A New Reduction for Event-driven Distributed Programs

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 [4] correct for the receptiveness property, which was hitherto not provable using state-of-the-art model-checkers.

1. Introduction

Asynchronous, event-driven programming paradigm, which involves concurrently evolving components communicating using messages and reacting to input events, is a popular paradigm that is widely used today to model distributed programs communicating on a network (eg., client-server communication, web programs) or even programs (like device drivers) running on a single-processor or a multi-core system. In this work, we wish to build techniques that provably verify such asynchronous event-driven programs against local assertions. There are many sources of infinity in the verification of event-driven programs— the local data, the message buffer sizes, and the *number* of spawned processes being the primary ones, and in general the problem is undecidable. Our primary concern in this work 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 (it is known that the reachability problem for this restricted setting consisting of finite state machines communicating via unbounded message buffers is also undecidable [2]).

Though the problem at hand and the proof technique we describe in this paper is very general, in this work, we specifically look at programs written in P [4], which is a domain-specific language for event-driven programs developed recently at Microsoft Research for building the Windows 8 USB driver stack. P programs consist of a collection of interacting state machines that communicate with each other by exchanging messages. The primary specification that P programs are required to satisfy in [4] 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 [4] includes a *systematic testing* tool for model-checking device driver models for the responsiveness property. However, such model-checking seldom succeeds in proving the drivers correct, since there are many sources of infinity, including message buffer sizes.

Consider the simple scenario where a machine p sends a machine q unboundedly many messages, like in a producer-consumer setting (See Figure 1). 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 [1] used to systematically test P programs in [4] will fail to finish, since the message buffers are unbounded and are part of the global state that's explored.

Partial-order reduction (POR) techniques [6, 7] have been traditionally used to reduce the number of interleavings while exploring the set of reachable states for such concurrent systems. In an ideal scenario, these techniques explore every partial-order using just one linearization. However, POR techniques do not necessarily ensure termination, and are of little help in this case. In the above example, a send from p to q followed by the receive takes the system back to its initial state and hence induces a cycle in the state space. In view of this, after the send from p to q has been explored, the *ample set* of transitions [3, 6] would consist of the receive as well as the successive send. Each global state along the execution that takes the send would be *different* because the message buffer content is different in each step, and the exploration will not terminate.

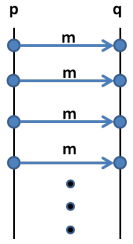

Figure 1: Producer Consumer Scenario

Almost-synchronous Invariants: Our primary thesis is that *almost-synchronous invariants* often suffice to prove asynchronous event-driven programs correct with respect to local assertions, 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 covered. 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 our work is a sound and complete reduction scheme that, unlike POR, explores interleavings (involving almost-synchronous states) that keep message buffers to a minimal size, while at the same time finding a closure argument that argues that all local states have been discovered, at which point we can terminate.

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 [2]. 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. Finding these almost-synchronous invariants often suffices in capturing the dynamics of event-driven, asynchronous programs and natural proofs that target finding such invariants can prove their correctness efficiently. We discover almost-synchronous invariants in this work using state-space exploration and model-checking for a variety of asynchronous systems.

Implementation and Evaluation: We have implemented our reduction mechanism for discovering almost-synchronous invariants for P programs. Our invariant synthesis is built over the Zing model-checker [1], 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) [4] 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. 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. Main Idea: Almost-Synchronous Reductions

The main idea of this paper is that almost-synchronous invariants suffice to find proofs of local assertions in event-driven asynchronous programs. We present a *reduction* technique that is sound and complete and that constructs these invariants as a collection of almost-synchronous states of the system.

Prioritizing receive events. The first rule of our almost-synchronous reduction 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. In the producer-consumer scenario in Figure 1, our reduction explores 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 (due to receive events being prioritized), 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, which includes the local states of all machines and the contents of all message buffers, repeats and our exploration procedure will terminate. This is entirely because the message buffer gets constantly depleted causing the global state to recur.

However, exploring synchronous interleavings (where all sends enabled always have the matching receive events immediately enabled and scheduled in the receiving process) does not always suffice to prove a system correct, which is why we need to explore almost-synchronous global states and not just synchronous ones. As an example, con-

sider the scenario in Figure 2 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 [4] 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 after a mild asynchronous excursion.

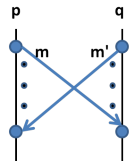

Figure 2:

Enabling a subset of send events. What happens when there are no receive events enabled in the current state? Enabling all send events in this case is an option, but one that might lead to the problem of unnecessarily flooding message buffers. Our reduction technique on the other hand choses a *subset* of these send events (based on the possible communication amongst the machines in the system) such that enabling just this subset of events is both sound and complete.

To understand this, let us consider the example in Figure 3— 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. In this case, our reduction will schedule the synchronous send event from q to r. However to ensure completeness, our reduction might also need to enable the send from p to q depending upon whether the machine p eventually sends a message to r or not. Consider the case where 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 the synchronous send from q to r in the current state. Therefore, depending upon this statically determined communication information, if p is a potential sender of r, our reduction enables both the sends of p and q. On the other hand, if p is not a potential sender of r (and the dotted arrow does not exist), our reduction enables just the synchronous send from q to r.

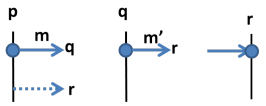

Figure 3:

In general, in every state, our reduction computes a subset of machines called the *destination set* based on the possible communication amongst the machines in the system, as well as the communication pattern amongst them in current state. From a given state, our reduction only enables those events that send messages to machines in the destination set. In figure 3, when p is a potential sender of r, the destination set for the current state is $\{r, q\}$ and, as mentioned above, our reduction enables all the send transitions to them, i.e., sends from p to q and from q to r.

Blocking a subset of machines. Whenever a reduction enables only a subset of events from a given state, if there are cycles in the original state space of the system, there is a chance of missing exploring local states that are reachable along (sequences of) actions that are not enabled. To circumvent this source of incompleteness and explore such local states, our reduction allows a move that *blocks* all machines whose sends to the destination set were enabled/prioritized. A machine once blocked is not allowed to transition any longer. Also once blocked, all processes that are sending messages to it 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. For the example in Figure 3, the machines blocked in the current state will be p and q.

Given a system $\mathcal{P}$, our reduction mechanism constructs and explores a transition system $\mathcal{P}_{\mathcal{R}}$ such that the set of reachable states of $\mathcal{P}_{\mathcal{R}}$ correspond to the reduced set of global configurations of $\mathcal{P}$ that form an almost-synchronous invariant of the system. As alluded to above, the states in $\mathcal{P}_{\mathcal{R}}$ are of the form (C, B) where C is a configuration of the original system $\mathcal{P}$ and B is a subset of blocked machines. The informal algorithm for our reduction is as follows.

Given that the system $\mathcal{P}_{\mathcal{R}}$ 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 $X = \textit{destination-set}(C, B)$. Then we schedule *all* send events 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.

Let $Reach_G$ be the set of reachable configurations of the original system $\mathcal{P}$, and let Bad_G be the set of its error configurations. Similarly, let $Reach_R$ be the set of configurations of $\mathcal{P}_{\mathcal{R}}$ explored by our reduction and let $Bad_R = \{(C, B) \mid C \in Bad_G\}$. Then we can show that our reductions is both sound and complete. Formally, we can prove the following theorems:

Theorem 2.1 (Soundness)**.** *If some state* $(C_e, B_e) \in Reach_R \cap Bad_R$*, then there exists a configuration* $C' \in Reach_G \cap Bad_G$*.*

Theorem 2.2 (Completeness)**.** *If some configuration* $C \in Reach_G \cap Bad_G$*, then there exists* C', B' *such that* $(C', B') \in Reach_R \cap Bad_R$*.*

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

3. Evaluation

In this section we briefly present an empirical evaluation of our reduction approach for the 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. We evaluate our approach on the following P programs: the elevator controller model described in [4], 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.

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 [4]. Table 1 compares our reduction technique to the exploration using the ZING bounded model checker [4] on all the above mentioned P programs. We report the ZING results for an under-approximation of the actual state space that bound the maximum number of occurrences of an event in message buffers to a constant value. On the other hand, our results are for the complete verification of the models, where message buffers are unbounded.

Using our reduction technique, we were able to completely verify the Windows Phone (WP) driver and the German protocol, while ZING failed to explore the state space completely for these systems (even when message buffers were bounded). We found that in our almost-synchronous exploration, the size of message buffers never exceeded four, indicating that the buffers remain bounded to a small size under our reduction. Even for comparatively smaller models, we were able to prove the models correct in much smaller time as compared to ZING because of the large state space reduction obtained.

Finding bugs 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 compares results for our exploration technique to the iterative depth-bounding [8] in ZING in terms of the number of states explored and the time taken before finding the bug. On our benchmarks, our reduction technique explores orders of magnitude less states and finds bugs much faster than ZING for all the models. We expect a comparison with other bounding techniques like delay-bounding [5] to give similar results and a detailed comparative study is part of future work. Our experiments suggest that almost-synchronous reductions may also be a good prioritization strategy for finding bugs.

4. Conclusions

We have shown a 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.

References

[1] 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.

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

[3] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-03270-8.

[4] 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.

[5] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.

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

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

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