Distributed Explicit State Model Checking of Deadlock Freedom

Brad Bingham[1], Jesse Bingham[2], John Erickson[2], and Mark Greenstreet[1]

[1] University of British Columbia,
Department of Computer Science
{binghamb,mrg}@cs.ubc.ca
[2] Intel Corporation
{jesse.d.bingham,john.erickson}@intel.com

Abstract. This paper presents a practical method and associated tool for verifying deadlock freedom properties in guarded command systems. Such properties are expressed in CTL as **AGEF** q where q is a set of quiescent states. We require the user to provide transitions of the system that are "helpful" in reaching quiescent states. The distributed search constructs a path consisting of helpful transitions from each reachable state to a state that is either quiescent or is known to have a path to a quiescent state. We extended the PReach model-checker with these algorithms. Performance measurements on both academic and industrial large-scale models shows that the overhead of checking deadlock-freedom compared with state-space enumeration alone is small.

Keywords: distributed model checking, murphi, deadlock-freedom, liveness.

1 Overview

Automatic checking of liveness properties is a challenging task. Approaches to address this generally require the user to carefully specify system *fairness* assumptions that are necessary for liveness to hold. Furthermore, checking liveness is computationally expensive, being more sensitive to the state-space explosion problem than simple state-space enumeration. A broad class of liveness failures of practical importance is *deadlock*, wherein one or more transaction is blocked due to a cyclic resource dependency [1]. In such a state, there exists no path to a state where all transactions have completed; this is our motivation for characterizing deadlock-freedom by a property **AGEF** q.[1]

PReach [2,3] is a distributed explicit-state model checker for systems described in the Murφ modelling language [4]. At a high level, PReach implements the Stern-Dill algorithm [5] for message passing based, parallel state-space enumeration. This algorithm statically partitions the state-space according to a

[1] Some literature and tools identify deadlock with the much weaker property that all reachable states have at least one (possibly unique) successor. We use the stronger from, **AGEF** q throughout this paper.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 235–241, 2013.

random uniform hash function that assigns states to *owner* threads, the owner of a state being responsible for storing it once the state is *expanded*, i.e., had its successors computed. PReach is designed to be scalable, extensible and robust, and is capable of checking models with billions of states using hundreds of heterogenous machines. Several mechanisms are necessary for handling such large-scale models such as load balancing, flow control methods and state batching. The message passing layer of PReach is implemented in the distributed functional language *Erlang*, and Murφ's C++ libraries are borrowed for certain computationally demanding tasks.

A new feature added to PReach and the focus of this paper is an explicit state model checking technique to verify the CTL [6] property **AGEF** q. This property (of recent interest [7]) says "*for all reachable states, there exists a path to some q-state*". In our approach to verifying **AGEF** q, the system is modeled using guarded commands, and the user identifies a subset, $\mathcal{H}$, of these commands as *helpful*. These commands are the ones that the user expects will cause the system to make progress towards q. If s is a state and $s' \neq s$ can be reached from s by performing a helpful command, then we say that s' is a *helpful successor* of s. Thus, from any reachable state s_1,[2] we look for a *witness path*: If s_1 is a q-state, then the path is trivial; otherwise, PReach computes s_2, a helpful successor of s_1. If s_2 is a q-state then we have found a witness path for s_1; otherwise, s_3, a helpful successor of s_2 is computed. This process iterates, building a witness path $\rho = s_1, s_2, \ldots$ until a state s_i is found where either

- s_i is a q-state, or
- s_i has no helpful successor (referred to as $\mathcal{H}$-*stuck*), or
- s_i already appears in ρ.

In the first case, ρ is a witness path for s_1 and PReach continues with its standard state-space exploration algorithm. If a witness path is found for every reachable state then **AGEF** q necessarily holds. In the other two cases, PReach halts and reports the path ρ to the user. These cases do not imply $\neg$**AGEF** q. For example, if the helpful rule list is empty and there exists a reachable state that is not a q-state, then nontrivial witness paths will never be found. Likewise, PReach may choose a sequence of transitions that leads to a cycle even though a path to a q-state exists. While either error could be a false negative, as we report in Section 4, in practice such failures can show that behaviours of the model are not those intended by the designer and thereby reveal real errors.

In our experience, guarded command models have a clear partition between commands that inject new requests and those that service existing ones. Therefore, it is easy to decide suitable $\mathcal{H}$ and q. We show through experimentation that using only helpful commands to form paths is not only sufficient to verify **AGEF** q, but is *efficient* relative to the performance of state-space enumeration.

It is critical for performance to leverage the known witness paths during subsequent searches. Suppose a witness path ρ_1 has been found for state s, and s is

[2] PReach can also verify the more general property **AG** $(p \rightarrow$ **EF** $q)$, but for this paper we assume for brevity that p is *true*.

encountered on path ρ_2 while searching for witness path for s'. Clearly, the concatenation of paths ρ_1 and ρ_2 is a witness path for s'. PREACH uses a dedicated state hash table for this purpose, called **EFHT**. Each time a witness path is found for some state, it is added to **EFHT**; when we check if some state s_i is a q-state, we also check for membership in **EFHT**. Henceforth, we use $q \vee$ **EFHT** to denote states that are either q-states or members of **EFHT**.

1.1 Related Work

The idea of iteratively firing certain commands to complete in-flight transactions is very similar to the *completion functions* used by Park and Dill [8], though their goal was to verify refinement. As far as we are aware, the two tools closest to PREACH are Divine [9] and Eddy [10]. Neither of these tools are capable of checking CTL properties such as **AGEF** q, and to the best of our knowledge, neither has been applied to large scale problems as has been done with PREACH [2]. Our approach differs from the classical CTL model checking algorithm [6], which performs a pre-image fix-point computation from q to compute the set of states that satisfy **EF** q, and then checks that the reachable states are contained in this set. By using a forward search, we ensure that the space and time complexity is proportional to that of doing (explicit) state-space exploration. Doing CTL model checking only using forward searches has been investigated for symbolic model checking, e.g. the work of Iwashita et al. [11].

2 Implementation

Extending this core idea work with distributed model checking is non-trivial, and can be done in several ways. We now summarize the approaches we considered.

2.1 Local Search

This search involves no communication to other threads during a witness search. When a process in the distributed reachability algorithm encounters a new state s, the process computes a path ρ as described in Section 1. This path computation is not distributed across processors, and thus, redundant paths computions occur across different processors. While this approach scales poorly when the reachability analysis is run on a large number of machines, it provides a baseline that is free from communication overhead.

2.2 Pass-the-Path

Pass-the-path (PP) distributes the witness path searches by forwarding the current path prefix to the owner of the next state. When a state s is found in the reachability analysis, if it is already in $q \vee$ **EFHT** then a witness path is known to exist and no further work is needed. Otherwise, an enabled helpful rule is chosen; the successor state, s' is computed, and the search message $([s], s')$ is

sent to the owner of s'. Here, $[s]$ is a list of states representing the current prefix path. This process continues constructing a prefix path ρ, communicating search messages of the form (ρ, s_{cur}) to the owner of s_{cur}, until a member of $q \vee$ **EFHT** is reached, a cycle is encountered, or no helpful commands are enabled. In the first case, the owners of all states along the path are notified and they update their **EFHT**s, and otherwise a failure is reported. Notice that PP allows redundant searches and acknowledgments to occur because threads keep no record of which states have pending searches for paths to q-states.

2.3 Outstanding Search Table

These redundant searches can be avoided if threads keep track of which states have outstanding witness searches. We have implemented such an approach where each thread maintains the pending searches in local table **ST**. This approach called OST has the benefit of search messages containing only a pair of states (s, s'). If such a message arrives and there is a pending search for s' in the table, then s is added to a list of states that must be acknowledged as having a witness path once s' is acknowledged.

3 Performance

We ran PReach on a variety of combinations of Murφ models and hardware configurations, summarized in Table 1. For each, we measured the performance of

Table 1. The column "runtime" is the mean runtime of three trials, with the exception of the large cluster runs which are based on one trial (marked with †). The "overhead" columns are the additional runtime relative to that of the "no EF" mode run of the same model run on the same hardware. In PP mode, column "avg. path" is the number of helpful transitions needed to reach a state that is a member of $q \vee$ **EFHT**, averaged over the searches launched for each non-q-state. This number is *also* the average number of times each non-q-state is acknowledged for insertion to **EFHT**.

model	hardware	no EF	local	PP		OST	#states
		runtime	overhead	overhead	avg. path	overhead	
`german9`	multicore	1365.12	0.76	0.16	1.005	0.21	19844513
`flash5`	multicore	752.75	1.07	0.30	1.005	0.39	24063542
`peterson12`	multicore	4018.34	1.11	timeout	-	0.38	116039964
`mcslock6`	multicore	230.09	1.12	0.43	1.093	0.67	12838266
`german9`	small cluster	85.91	3.43	0.24	1.642	0.35	19844513
`flash5`	small cluster	57.46	1.52	0.32	1.307	0.52	24063542
`flash6`	small cluster	1854.42	1.56	0.23	1.021	0.29	609827554
`peterson12`	small cluster	254.27	9.50	timeout	-	0.50	116039964
`mcslock6`	small cluster	22.04	1.45	2.73	4.763	1.09	12838266
`intel_small`†	large cluster	1025.20	7.10	0.84	2.242	0.55	22738573
`intel_large`†	large cluster	49041.70	timeout	0.57	-	-	906695343

regular state-space enumeration (no EF), local search mode (Section 2.1), Pass-the-Path mode (Section 2.2) and Outstanding-Search-Table mode (Section 2.3). The Murφ models used are the German and Flash cache coherence protocols, the Peterson mutual exclusion algorithm, the MCS lock mutual exclusion algorithm and an Intel proprietary cache coherence protocol. We use `germanX` and `flashX` to denote these models configured with `X` caches and two data values; `peterson12` is Peterson's algorithm with 12 threads and `mcslock6` is the MCS Lock algorithm with 6 threads. All benchmarks and PReach source code is available online[3]. The compute server farms are as follows:

- multicore: 8 PReach threads on a 2 socket server machine, each processor is a Intel® Xeon® E5520 at 2.26 GHz with 4 cores.
- small cluster: 80 PReach threads on a homogenous cluster of 20 Intel Core i7-2600K at 3.40 GHz with 4 cores.
- large cluster: 100 PReach threads on a heterogenous network of contemporary Intel® Xeon® machines.

4 Summary

We have shown an efficient distributed algorithm and implementation for checking deadlock freedom properties. The simpler approach of PP does some redundant work, but performs well on models of cache coherence where paths to q-states tend to be relatively short (see Figures 1 and 2). The approach is intolerably slow for the Peterson mutual exclusion algorithm where these paths are much longer. In this case, our more involved OST algorithm has a favorable runtime and manageable memory overhead (as shown on the right of Figure 2). We find that using OST to check deadlock freedom is inexpensive – at most a 109% runtime penalty but typically much smaller.

The utility of our approach was underscored when a counterexample trace was generated on the `peterson12` benchmark. After carefully checking our definitions of q and $\mathcal{H}$, we found a critical typo in the Murφ model, which despite being

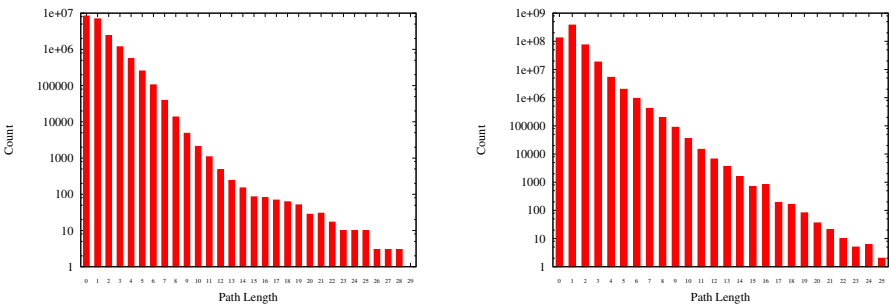

Fig. 1. Semi-log histograms of path lengths in PP mode, as defined in Table 1. The left is from `german9`/multicore, the right is from `flash6`/small cluster.

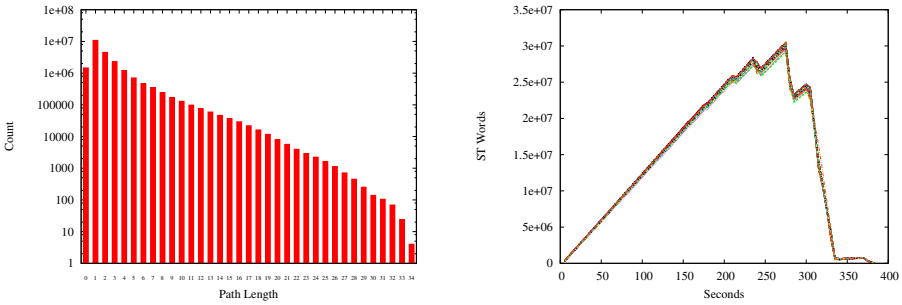

Fig. 2. Left: semi-log histogram of path lengths for `intel_small`/large cluster in PP mode. Right: memory usage of **ST** for `peterson12`/small cluster in OST mode. Here, "words" are 8 bytes; the peak memory usage over all threads is about 233 MB.

an example model in the popular Murφ distribution, has persisted for nearly 20 years. Interestingly, the bug in question was not revealed by checking the safety property mutual exclusion, or by "Murφ deadlock" (states with no successors) or *even* by checking **AGEF** q. With the bug, there exists a state where a thread attempting to enter the critical section may not do so until another thread makes the attempt first. Thus, the model does not satisfy $\mathbf{AGEF}_{\mathcal{H}}\ q$.

Checking **AGEF** q has a "buy-one, get-one free" appeal. Once model checking has been done for safety properties, a small amount of human effort is needed to identify helpful rules and write a quiesence predicate, q. While this approach cannot check for subtle liveness errors, especially ones that rely on fairness constrains, deadlocks and violations of designer intent can be found as illustrated by the Peterson example.

References

1. Holt, R.C.: Some deadlock properties of computer systems. ACM Computing Surveys 4(3), 179–196 (1972)
2. Bingham, B., Bingham, J., de Paula, F.M., Erickson, J., Singh, G., Reitblatt, M.: Industrial strength distributed explicit state model checking. In: Parallel and Distributed Model Checking (2010)
3. Bingham, B., Bingham, J., Erickson, J.: Preach online (2013), `https://bitbucket.org/binghamb/preach-brads-fork`
4. Dill, D.L.: The murphi verification system. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 390–393. Springer, Heidelberg (1996)
5. Stern, U., Dill, D.L.: Parallelizing the murphi verifier. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 256–278. Springer, Heidelberg (1997)
6. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
7. Hassan, Z., Bradley, A.R., Somenzi, F.: Incremental, inductive CTL model checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 532–547. Springer, Heidelberg (2012)

8. Park, S., Dill, D.L.: Protocol verification by aggregation of distributed transactions. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 300–310. Springer, Heidelberg (1996)
9. Barnat, J., Brim, L., Češka, M., Lamr, T.: CUDA accelerated LTL Model Checking. In: ICPADS 2009. IEEE (2009)
10. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. Int'l. J. Softw. Tools Technol. Transf. 11(1), 13–25 (2009)
11. Iwashita, H., Nakata, T., Hirose, F.: CTL model checking based on forward state traversal. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 82–87. Springer, Heidelberg (1996)