective set of interleavings that leads to the discovery of ASIs and simultaneously proves their inductiveness. The primary aim of the reduction is to explore interleavings that keep the message buffers to the minimal size needed, while still ensuring that all local states are eventually explored. The reduction will be *sound* and *complete*— all errors will be detected (if the search finishes) and all reported errors will be real errors.

The first rule of our almost-synchronous reduction (presented in Section 4) is to schedule *receive*-events whenever they are enabled, suppressing *send*-events. This rule ensures that messages are removed from message queues as soon as possible, thus ensuring message buffers are contained, and as we show in practice, often bounded.

To appreciate this prioritization, consider the producer-consumer scenario on the right, where process p sends an unbounded number of messages to q, which q receives (p could do this by having a recurring state send out messages received by a recurring state of q).

The reduction that we propose will explore this scenario (partial-order) using the linearization consisting of an unbounded number of rounds, where in each round p sends to q followed by q immediately receiving the message from p, thus exploring an essentially synchronous interleaving where the message buffer is bounded by 1. Furthermore, and very importantly, when exploring this interleaving, the search will discover that the global state repeats, which includes the local states of all machines and the contents of all message buffers. This is entirely because the message buffer gets constantly depleted causing the global state to recur.

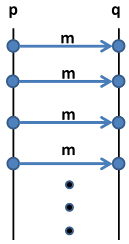

Figure 1: Producer Consumer Scenario

Difference with partial-order reduction: Note that techniques such as *partial-order reduction* [12, 14] do not necessarily help here. Even an optimal static or dynamic partial-order reduction that promises to explore every partial-order using just one linearization, cannot assuredly help. In the above example, if the linearization chosen is the one where the sends from p are all explored first (or a large number of them are explored) before the corresponding receives are explored, then each global state along this execution would be *different* because the message buffer content is different in each step. This turns out to be true for both depth-first and breadth-first searches with partial-order reduction. Note that this problem does not, in general, arise when systems communicate through bounded shared memory only; it is message-passing that causes the problem. Consequently, partial order techniques do not address the concerns that our constructions address in this paper. In fact, we do not do much partial order reduction in this paper, and in fact may explore several interleavings that correspond to the same partial order, provided these interleavings exhibit almost synchronous behavior. Our reductions sometimes do achieve some partial-order reduction, but it is minimal and unintentional! Adding partial-order reduction *over and above* our reduction is indeed a possibility (in order to reduce the number of interleavings explored) and is an interesting future direction.

Handling truly asynchronous behaviors:
If a system readily presents always synchronous events (all sends enabled always have the matching receive events immediately enabled in the receiving process), then one can solely explore the executions with synchronous events only and keep the sum of all message buffer sizes to 1. While

this often happens, it does not typically happen all the time in a system's evolution, which is why we need almost-synchronous global states to be explored. Let us consider several scenarios where such asynchrony happens and explain how our reduction technique mitigates this.

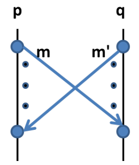

Figure 2:

First, consider the following scenario where p wants to send a message to q and q also is sending a message to p. Clearly, we cannot explore synchronous messages at this point, and we need to let these sends happen without their corresponding receive events. It turns out that in many asynchronous message-passing programs, this scenario does occur (even the simple *elevator* example in [9] has such a scenario). However, it turns out that the system often quickly recovers where p after sending the message, soon gets to a receive mode where it accepts the message from q, and similarly q, after sending its message, soon receives the message from p. Hence a careful execution of these sends followed by prioritizing receive-events over send-events often lets us recover a synchronous state. The reduction that we propose will explore such an interleaving that leads to recovery of a synchronous state after a mild asynchronous excursion.

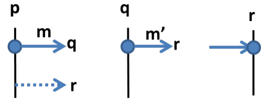

Figure 3:

Let us now consider another example, the one on the right— here p is sending a message to q, and q is sending a message to r, where r is able to receive messages from either. Now note that scheduling only synchronous events in this situation, which means only scheduling the send of q and the receive of r, will lead to *incompleteness* (i.e., the exploration can miss local states). For instance, it may be the case that p, after sending the message to q, sends a message to r (denoted by the dotted arrow), and r receives this message before the send-event of q happens. This execution will be missed if we only scheduled synchronous events. Hence it is important to note that scheduling only synchronous events is complete only when in the current state *all sends* have matching receives enabled.

The simplest way to address the problem is to enable both the sends of p and q. However, this could lead to flooding the message queues unnecessarily, for example if p and q continue sending more messages. Our mechanism will actually split this scenario into two cases. The first case is when p is a *potential* sender to r, i.e., in some state of p, it could send a message to r. When that's the case, the above execution we outlined could happen, and our reduction will enable both sends of p and q, which will lead to the execution being discovered.

However, in the case when p is *not* a potential sender to r (and assuming there are no other processes), we will do the following. We will enable the send of q (followed by the receive of this message in r). Also, we will allow a move that *blocks* process q, which means that process q will not be able to transition any longer. Once q is blocked, all processes that are sending messages to q are essentially sending messages that will never get received, and hence they can send their messages to ether, i.e., we can *lose* these messages and not store them in the configuration at all. In the above scenario, we will enable q getting blocked, and hence allow p to send its message to q (which promptly gets lost), and in this way enable p to proceed while at the same time keep message buffers small. Note that blocking any process at any time is always sound. Completeness is harder to establish, and crucially depends on the topology of the system, including the communication behavior of the machines. In the above scenario, it does turn out that blocking q is complete as well. When there are more processes in the system, the condition under which we will allow such a blocking is more complex, depending on the set of potential senders to r, etc.

The use of blocked processes is another unique aspect of our reduction, and crucially relies on the semantics of message passing. A generic reduction technique, such as partial-order reduction, which works by handling shared memory and message passing uniformly does not achieve such reductions, as what we do strays away from the normal semantics of transitions on the global state. In other words, we are *under-approximating* the global state description itself, while preserving soundness and completeness.

3. Event-driven automata

In this section we introduce an automaton model, called event-driven automata (EDA), for modeling event-driven programs, inspired by and very similar to P programs. Event-driven automata are however a lot simpler, allowing us to define the reductions and prove precise theorems about them. We will then lift the reductions to general programs, including programs written in P (see Section 5).

In our automaton model, a program is a finite collection of state machines communicating via messages. Each state machine is a collection of states, has local variables and has a set of actions. Each machine also has a single FIFO queue into which other machines can enqueue messages. We will not restrict any of the sets (states, domain of local variables, payload on messages, etc.) to be finite; all of them can be infinite, and hence our automata can model event-driven software. For instance, P programs allow function calls in local machines; these can be modeled in our automata using an appropriate encoding of the call-stack in the state. Also, for simplicity, we will assume there is no process/machine creation; our reduction does extend to this setting, but it is more clear to explain our algorithms without these complications. Section 5 describes how we extend our algorithms to work on general P programs.

A message is modeled as a pair $\pi = (m, l)$, consisting of a message type m (from a finite set) and an associated payload

l belonging to some (finite or infinite) domain. Let M be a finite set of message types and let Dom_M be the payload domain. Then, a message π belongs to $\Pi = M \times Dom_M$. We fix M, Dom_M, and Π for the rest of the paper.

Let Dom be the domain for the local variables in the state machines. Without loss in generality, we assume that each machine in the program has a single local variable, and fix Dom for the rest of the paper. Also let us denote $f^{\mathcal{T}}$ to be the class of all (computable) functions of type $\mathcal{T}$.

Event-driven automata (EDA): An *event-driven automaton* over $\Pi = (M \times Dom_M)$ and Dom is a tuple $\mathcal{P} = (\{P_i\}_{i \in N})$, where $N = \{1, \cdots, n\}$, $n \in \mathbb{N}$, and each $P_i = (Q_i^s, Q_i^r, Q_i^{int}, q_i^0, val_i^0, T_i, Def_i, q_i^{err})$, where

- $Q_i = Q_i^s \uplus Q_i^r \uplus Q_i^{int} \uplus \{q_i^{err}\}$ is the set of states, partitioned into states that send a message Q_i^s, states that receive messages Q_i^r, internal states Q_i^{int}, and an error state q_i^{err}.
- $q_i^0 \in Q_i^s \cup Q_i^r$ is the initial state of P_i;
- $val_i^0 \in Dom$ is the initial valuation for the local variable in P_i;
- T_i is the set of transitions for P_i and is partitioned into send transitions T_i^s, receive transitions T_i^r and internal transitions T_i^{int}.
  Send transitions are of the form:
  $T_i^s : Q_i^s \longrightarrow (Q_i^{int} \cup \{q_i^{err}\}) \times (N \backslash \{i\}) \times M \times f^{Dom \longrightarrow Dom_M}$,
  Receive transitions are of the form:
  $T_i^r : Q_i^r \times M \longrightarrow (Q_i^{int} \cup \{q_i^{err}\}) \times f^{Dom \times Dom_M \longrightarrow Dom}$
  Internal transitions are of the form:
  $T_i^{int} : Q_i^{int} \longrightarrow 2^{(Q_i^s \cup Q_i^r \cup \{q_i^{err}\}) \times f^{Dom \longrightarrow Dom}}$;
- $Def_i : Q_i^r \longrightarrow 2^M$ associates a deferred set of messages to each receive state. □

When $T_i^s(q) = (q', j, m, f)$, this means that machine P_i, when in state q and local variable valuation v can transition to q', sending the message of type m with a payload $f(v)$ to machine P_j. Note that a machine cannot send messages to itself (we assume this mainly for technical convenience). Similarly, when $T_i^r(q, m) = (q', f)$, this means that P_i can receive message (m, l) when in state q and valuation v, and update its state to q' and local variable to $f(v, l)$. When $T_i^{int}(q)$ contains (q', f), it means that P_i can (non-deterministically) transition from state q and local variable valuation v to state q' and local valuation $f(v)$.

Note that, by definition, send transitions are deterministic, and receive transitions are deterministic for any received message; true local non-determinism is only present in internal transitions. Also note that every send or receive transition takes the control of the machine to an internal state and is immediately followed by an internal transition which non-deterministically transitions the machine to a send or a receive state. From the way we have defined transitions, the automaton can transition from any state to the error state q_i^{err} and from the error state, no further transitions are enabled.

$$\frac{C[i] = (q_i, v_i, \mu_i) \quad (q_i', f) \in T_i^{int}(q_i)}{C \xrightarrow{i} C[i \mapsto (q_i', f(v_i), \mu_i)]}\ \text{INTERNAL}$$

$$\frac{\begin{array}{c} C[i] = (q_i, v_i, \mu_i) \qquad C[j] = (q_j, v_j, \mu_j) \\ T_i^s(q_i) = (q_i', j, m, f) \end{array}}{C \xrightarrow{i!j} C[i \mapsto (q_i', v_i, \mu_i)][j \mapsto (q_j, v_j, \mu_j\ (m, f(v_i)))]}\ \text{SEND}$$

$$\frac{\begin{array}{c} C[i] = (q_i, v_i, \mu_i\ (m, l)\ \mu_i') \\ \mu_i \in [Def_i(q_i) \times Dom_M]^* \qquad m \notin Def_i(q_i) \\ T_i^r(q_i, m) = (q_i', f) \end{array}}{C \xrightarrow{i?} C[i \mapsto (q_i', f(v_i, l), \mu_i\ \mu_i')]}\ \text{RECEIVE}$$

$$\rightarrow\ =\ \xrightarrow{i} \uplus \xrightarrow{i!j} \uplus \xrightarrow{i?}$$

Figure 4: Semantics of EDA

As we noted above, in our automaton model, messages sent to a machine are stored in a FIFO queue. However, as in P programs, we allow the possibility to influence the order in which the messages are received by deferring them. In a given receive state q in machine P_i, some messages can be *deferred*, which is captured by the set $Def_i(q)$. When a machine is in this state, it skips all the messages that are in its deferred set and dequeues the first message that is not in its deferred set.

The communication model we follow is that whenever a machine sends a message to another machine, the message is immediately added to the receiver's queue. This is the same communication model as in P, which was mainly designed to model event-driven programs running on a single machine, for example an operating system driver. Note that this communication model is, however, general enough and can be used to also model distributed systems where messages sent by a machine reach the receiving machine after arbitrary time delay (but in FIFO order). One can model such a system by introducing a separate channel process between every pair of machines. This process dequeues messages from its sender and immediately forwards it to the receiver. Since there are multiple channel processes forwarding messages to a given machine, interleavings between them has the same effect as having messages delivered with delay.

3.1 Formal semantics of EDA

A (global) configuration of an EDA consisting of n machines is a tuple $C = (\{C_i\}_{i \in N})$, where $N = \{1, \cdots, n\}$, and where C_i (denoted as $C[i]$) is the local configuration of the i^{th} machine. The configuration $C[i]$ belongs to $(Q_i \times Dom \times \Pi^*)$. The first and the second component of $C[i]$ refer to the current state of the i'th machine and the value of its local variable; the third component is the incoming message queue to machine P_i, modeled as a sequence of pairs of a message type and a payload. For a given configuration C and a local configuration of the i'th machine C'_i, let $C[i \mapsto C'_i]$ be the configuration which is the same as C except that its i^{th} configuration is C'_i.

The initial configuration of the EDA is C_{init} where $C_{init}[i] = (q_i^0, val_i^0, \epsilon)$ for all $i \in N$. The rules for the operational semantics of EDAs are presented in Figure 4. The rules for the send and the internal transitions are straightforward; the rule for a receive transition is slightly more complex. From a receive state q_i, machine P_i skips all the messages in its queue that are in its deferred set and dequeues the first message m from its queue that is not in its deferred set. The state of the machine and the value of its local variable is updated according to the semantics of the receive transition.

Let $Reach_G$ be the set of global configurations of the EDA that can be reached from its initial configuration, and it can be computed as $lfp(\lambda S.C_{init} \cup \{C' \mid C \rightarrow C', C \in S\})$. Let $Bad_G = \{C \mid C[i] = (q_i^{err}, v_i, \mu_i)$ *for some* v_i, μ_i, *and* $i \in N\}$ be the set of error configurations of the EDA. Then we say that the EDA is safe or correct if $Reach_G \cap Bad_G = \emptyset$.

Note that even when the states, Dom and Dom_M are finite, the problem of checking whether a given EDA is safe is an undecidable problem [8].

4. Almost-Synchronous Reductions for Event-driven Automata

Given an event-driven automaton, we describe in this section a reduction that selectively explores a subset of the global reachable configurations of the EDA, such that the exploration is sufficient to cover all the local states that can be reached by the EDA. Our reduction mechanism does so by constructing *almost-synchronous invariants*, which are invariants for proving local assertions in distributed programs and are expressed as a set of global configurations of the system where the message buffers are close to empty. Finally, we argue in this section that our reduction is both sound and complete and can be effectively used for verifying local assertions in asynchronous/distributed programs.

Given an automaton $\mathcal{P}$, we present a construction of a transition system $\mathcal{P}_\mathcal{R}$ such that the set of reachable states of $\mathcal{P}_\mathcal{R}$ correspond to a reduced set of global configurations of $\mathcal{P}$ that form an almost-synchronous invariant of the system. Unlike standard reductions, the states as well as transitions will be *different* than that of the automaton. States in $\mathcal{P}_\mathcal{R}$ are of the form (C, B) where C is a configuration of the automaton $\mathcal{P}$ and is of the form $(\{C_i\}_{i \in N})$, $C_i \in (Q_i \times Dom \times \Pi^*)$, and $B \subseteq N$ is a subset of *blocked machines*.

As briefly motivated in Section 2, transitions in $\mathcal{P}_\mathcal{R}$ prioritize receive actions over send transitions, thereby ensuring that the message queues remain small. From a configuration that cannot receive any further messages from its queues, $\mathcal{P}_\mathcal{R}$ enables a subset of send transitions whose choice depends on the communication pattern amongst the machines in the current configuration as well as the system-wide global communication pattern amongst machines that is statically determined. Naively enabling only a subset of send transitions will miss out on exploring states that can be reached on taking transitions that are never enabled. To circumvent this problem, $\mathcal{P}_\mathcal{R}$ allows at every step a move that blocks those machines whose send transitions were prioritized.

Blocked machines remain forever blocked and can take no transitions. Furthermore, messages sent to blocked machines do not end up in its queue, but are lost to ether, since the blocked machine will anyway not receive them. Consequently, these transitions deviate from the semantics of EDA, but we will show that nevertheless the reduced transition system is sound and complete in discovering all local states.

Before we give the construction of $\mathcal{P}_\mathcal{R}$, let us first introduce certain concepts that are important for understanding the construction.

Definition 4.1 (Senders)**.** *For a given machine $j \in N$ and a configuration C of the EDA, senders(j, C) is the set of machines $i \in N$ such that $C[i] = (q_i, v_i, \mu_i)$ and $T_i^s(q_i) = (q'_i, j, m, f)$, for some $q_i, q'_i, v_i, \mu_i, m, f$.* □

Intuitively, *senders*(j, C) is the set of all machines i that are sending a message to machine j in configuration C. This is used to capture the communication pattern amongst the machines in the current configuration.

Definition 4.2 (Potential-Senders)**.** *For a given machine $j \in N$, potential-senders(j) is the set of machines $i \in N$ such that there exists a send state $q_i \in Q_i^s$ such that $T_i^s(q_i) = (q'_i, j, m, f)$ for some q'_i, m, f.* □

Unlike senders, the notion of potential senders is independent of the current configuration. The potential senders of a machine j is the set of all machines that can possibly send a message to it. This depends on the system-wide global communication pattern amongst the machines which can be statically determined.

Definition 4.3 (Unblocked-Senders)**.** *For a given machine $j \in N$ and an extended configuration (C, B), unblocked-senders(j, C, B) is the set of machines $i \in N$ such that $i \notin B$ and $i \in$ senders(j, C).* □

For a state (C, B) of the transition system $\mathcal{P}_\mathcal{R}$, *unblocked -senders*(j, C, B) = *senders*$(j, C) \setminus B$. Given that the machines in B are blocked and not allowed to transition, *unblocked-senders*(j, C, B) captures the set of machines that are allowed to send a message to j from the current state (C, B).

Definition 4.4 (isReceiving)**.** *Given a machine $j \in N$ and a configuration C such that $C[j] = (q_j, v_j, \mu_j)$, the predicate isReceiving(j, C) is true iff $q_j \in Q_j^r$.*

Example 1. *Consider the producer-consumer scenario in Figure 1 and let C be its starting configuration. Then, senders(q, C) = $\{p\}$, senders(p, C) = $\emptyset$, and $p \in$*

potential-senders(q). *Also, isReceiving*(p, C) = *false while isReceiving*(q, C) = *true.*

Secondly, consider the scenario in Figure 3 and let C be its starting configuration. Then, senders$(r, C) = \{q\}$, *senders*$(q, C) = \{p\}$, *and potential-senders*$(r) = \{p, q\}$. *Further, senders*$(p, C) = \emptyset$, *isReceiving*(r, C) = *true and isReceiving*(q, C) = *isReceiving*(p, C) = *false.*

We now introduce an important concept, called *destination sets*. From an extended configuration (C, B) of the transition system $\mathcal{P}_{\mathcal{R}}$, our reduction mechanism only explores a subset of the possible send transitions. From a state (C, B) of $\mathcal{P}_{\mathcal{R}}$, our algorithm enables only those transitions that send a message to machines in a destination set, which is defined below.

Definition 4.5 (Destination sets)**.** *Given an extended configuration* (C, B), $X \subseteq N$, *a subset of machines, is a destination set if X contains at least one machine* $x \in N$ *such that unblocked-senders*$(x, C, B) \neq \emptyset$ *and for all machines y such that there is an* $x' \in X$ *with* $y \in$ *potential-senders*(x) *and* $y \notin B$, *the following conditions hold:*

1. *if isReceiving*(y, C) *is true, then* $y \in X$,
2. *if for some machine* $z \in N$, $y \in$ *unblocked-senders*(z, C, B), *then* $z \in X$. □

In other words, for an extended configuration (C, B), a destination set X is a set that includes a machine who has at least one unblocked sender, and for every machine $x \in X$, if y is a potential sender of x, then (1) if y is in receive mode, then $y \in X$, and (2) if y is unblocked and in send mode, then the machine it is sending to is in X.

Note that there could be many destination sets for an extended configuration. Also, note that the set of *all* machines is always a destination set, provided there is at least one machine with an unblocked sender.

Further, note that the two conditions on X are *monotonic*, and hence we can start with a single machine x that has at least one unblocked sender, and close it with respect to the two conditions to get the *least* set containing x that is a destination set.

We now fix a particular choice of destination set for every extended configuration (C, B) that has a machine with at least one unblocked sender. This could be the one obtained by choosing a canonical machine with an unblocked sender and closing it with respect to the two conditions, as described above.

In any case, let us fix a function *destination-set* that maps every extended configuration (C, B) to a destination set if there is at least one machine with an unblocked sender, and to the empty set otherwise.

Example 2. *In the scenario in Figure 2, let C be the starting configuration and let the blocked set B be empty. Then we can argue that one of the destination sets is* $\{q\}$. *This can be computed by taking q, which has an unblocked sender, and closing it with respect to the conditions, which doesn't add any more machines. Note that* $\{p\}$ *is also a destination set.*

Thus, in the reduction, if we choose the destination set $\{q\}$, *then we will enable the send from p to q. Now if p after sending the message gets to a receive state, the destination set constructed for this new state will be* $\{p\}$, *which will force us to enable the other send, from q to p.*

Example 3. *In the scenario in Figure 3, let C be the starting configuration and let the blocked set B be empty. Then notice that* $\{r, q\}$ *is a destination set, with r having an unblocked sender. However,* $\{r\}$ *is not a destination set, and in fact* $\{r, q\}$ *is the smallest destination set including* $\{r\}$. *If we choose this destination set, then our reduction will enable all the send transitions to them, i.e., the sends from p to q and from q to r. Notice the fact that p being a potential sender to r forces our reduction to also enable the send transition from p to q.*

The Reduction

We are now ready to define the reduction. The informal algorithm for the reduction is as follows.

Given that the system is in an extended configuration (C, B), we will explore the following transitions from it:

- If any machine is in receive mode and there is an undeferred message on its incoming queue, then we will schedule *all* such receive events and disable all send events.
- If no receives can happen, then we construct the set X = *destination-set*(C, B). Then we schedule *all* sends that send to some machine in X, including sends emanating from X. Furthermore, we also enable a transition that blocks the unblocked senders to X.

The first rule prioritizes receives over sends. The second one selects a subset of sends to enable, depending on the destination set computed. Furthermore, it also enables blocking the unblocked senders to X, which results in a new configuration where senders to X will not be explored, while other send events can be explored. Also, note that sends to blocked machines will have their messages sent to ether.

Figure 5 describes the construction of the transition system $\mathcal{P}_{\mathcal{R}}$ with a transition relation $\longrightarrow \subseteq (C \times 2^N) \times (C \times 2^N)$. The initial state of $\mathcal{P}_{\mathcal{R}}$ is $(C_{init}, \emptyset)$ where C_{init} is the initial configuration of the EDA $\mathcal{P}$ and the set of blocked machines is empty. Let us define $Reach_R$, in the natural way, as the set of states reachable by $\mathcal{P}_{\mathcal{R}}$ from its initial state. By definition, $Reach_R$ is an almost-synchronous reduction of the set of configurations that can be reached by $\mathcal{P}$.

If $\mathcal{P}_{\mathcal{R}}$ is in state (C, B) such that a receive transition is enabled from the configuration C of EDA $\mathcal{P}$, $\mathcal{P}_{\mathcal{R}}$ prioritizes the receive transition (rule Receive in Figure 5). The other three rules in Figure 5– send-to-unblocked , send-to-

$$\textsc{receive}\ \dfrac{C \xrightarrow{i?} C'}{(C, B) \longrightarrow (C', B)}$$

$$\textsc{send-to-unblocked}\ \dfrac{\begin{array}{c} \mathit{NoReceivesEnabled(C)} \quad C \xrightarrow{i!j} C' \\ j \in \mathit{destination\text{-}set}(C, B) \quad i, j \notin B \end{array}}{(C, B) \longrightarrow (C', B)}$$

$$\textsc{send-to-blocked}\ \dfrac{\begin{array}{c} \mathit{NoReceivesEnabled(C)} \quad C \xrightarrow{i!j} C' \\ j \in \mathit{destination\text{-}set}(C, B) \quad i \notin B \quad j \in B \end{array}}{(C, B) \longrightarrow (C[i \mapsto C'[i]], B)}$$

$$\textsc{block}\ \dfrac{\begin{array}{c} \mathit{NoReceivesEnabled(C)} \\ B' = \{\, i \mid i \in \mathit{unblocked\text{-}senders}(j, C, B), j \in \mathit{destination\text{-}set}(C, B)\} \quad B' \neq \emptyset \end{array}}{(C, B) \longrightarrow (C, B \cup B')}$$

$$\text{where}$$
$$\mathit{NoReceivesEnabled(C)} : \quad \textit{for all k, if } C[k] = (q_k, v_k, \mu_k) \textit{ and } \mathit{isReceiving}(k, C) \textit{= true, then } \mu_k \in [\mathit{Def}_k(q_k) \times \mathit{Dom}_M]^*$$

Figure 5: The reduced transition system $\mathcal{P}_\mathcal{R}$ whose reachable states $Reach_R$ is an almost-synchronous reduction that includes all local states reachable in $\mathcal{P}$.

blocked and block apply only when no receive transitions are enabled from configuration C (captured by the condition *NoReceivesEnabled*(C)). In this case, our reduction mechanism first constructs the destination set for the current state (C, B). Then, P_R enables all send transitions that send a message to a machine j belonging to this set. This case is split into two rules: the rule send-to-unblocked handles the case where j is not blocked and the second rule send-to-blocked handles the case where j is blocked and the message sent is not enqueued but is lost to ether. In the latter case, note that the configuration of the sender machine i is only updated, and the receiver j's configuration is unaffected. At the same time, to ensure that our selective exploration does not miss any behaviors, from the state (C, B), $\mathcal{P}_\mathcal{R}$ also blocks the machines i whose send transitions to machines j were selectively enabled (rule Block). Note that Figure 5 does not depict the internal transitions. However, $\mathcal{P}_\mathcal{R}$ does include internal transitions ((C, B) can transition to (C', B) if any internal transition takes C to C'), and in fact these internal transitions are prioritized so that they immediately happen. Since there is no shared state, we do not need to interleave internal transitions in different machines, and hence they happen atomically with the earlier send/receive transition.

Observe that whenever a machine is added to the blocked set, it is in a send state (the rule block in Figure 5). It follows that a machine, if blocked, remains forever blocked and can take no further transitions.

Let $Bad_R = \{(C, B) \mid C \in Bad_G\}$. Also let $\longrightarrow^* \subseteq (C \times 2^N) \times (C \times 2^N)$ be the transitive closure of the single step transition relation $\longrightarrow$ of $\mathcal{P}_\mathcal{R}$. We next argue that only exploring states that are reachable in $\mathcal{P}_\mathcal{R}$ is both sound and complete with respect to proving the correctness of the automaton $\mathcal{P}$. In other words, a local state is reachable in the program iff it is reachable in the reduced transition system.

Theorem 4.6 (Soundness)**.** *If some state $(C_e, B_e) \in Reach_R \cap Bad_R$, then there exists a configuration $C' \in Reach_G \cap Bad_G$.*

Proof sketch: Consider the $\mathcal{P}_\mathcal{R}$-reachable, error trace $(C_{init}, \emptyset) \longrightarrow \cdots (C, B) \longrightarrow \cdots (C_e, B_e)$ where $(C_e, B_e) \in Bad_R$. Then we can show that essentially the same set of actions can be mimicked in $\mathcal{P}$ as well, except that the configurations may contain a bit more information on certain message buffers. As we traverse the trace in $\mathcal{P}_\mathcal{R}$, at any point, we construct a $\mathcal{P}$-reachable configuration C' which is same as C except for the queue contents of machines that have been already blocked along the error trace. For receive, send-to-unblocked and block transitions along the error trace, the update to C' is straight forward. On a send-to-blocked transition along the error trace, the update to C' departs from the update to (C, B). The update to C', in this case, follows the semantics of EDA $\mathcal{P}$ and enqueues the message into the queue of the blocked machine. As we know that machines that have been blocked cannot take any further transitions, this means that the message enqueued in the blocked machine's queue will be never received by it as we move forward along the error trace. Hence, though C' differs from C it never diverges away from it (i.e., a $\mathcal{P}_{\mathcal{R}}$-transition enabled from (C, B) will be always enabled from configuration C'; also the states of machines in C' are the same as the states of machines in C). Since $C_e \in Bad_R$, it follows that configuration C'_e we end up with is such that $C'_e \in Reach_G \cap Bad_G$. □

We next argue the completeness of our reduction mechanism. For that, let us introduce $\rightarrow_B \subseteq C \times C$ for $B \subseteq N$ such that $C \rightarrow_B C'$ if configuration C' of automaton $\mathcal{P}$ is reachable from C along a $\mathcal{P}$-trace that involves no transition by any of the machines in the set B. Formally,

$$\rightarrow_B = (\textstyle\bigcup_{i \notin B} \xrightarrow{i}) \cup (\bigcup_{i \notin B} \xrightarrow{i!j}) \cup (\bigcup_{i \notin B} \xrightarrow{i?})$$

and let $\rightarrow^*_B$ be the transitive closure of $\rightarrow_B$. The completeness result, Theorem 4.8, follows essentially from the following lemma. This lemma asserts that whenever we can reach an error configuration from a configuration C in the original program without involving any transition of machines in the set B, we can reach an error configuration in the reduced transition system from the extended configuration (C, B).

Lemma 4.7. *If for configurations C, C_e and set $B \subseteq N$ such that $C \rightarrow^*_B C_e$ where $C_e \in Bad_G$, then there exists C', B' such that $(C, B) \longrightarrow^* (C', B')$ and $(C', B') \in Bad_R$.*

Proof sketch: First, we will assume that $C \notin Bad_G$, for otherwise the lemma is obvious. The proof is by contradiction. Assume that there are configurations C, C_e and set $B \subseteq N$ such that $C \rightarrow^*_B C_e$ where $C_e \in Bad_G$, and that there is no $\mathcal{P}_{\mathcal{R}}$-state $(C', B') \in Bad_R$ such that $(C, B) \longrightarrow^* (C', B')$. Let us consider an ordering over the space of C, C_e and B. Let this ordering be the standard lexicographic ordering over $(\mathbb{N} \times \mathbb{N} \times \mathbb{N})$, where the first component is the length of the trace $C \rightarrow^*_B C_e$; the second component is the sum (over all machines) of the messages pending in the queues in configuration C; and the third component is size of the complement of the blocked set B. This is a well-founded ordering. Let us pick configurations C, C_e and set B that satisfy all the assumptions and is smallest with respect to this lexicographic ordering.

We show that we can always make "progress" along the $C \rightarrow^*_B C_e$ trace via a $\mathcal{P}_{\mathcal{R}}$-transition, thereby getting a smaller counter-example with respect to the lexicographic ordering, leading to a contradiction.

We split using two cases, the first when a receive is enabled in configuration C, and the second when no receive is enabled.

Case 1: Let us first consider the case when machine i ($i \notin B$) in configuration C is ready to receive a message from its queue. Let the $\rightarrow^*_B$-trace be $\tau : C \rightarrow_B \cdots \rightarrow_B C_e$. Now, consider the case where there is a transition of machine i in the sequence τ. Then the first transition of machine i in τ must be a receive event. Consider the trace $\tau' = C \xrightarrow{i?}_B C_1 \rightarrow_B \cdots \rightarrow_B C_e$ obtained from τ by moving this receive transition to the front; this is a valid $\rightarrow^*_B$-trace. Using rule RECEIVE in the reduced transition system, it follows that $(C, B) \longrightarrow (C_1, B)$ and also that there exists no state $(C', B') \in Bad_R$ such that $(C_1, B) \longrightarrow^* (C', B')$. Note that τ'-suffix from C_1 to C_e has a shorter length than τ. This means that the choice C_1, C_e and B is a strictly smaller counter-example, which is a contradiction.

When no transition of i is present along τ, the trace $\tau_1 : C \rightarrow_B \cdots C_e \xrightarrow{i?}_B C'_e$ obtained from τ by augmenting it with transition $i?$ is a valid $\rightarrow^*_B$-trace such that $C'_e \in Bad_G$. As before, trace $\tau'_1 = C \xrightarrow{i?}_B C_1 \rightarrow_B \cdots C'_e$ is also a valid $\rightarrow^*_B$-trace but one whose suffix from C_1 to C'_e has the same length as τ. However, note that C_1 has one less message pending in its queues compared to C. The choice C_1, C'_e and B is a counter-example and is strictly smaller than C, C_e and B, which is a contradiction.

Case 2: Now consider the second case when no receive transitions are enabled in configuration C. Let $X = \textit{destination-set}(C, B)$.

Consider the subcase where the $\rightarrow^*_B$-trace $\tau : C \rightarrow_B \cdots \rightarrow_B C_e$ contains a transition that sends a message to $x \in X$. Let $p!x$ be the first such transition occurring along τ. We will argue that the transition $p!x$ in this case can be commuted to the beginning and it is possible to construct a valid $\rightarrow^*_B$-trace $\tau' : C \xrightarrow{p!x}_B C_1 \cdots \rightarrow_B C_e$. From the rules SEND-TO-UNBLOCKED OR SEND-TO-BLOCKED, it follows that $(C, B) \longrightarrow (C_1, B)$. We can argue that C_1, C_e and B is a smaller counter-example (since the suffix of τ' from C_1 is shorter), leading to a contradiction. Now let us argue that the first transition of p in τ is $p!x$ (if this is so, it is easy see that $p!x$ can be commuted to the front). By definition, $p \in \textit{potential-senders}(x)$ and $p \notin B$. We will show that in C, p is in a state sending to x. If p is in a receive state in C, then by the definition of destination sets, $p \in X$ (since $x \in X$, $p \in \textit{potential-senders}(x)$, and p is in a receive state). This implies that in τ, before the send event by p happens, there must be a send-event by some machine to p (since we are in the case where the buffers to enabled receivers are empty). Since $p \in X$, this send is a send event to X, which contradicts the assumption that $p!x$ was the first transition along τ sending a message to a machine in X. If p is in a send state in C but it is sending a message to a machine $y \neq x$, then from the definition of destination sets, $y \in X$. Again, this implies that τ has a transition $p!y$ for $y \in X$ before the $p!x$ event, which is again a contradiction. The only option left is that $p!x$ is enabled in configuration C.

We still need to arrive at a contradiction when the $\rightarrow^*_B$-trace $\tau : C \rightarrow_B \cdots \rightarrow_B C_e$ contains no transition that sends a message to a machine $x \in X$. Let $B' = \bigcup_{x \in X} \textit{unblocked-senders}(x, C, B)$ for $x \in X$. Since τ has no transitions sending messages to X, τ involves no transitions by machines in B'. Now let us show that B' is non-empty. Note that the machine involved in the first transition along τ is an unblocked sender. This implies that $X = destination - set(C, B)$ is non-empty. Hence, there must be an unblocked sender to X (by definition of destination sets). Hence B' is non-empty. From the rule BLOCK, it follows that $(C, B) \longrightarrow (C, B \cup B')$. Also, $C \rightarrow^{*}_{B \cup B'} C_e$ is *true*. Since $B \cup B'$ is strictly larger than B, the counter-example C, C_e and $B \cup B'$ is smaller, which is a contradiction. □

Theorem 4.8 (Completeness)**.** *If some configuration $C \in Reach_G \cap Bad_G$, then there exists C', B' such that $(C', B') \in Reach_R \cap Bad_R$.*

Proof: The theorem follows directly from the above lemma by substituting B in the lemma to be the empty set and C to be the initial state C_{init} of the automaton $\mathcal{P}$. □

5. Lifting ASI Reductions to P

A P program [9] is a collection of state machines communicating via asynchronous events or messages. Each state machine in P is a collection of states; it has a set of local variables whose values are retained across the states of the machine, has an entry method which is the state in which the control transfers to on the creation of a new machine and finally, has a FIFO incoming queue through which other machines can send messages to it. Each state in a P state machine has an entry function which is the sequence of statements that are first executed whenever the control reaches that state. Additionally, each state has a set of transitions associated with incoming message types, has a set of action handlers associated with the incoming message types, as well as a classification of certain message types as being deferred or ignored in the given state. After the entry function has been executed, the P machine continues to remain in the same state till it receives a message in its queue. The machine dequeues the first message from its queue that is not deferred and checks if the message is ignored in the current state. If it is, the message is simply dropped from the queue, and the machine continues to remain in the same state. If the message is not ignored, the machine dequeues the message; the next state to which the machine transitions to along with the update to its local state on dequeuing the message is determined by the state's transitions and action handlers. P statements include function calls and calls to foreign functions that are used to model interaction with the environment. A P state machine can have call statements and call transitions in it which are used to implement hierarchical state machines.

We will describe next the mapping between P features and the EDA we introduced in Section 3. EDAs do not support dynamic creation and deletion of machines. We did not find this to be a serious limitation as most driver programs written in P and distributed protocols we modeled in P had a statically determined bounded number of machines. Hence the global communication pattern, amongst the machines in the EDA, required for our reductions could also be statically determined.

Also note that we do not restrict the domain *Dom* for the local variables in the state machines to be finite. The entry statement in each state can have multiple sends which can be encoded as a separate send state in EDA connected by local internal transitions. The nondeterministic choice statement can be encoded in the form of nondeterminism on internal transitions. The set of out going transition in each P state can be easily mapped on to transitions in EDA. Actions in P can be expanded as a state transition logic implementing the action handler. Similarly, the call statements and call transitions can be handled by repeating the sub-state machines at all call points. By encoding a stack in the local state of the machines, we can model function calls in P programs, in our automaton.

Every machine P_i in an EDA has an error state q_i^{err} that can be used to model local assertions in the P program. An important safety property in P programs is to check the responsiveness of the system, i.e., for every receive state, if m is the first message in the queue that is not deferred, then there should be a receive action enabled from this state that handles m. Checking if a P program is responsive can be easily reduced to checking that the error state q_i^{err} is not reachable, for all $i \in N$.

The upshot of the above relationship is that the reduction algorithms for EDAs described in the earlier sections can be easily lifted to P programs. States in P can perform multiple actions within the state (such as internal actions and sending multiple messages), but these can be broken down into smaller states to simulate our reduction.

6. Implementation

We have implemented our ASI reductions by adapting the ZING model-checker [4]. The P compiler translates P programs into ZING models, preserving the input programs execution model. The explorer in ZING supports guided-search based on a scheduler that is external to the model checker. The ASI reduction in ZING is implemented in the form an external ASI scheduler that guides the explorer on which set of actions are enabled in the current state and the explorer then iterates over these actions. The ZING program is instrumented appropriately to communicate the current state configuration information to the ASI scheduler. This instrumentation is performed automatically by our modified P compiler. The model is instrumented to pass information such as (1) the current state of each state machine, whether its in a send or a receive state (2) size of the message queues, etc. Based on the current state of each state machine, and the communication pattern amongst the machines, the ASI scheduler calculates the destination set mentioned in Section 4. Using this destination set, the set of next actions to be performed are prioritized by the ASI scheduler and executed by the ZING explorer.

The implementation of the reduction can be seen as a composition of an almost synchronous ASI scheduler and the asynchronous Zing model, exploring only the state space of the composite system. Most part of the ASI reduction can be implemented as being external to the model checker except for the case when a state machine is pushed into a *blocked* state. The blocking of a state machine is part of the state of the system and is handled as a special case. A special state machine called *blocking-state-machine* is created with respect to each state machine in the model. The job of the *blocking-state-machine* is to enqueue a special event *block* in the associated state machine. Each state machine in P is extended to handle block event in all states, such that on dequeuing the block event it enters a new state where it keeps dropping all enqueued messages. Now the ASI scheduler can block a state machine by simply scheduling the corresponding *blocking-state-machine* and atomically executing the transitions enqueuing and dequeueing the block event.

7. Evaluation

In this section we present an empirical evaluation of the ASI reduction approach for verification of P programs and also evaluate it for finding bugs. All the experiments reported are performed on Intel Xeon E5-2440, 2.40GHz, 12 cores (24 threads), 160GB machine running 64 bit Windows Server OS. Zing can exploit multiple cores during exploration as its iterative depth-first search algorithm is completely parallelised[43]. The timing results reported in this section are when Zing is run with 24 threads and uses iterative depth bounding by default for exploring the state space.

In order to thoroughly evaluate our ASI technique, we applied it on models from various domains. We used P for writing all our benchmarks, and used the P compiler to generate Zing models for verification. Our benchmark suite includes:

- The Elevator controller model as described in [9]
- The OSR driver used for testing USB devices
- The Truck lifts distributed controller protocol;
- Time synchronization standards protocol used for synchronization of nodes in distributed systems
- The German cache coherence protocol, and
- The Windows Phone(WP) USB driver, which is the actual driver shipped with the Windows Phone operating system.

Note that the lines of code reported in Table 1 are for models written in P which is a domain specific language designed for writing protocols compactly. We could not evaluate our approach on Windows 8 USB driver used in [9] as it was not available to us. We however could evaluate our technique on the Windows Phone USB driver under a license agreement.

7.1 Proving P programs:

Message buffers in P can become unbounded and the systematic exploration by Zing fails to prove P programs correct in the presence of such behaviors [9]. In general, the queues can become unbounded when a state machine pumps in events at arbitrarily fast rates. For Zing to be able to explore such models, P users were allowed to provide bound on the maximum occurrence of an event in any queue. This indirectly bounds the queue size of each state machine during the state space exploration.

Table 1 shows the results for the Zing Bounded Model Checker [9] as well as our ASI based reduction technique. The Zing results are only for an under-approximation of the state space, restricted by bounding the maximum number of occurrences of an event to a constant value that was picked on the basis of domain knowledge. On the other hand, the ASI based reduction results are for complete verification of the models, where message buffers are unbounded. For ASI, we report the total number of states explored, the time taken by the tool, and whether it was able to prove the programs correct.

We found in the ASI exploration that the size of queues never exceeded 4, indicating that the queues remain bounded to a small size under our reduction. On the other hand, even after bounding the queue sizes, Zing could not prove large P programs correct or took a very long time.

P is being used for development of the Windows Phone USB Drivers in Microsoft. The most surprising result here is that our reduction-based technique was able to verify that this driver is receptive (i.e., there is no reachable configuration where a machine receives a message that it cannot handle).

Using ASI based reduction we were able to verify the Windows Phone (WP) driver and the German protocol, while Zing failed to explore the state space completely (even when message buffers we bounded). Also, for comparatively smaller models, ASI was able to prove the models correct faster because of the large state space reduction obtained. Zing is a state of the art explicit state model checker tuned for efficiently exploring P programs. It uses state caching to avoid re-explorations but does not implement partial-order reduction techniques, as prior experience suggested that partial-order techniques were not very useful in this domain. As described in Section 2, partial-order reduction can easily fail to keep message buffers small (as in the producer-consumer scenario) and hence often leads to infinite state-spaces, which precludes exhaustive search.

7.2 Bug finding in P programs:

To demonstrate the soundness of our approach, we created buggy versions of the benchmark models by introducing known safety errors in them. Table 2 shows results in terms of the number of states explored and the time taken before finding the bug, with and without ASI. The search terminates as soon as a bug is found. Notice that our reduction technique explores orders of magnitude less states and also finds bugs faster for all the models. The experiments suggest almost-synchronous reductions may also be a good pri-

Models	Lines of code in P	Zing Model Checker (with buffer bounds)				Almost-synchronous Invariants (with *no* buffer bounds)		
		Bound on max occurrence of an event in queue	Total number of states	Time (h:mm)	State-space exhaustively Explored?	Total number of states	Time (h:mm)	Program Proved Correct?
Elevator	270	2	1.4×10^6	0:22	Yes	2.8×10^4	**0:08**	**Yes**
OSR	377	2	3.1×10^5	0:16	Yes	3.9×10^3	**0:02**	**Yes**
Truck Lifts	290	2	3.3×10^7	2:07	Yes	1.1×10^5	**0:24**	**Yes**
Time Sync (Linear Topology)	2200	4	7.4×10^{10}	5:34	Yes	1.0×10^7	**3:07**	**Yes**
German	280	3	$> 1 \times 10^{12}$	*	No	4.7×10^8	**2:32**	**Yes**
Windows Phone USB Driver	1440	3	$> 1 \times 10^{12}$	*	No	2.4×10^9	**3:48**	**Yes**

** denotes timeout after 12 hours*

Table 1: Results for proof based on almost-synchronous invariants for P.

Buggy Models	Zing Bounded Model Checker (with buffer bounds)				Almost-synchronous Invariants (with *no* buffer bounds)		
	Bound on max occurrence of an event in queue	Total number of states	Time (h:mm)	Bug Found?	Total number of states	Time (h:mm)	Bug Found?
Truck Lifts	2	950005	1:17	Yes	13453	**0:14**	Yes
Time Sync (Ring Topology)	4	*	*	No	129973	**1:37**	**Yes**
German	3	595723	0:44	Yes	2345	**0:10**	Yes
Windows Phone USB Driver	3	1616157	2:04	Yes	23452	**0:38**	Yes

** denotes timeout after 12 hours*

Table 2: Results for bug finding using almost-synchronous invariants for P

oritization strategy for finding errors faster. For the Time Sync model with nodes in a ring topology, iterative depth bounding in Zing failed to find the bug while ASI was able to find it. Traditionally in model checking, iterative depth bounding[43] has been used as an efficient way for finding bugs quickly. We implemented ASI in Zing, on top of iterative depth bounding, and found that we outperform it.

8. Related Work

The reachability problem for finite state machines communicating via unbounded fifo queues is undecidable [8]. The undecidability stems from the fact that the unbounded queues can be used to simulate the tape of a Turing machine. To circumvent this undecidability barrier, there has been work in several directions. It has been shown that the problem becomes decidable under certain restrictions like when the finite state machines communicate via unbounded *lossy* fifo queues that may drop messages in an arbitrary manner [2], when the communication is via a bag of messages and not via fifo queues [19, 39], when only one kind of message is present in the message queues [35], when the language of each fifo queue is bounded [17], or when the communication between the machines adheres to a forest architecture [22]. For verification of these machines communicating via messages, techniques that over-approximate the set of reachable states have also been studied [8, 34].

Several under-approximate bounding techniques have been explored to find bugs, even when the machines have shared memory, including depth-bounding [13], bounded context-switching reachability [22, 36, 37], bounded-phase reachability [7], preemption-bounding [31], delay-bounding [10], bounded-asynchrony [11], etc. These techniques systematically explore a bounded space of reachable states of the concurrent system and are used in practice for finding bugs. It has also been shown that several of these bounding techniques admit a decidable model-checking problem even when the underlying machines have recursion [21, 27, 36]

Unlike the above bounding techniques which are not complete, partial order reduction (POR) methods retain completeness while trying to avoid exploring interleavings that have the same partial order [14, 29]. POR techniques use persistent/stubborn sets [15, 44] or sleep sets [16] to selectively search the space of reachable states of the concurrent system in a provably complete manner. Dynamic POR [12] and its variants [23, 24, 32, 42] including the recently proposed optimal one [1] significantly improve upon the earlier works by constructing these sets dynamically. Dynamic POR for restrictions of MPI programs where synchronous moves are sufficient have also been explored in [32, 40, 41].

Message sequence charts (MSC), which provide a specification language for specifying scenarios of different communication behaviors of the system, have a partial order semantics, and high-level message sequence charts can combine them with choice and recursion. While checking linear time properties of scenarios of these graphs is undecidable [3], surprisingly, checking MSO properties over MSCs directly was shown to be decidable in [25]. Furthermore, this kind of model-checking can be done using linearizations that keep the message buffers bounded [26], similar to the almost-synchronous interleavings explored in this paper.

The work reported in [19] solves the problem of data flow analysis for asynchronous programs using an under-approximation and over-approximation bounding the counters representing the pending messages, where messages are delivered without in non-fifo order. The authors in [5, 6] present a technique where choreography of asynchronous machines can be checked when the asynchronous communication can be replaced by synchronous communication. The authors use their analysis technique for verifying channel contracts in the Singularity operating system [18]. Though our approach of almost-synchronous reduction has a similar flavor, we do not restrict our analysis to systems where asynchronous message passing can be entirely replaced by synchronous communication.

Our present work builds on top of P [9], which is a language for writing asynchronous event-driven programs. While [9] uses a model checker to systematically test P programs for receptiveness, our reduction technique provides a methodology for *verifying* P programs. In addition, our experiments strongly suggest that our reductions can be also used to find bugs much faster.

9. Conclusions and Future Directions

We have shown an sound and complete reduction for asynchronous event-driven programs that can effectively control the size of message buffers, leading to faster techniques to both prove and find bugs in programs. Exploring almost-synchronous interleavings that grow the buffers only when they really need to grow seems to capture more interesting interleavings as well as discover smaller adequate invariants.

There are several interesting questions worthy of future study. First, we believe that partial-order reductions can help further reduce the number of interleavings, once almost-synchronous reductions have curtailed the blow-up due to unbounded message-buffers. Combining partial-order reduction techniques with almost-synchronous reductions would be worthwhile. Secondly, though almost-synchronous invariants often suffice, there are instances where *environment* machines can flood message buffers. We believe that in these cases, a simple *over-approximation* of these channel contents will be sufficient in proving programs correct. Finding a tractable but adequate approximation scheme would be useful. Third, in the framework of event-driven programs, it would be interesting to compare ASI with other under-approximate bug-finding techniques that work by bounding scheduling metrics such as delay bounding[10] and context bounding[30]. Also, since ASI is an independent reduction technique, there is potential in combining it with these bounding techniques for finding bugs faster. Finally, there are several other message-passing domains where we believe that our reductions could be useful; in particular, analysis of truly distributed programs (such as protocols for replicated database systems in the cloud) and analysis of MPI programs for verifying high-performance computing algorithms could benefit from our technique.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. POPL '14, pages 373–384, New York, NY, USA, 2014. ACM.

[2] P. A. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170, 1993.

[3] R. Alur and M. Yannakakis. Model checking of message sequence charts. CONCUR '99, pages 114–129, London, UK, UK, 1999. Springer-Verlag.

[4] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV*, pages 484–487, 2004.

[5] S. Basu and T. Bultan. Choreography conformance via synchronizability. WWW '11, pages 795–804, New York, NY, USA, 2011. ACM.

[6] S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In *VMCAI*, pages 56–71, 2012.

[7] A. Bouajjani and M. Emmi. Bounded phase analysis of message-passing programs. TACAS'12, pages 451–465, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, Apr. 1983. ISSN 0004-5411.

[9] A. Desai, V. Gupta, E. K. Jackson, S. Qadeer, S. K. Rajamani, and D. Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332, 2013.

[10] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. POPL '11, pages 411–422, New York, NY, USA, 2011. ACM.

[11] J. Fisher, T. A. Henzinger, M. Mateescu, and N. Piterman. Bounded asynchrony: Concurrency for modeling cell-cell interactions. In *FMSB*, pages 17–32, 2008.

[12] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[13] P. Godefroid. Model checking for programming languages using verisoft. POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[14] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem.* PhD thesis, University of Liege, 1995.

[15] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). CAV '93, pages 438–449, London, UK, UK, 1993. Springer-Verlag.

[16] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[17] M. G. Gouda, E. M. Gurari, T. H. Lai, and L. E. Rosier. On deadlock detection in systems of communicating finite state machines. *Comput. Artif. Intell.*, 6(3):209–228, July 1987. ISSN 0232-0274.

[18] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007. ISSN 0163-5980.

[19] R. Jhala and R. Majumdar. Interprocedural analysis of asynchronous programs. POPL '07, pages 339–350, New York, NY, USA, 2007. ACM.

[20] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4): 596–619, 1983.

[21] S. La Torre, P. Madhusudan, and G. Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170, 2007.

[22] S. La Torre, P. Madhusudan, and G. Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS*, pages 299–314, 2008.

[23] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha. A framework for state-space exploration of java-based actor programs. ASE '09, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3891-4.

[24] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. FASE'10, pages 308–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[25] P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. ICALP '01, pages 809–820, London, UK, UK, 2001. Springer-Verlag.

[26] P. Madhusudan and B. Meenakshi. Beyond message sequence graphs. In *FSTTCS*, pages 256–267, 2001.

[27] P. Madhusudan and G. Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294, 2011.

[28] P. Madhusudan, X. Qiu, and A. Stefanescu. Recursive proofs for inductive tree data-structures. In *POPL*, pages 123–136, 2012.

[29] A. W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324, 1986.

[30] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[31] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[32] R. Palmer, G. Gopalakrishnan, and R. M. Kirby. Semantics driven dynamic partial-order reduction of mpi-based parallel programs. PADTAD '07, pages 43–53, New York, NY, USA, 2007. ACM.

[33] E. Pek, X. Qiu, and P. Madhusudan. Natural proofs for data structure manipulation in c using separation logic. In *PLDI*, 2014. To Appear.

[34] W. Peng and S. Puroshothaman. Data flow analysis of communicating finite state machines. *ACM Trans. Program. Lang. Syst.*, 13(3):399–442, July 1991. ISSN 0164-0925.

[35] W. Peng and S. Purushothaman. Analysis of a class of communicating finite state machines. *Acta Inf.*, 29(6/7):499–522, 1992.

[36] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. TACAS'05, pages 93–107, Berlin, Heidelberg, 2005. Springer-Verlag.

[37] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[38] X. Qiu, P. Garg, A. Stefanescu, and P. Madhusudan. Natural proofs for structure, data, and separation. In *PLDI*, pages 231–242, 2013.

[39] K. Sen and M. Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. CAV'06, pages 300–314, Berlin, Heidelberg, 2006. Springer-Verlag.

[40] S. F. Siegel. Efficient verification of halting properties for mpi programs with wildcard receives. VMCAI'05, pages 413–429, Berlin, Heidelberg, 2005. Springer-Verlag.

[41] S. F. Siegel and G. S. Avrunin. Modeling wildcard-free mpi programs for verification. In *PPOPP*, pages 95–106, 2005.

[42] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag.

[43] A. Udupa, A. Desai, and S. K. Rajamani. Depth bounded explicit-state model checking. In *SPIN*, pages 57–74, 2011.

[44] A. Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.