Brief Announcement: MP-State: State-Aware Software Model Checking of Message-Passing Systems

Can Arda Muftuoglu, Péter Bokor, and Neeraj Suri

Technische Universität Darmstadt,
Darmstadt, Germany
{arda,pbokor,suri}@cs.tu-darmstadt.de

Introduction. Software model checking [4] is a useful and practical branch of verification for verifying the implementation of the system. The wide usability comes at a price of low time and space efficiency. In fact, model checking of even simple single-process programs can take several hours using state-of-the-art techniques [6]. Verification complexity gets even worse for concurrent programs that simultaneously execute loosely coupled processes. Verification efficiency can be greatly improved by capturing the state of the program, a technique generally referred to as *stateful* model checking [2]. Intuitively, state capture enables to detect that two states are identical and, therefore, to consider only a representative state for verification. Unfortunately, capturing the state in general software systems can be very hard, even if the entire state of the system resides in the (local) memory. As a result, certain verification approaches (commonly called stateless model checking) do not capture the system's state at all [4]. Stateful model checking is in principle possible for software, however, at a price of considerable overhead. Therefore, stateful model checking is efficient only if the achieved reduction of redundantly explored states compensate for the overhead.

Our focus is on fault-tolerant message-passing protocols, a class of systems that can particularly benefit from formal verification for various mission-critical applications. Although the verification of fault-tolerant message-passing protocols is known to be a hard problem due to concurrency and faults, model checking has proven to be an efficient approach to debug and verify small instances of deployed protocols [5].

In this brief announcement, we propose the state capture algorithm MP-State, which improves software model-checking of general message-passing protocols. MP-State makes use of two techniques that enable time- and space-efficient model checking. The first technique is a *selective hashing* mechanism that captures state information only if this might interfere with the specification. The second technique is a *selective push-on-stack* strategy, which is an optimization that filters the states that are pushed onto the search stack and, hence, are subject to backtracking. Selective push-on-stack is sound because filtered-out states have no unvisited successor states.

Motivating Example. We give the intuition behind the proposed approach through a simple message-passing example with two processes, p_1 and p_2. Process p_1 sends two messages m_1 and m_2 to process p_2. Process p_2 stores in its local state the messages it receives. It is possible for m_2 to arrive later than m_1 at p_2 due to network delays and p_2 can process available messages (m_1 and m_2) in one atomic step. Having received m_1 and m_2, p_2 sends an ack message to p_1, informing that it has successfully received the messages sent by p_1.

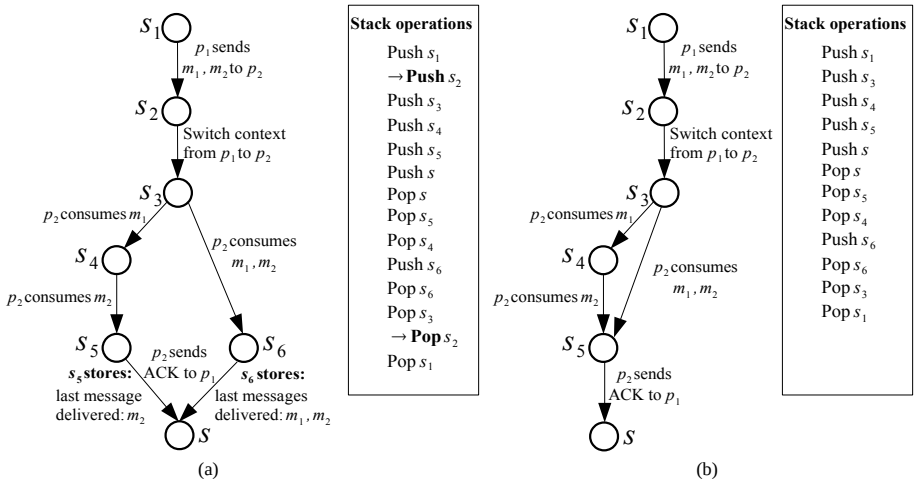

Fig. 1: (a) Naive depth-first search (DFS) and (b) MP-State search.

Figure 1(a) shows the state graph of the protocol as explored by a naive depth-first search (DFS) and the corresponding operations of the search stack. We observe that software model checkers can utilize *auxiliary variables* for the implementation of the model checking process. These variables are not specified by the protocol under test. For example, in Basset and MP-Basset [1], an auxiliary variable stores the messages delivered by a transition that is scheduled for execution. As a result, s_5 and s_6 are different states, with the overhead of storing two states and exploring the successor state s two times. In addition to auxiliary variables, model checkers may have *auxiliary transitions*. Auxiliary transitions are the transitions that are "independent" from the protocol under test. For example, Basset and MP-Basset uses auxiliary transitions for the purpose of switching context between processes, which is related to the model checker, not to the protocol. As a result, states involved in the execution of such transitions (such as s_2 and s_3 in Figure 1) are considered by DFS as any other state.

Selective hashing. We observe that (a) the transitions of common message-passing protocols depend only on the local states of the processes and pending (undelivered) messages; and (b) the usual properties of these protocols concern only about local states. Therefore, it is sufficient to capture local states and pending messages of each visited state. We refer to this technique as *selective hashing*. In our example, the state graph resulting from selective hashing is shown in Figure 1(b). Note that states s_5 and s_6 collapse into the same state because p_1 and p_2 have the same local states in both states and the set of pending messages is empty. The gain of selective hashing is that (i) different states resulting from differing values of auxiliary variables have to be processed only once by the model checker, e.g., for successor states of s_5 and s_6, which is s, and (ii) it is time efficient because state capture does not need to process the entire state.

Selective push-on-stack. We also observe that (c) usually auxiliary transitions are not concurrent with other transitions and (d) auxiliary transitions and states where these transitions are executed do not have to be remembered for counterexamples. Therefore,

states with enabled auxiliary transitions do not have to be pushed onto the search stack. We refer to this technique as *selective push-on-stack*. Consider the auxiliary transition t from s_2 to s_3 in our example. Since t is the only transition that can be executed in s_2, no state remains unvisited if s_2 is not backtracked by the search. Also, a path excluding s_2 and t preserves all protocol-specified information. The application of selective push-on-stack to our example leads us to the search stack in Figure 1(b), where s_2 is not involved in any stack operation. Note that selective push-on-stack visits the same states as the naive search but it is more time efficient thanks to fewer stack operations.

MP-State and other reductions. Broadly-studied and intuitive reductions are partial-order (POR) [3] and symmetry reductions (SR) [7]. Figure 1 demonstrates that MP-State is not a special case of these reductions. Firstly, POR is based on the idea of swapping the order of commutative transitions but the path ($s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_6 \rightarrow s$) that is excluded in the reduced state graph in Figure 1(b) cannot be obtained by re-ordering the transitions of another path in the graph. Formally, considering the mainstream POR semantics, Figure 1(b) is not a stubborn/persistent/ample set reduction of (a) because in every state of the reduced state graph the number of enabled transitions is the same as in the unreduced one.

Secondly, SR is based on the symmetrical structure of the state graph but there is no such symmetry in Figure 1(a). Formally speaking, there is no permutation acting over the set of states (the formal notion of symmetry [7]) that would preserve the transition relation. In fact, in order to symmetry reduce Figure 1(a) into (b), a permutation would have to transpose s_5 and s_6 but these two states are not "symmetric" because of s_4.

Our achieved reductions up to 69%. Our evaluation of MP-State with deployed fault tolerant message-passing protocols (Paxos consensus, distributed storage, and atomic broadcast) fortifies our initial claim that despite its overhead, stateful model checking outperforms stateless model checking. Besides, the results of our experiments show that MP-State is highly efficient, achieving a reduction of model checking time and memory by up to 69% over naive (unreduced) stateful model checking with depth-first search. In one of our experiments, we managed to reduce model checking time from 22 hours 19 minutes to 10 hours 22 minutes.

References

1. P. Bokor, J. Kinder, M. Serafini, N. Suri. Efficient Model Checking of Fault-Tolerant Distributed Protocols. *Proc. of DSN-DCCS*, pp. 73-84, 2011.
2. E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 2000.
3. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.
4. P. Godefroid. Model Checking for Programming Languages using VeriSoft. *Proc. of POPL*, pp. 174–186, 1997.
5. H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, L. Zhang. Practical Software Model Checking via Dynamic Interface Reduction. *Proc. of SOSP*, pp. 265-278, 2011.
6. V. Kuznetsov, J. Kinder, S. Bucur, G. Candea. Efficient State Merging in Symbolic Execution. *Proc. of PLDI*, pp. 193-204, 2012.
7. A. Miller, A. Donaldson, M. Calder. Symmetry in Temporal Logic Model Checking. *ACM Computing Surveys*, 38(3), 2006.