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

Abstract. The verification of concurrent programs is known to be hard where the execution of concurrent components (processes) can interleave. Therefore, practical verification of concurrent programs primarily focuses at various forms of partial-order reductions leveraging all possible interleavings. In this paper, we turn our attention towards another aspect of concurrent program verification, namely, capturing the state of the program. We show that efficient state capture can significantly improve the overall efficiency of verification, also, in addition to partial-order reductions.

In particular, 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; we prove that the non-interference property implies the soundness of selective hashing. 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. Our evaluation of MP-State with deployed fault-tolerant message-passing protocols shows a reduction of model checking time and memory by up to 69%, and 45% when used with partial-order reduction. In contrast to partial-order reductions, MP-State makes practically no assumptions about the message-passing protocol.

1 Introduction

Software model checking [11, 13] 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 (or go off-scale) using state-of-the-art techniques [20]. Verification complexity gets even worse for concurrent programs that simultaneously execute loosely coupled processes. For example, current results report that sound verification of implementation-level distributed protocols is feasible only in very small deployments [15, 24].

Verification efficiency can be greatly improved by capturing the state of the program, a technique generally referred to as stateful model checking [8]. 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 [11, 12, 14]. Stateful model checking is in principle possible for software [16, 27], however, at a price of considerable overhead. Therefore, stateful model checking is efficient only if the achieved reduction of redundantly explored states compensates for the overhead. In addition, state capture can be used to detect cycles; as a result, stateful model checking can be used to verify cyclic programs where the stateless search would run forever [8].

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. Example applications include different services in large-scale [26, 18] or embedded [25] fault-tolerant settings. 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. In fact, subtle bugs and incorrect specifications have been found in real deployments using model checking [26] and in prototype implementations of various fault-tolerant message-passing protocols [4, 5, 24]. Although some of these results utilize stateful model checking [4, 5, 24], the efficiency of the stateful over stateless model checking has not been studied for this kind of systems.

In this paper, we measure the efficiency of stateful model checking using the Java implementation of various fault-tolerant message-passing protocols. Our measurements with Java Pathfinder (JPF) [27] show that stateful model checking can be faster by several orders of magnitude compared to the stateless search. Secondly, we observe that the efficiency of stateful model checking can be further improved by only capturing specification-relevant state information. In fact, existing state capture implementations for message-passing protocols (such as in Basset [22] and MP-Basset [4]) might capture two states as different that are indistinguishable by the specification of the system. We propose MP-State, a new model checker that implements sound and efficient state capture for message-passing systems. In the following, we briefly introduce the key techniques of MP-State.

The first technique is called selective hashing and it is based on the observation that software model checkers attach auxiliary information to certain states to ease the implementation of the search. Our technique consists of capturing in each state the pending (undelivered) messages and the local states of each process, but not implementation-specific auxiliary information.\(^1\) The soundness of the approach is based on the assumption that the specification only depends on pending messages and local states. The benefit of selective hashing is that only a fraction of the state is captured. As a result, the overhead of state capture is reduced and, more importantly, the state capture detects more states as identical.

The second technique is called selective push-on-stack and it is based on the observation that software model checkers may execute auxiliary transitions that are needed to conduct the search but that are independent of the specification of the message-passing

\(^1\) The name of selective hashing is because the implementation of state capture usually means serializing and hashing states.
protocol. Our push-on-stack strategy does not push states that are involved in executing auxiliary transitions onto the depth-first search stack. As a result, selective push-on-stack reduces the overhead of stack operations, which results in reductions of the overall model checking time. As selective push-on-stack checks that auxiliary transitions are not concurrent with other transitions, states that are not pushed onto the stack (and, thus, are not backtracked) result in no new successor states. Therefore, the search with selective push-on-stack misses no states.

Specifically, the paper makes the following contributions:

- We demonstrate that stateful model checking is efficient using a selection of fault-tolerant message-passing protocols (see below). Our experiments show the efficiency of stateful model checking, even for acyclic versions of the protocols, by reducing model checking time (and memory) by several orders of magnitude.
- We propose MP-State, a new state capture algorithm and a new model checker implementing the algorithm. MP-State contains two novel techniques for efficient state capture, namely, selective hashing and selective push-on-stack. We formally prove the soundness of MP-State with respect to a general class of properties of message-passing protocols.
- We implement MP-State as an extension of MP-Basset [4] and JPF [27]. Our implementation is efficient, as shown by the experiments. It is also a general implementation enabling extensions for other optimizations such as symmetry reduction [7].
- We evaluate MP-State with various fault-tolerant message-passing protocols, namely Paxos consensus [21], Zab atomic broadcast of Yahoo’s Zookeeper [18], and a regular register in the style of ABD [1]. Our experiments justify that MP-State improves on stateful model checking by reducing model checking time and memory by up to 69%. In addition, MP-State is also efficient when used together with partial-order reduction [10] by achieving additional reductions up to 45%.

2 Motivating Example

We give the intuition behind the proposed approach through the example of a simple message-passing protocol with two processes, one sender process \( p_1 \) and one receiver process \( p_2 \). 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, independent of the order of messages.

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.\(^2\) 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 the model checkers Basset [22] and MP-Basset [4, 5], an auxiliary variable stores the message delivered by a transition that is scheduled for execution. As a result, \( s_5 \) and \( s_7 \) are different states, with the overhead of storing two states and exploring the successors of both states. Recall that the local state of process \( p_2 \) ignores the order in which \( m_1 \) and

\(^2\) In a real implementation, deadlock states (such as \( s_5 \)) need not be pushed/popped onto/from the search stack.
Naive DFS

Stack operations
Push s_1
Push s_2
Push s_3
Push s_4
Push s_5
Push s_6
Push s_7

Pop s_7
Pop s_6
Pop s_5
Pop s_4
Pop s_3

Switch context from p_1 to p_2

s_5, stores: last message delivered: m_2

s_7, stores: last message delivered: m_2

MP-State

Stack operations
Push s_1
Push s_2
Push s_3
Push s_4
Push s_5
Push s_6

Pop s_6
Pop s_5
Pop s_4
Pop s_3
Pop s_2
Pop s_1

Switch context from p_1 to p_2

S_1
S_2
S_3
S_4
S_5
S_6
S_7

S_1
S_2
S_3
S_4
S_5
S_6
S_7

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

m_2 have been delivered. In addition to auxiliary variables, model checkers may have auxiliary transitions. For example, Basset and MP-Basset uses auxiliary transitions for the purpose of switching context between processes. 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 running example, the state graph resulting from selective hashing is shown in Figure 1(b). Note that states s_5 and s_7 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 (1) 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_7, and (2) it is time efficient because state capture does not need to process the entire state.

Selective push-on-stack. We also observe that (c) an auxiliary transition is 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

---

3 In this context, a counterexample is a path witnessing the reachability of a state that is disallowed by the specification.
3 MP-State

3.1 System Model

Basic model. We adopt a natural and general model of message-passing protocols [2, 3]. The model consists of processes that communicate via messages. Every process maintains a local state. A local state is a tuple of variables and values assigned to each of these variables. The local state of a process can be updated by executing guarded transitions. Every transition is associated with a process. A transition \( t \) is executed if the guard of \( t \) evaluates to true. The execution of a transition is an atomic event, which consumes zero or more messages received by the executing process, changes the local state, and sends multiple messages on behalf of the process. We call a message pending if it was sent but not yet consumed by a transition. Both the guard and the execution of a transition only depend on the pending messages and the local states of the processes.

Given states \( s, s' \) and transition \( t \), we use the notation \( s \xrightarrow{t} s' \) to denote that the execution of \( t \) in \( s \) results in \( s' \). To simplify the discussion, we assume that transitions are deterministic, i.e., given a state \( s \) and a transition \( t \) enabled in \( s \), there is a single state \( s' \) such that \( s \xrightarrow{t} s' \).

The (global) state of the system executing the protocol is a tuple of all local states, the pending messages, and values assigned to auxiliary variables. Transitions might change the values of auxiliary variables. In the example of Section 2, the last message consumed by process \( p_2 \) is an auxiliary variable. Initially, the state is \( s_I \), where there is no pending message and the local state of each process and the values of auxiliary variables are assigned a default value. The model is extendible with more than one initial state.

Semantics/paths. The semantics of the model of a message-passing protocol is given by its state graph. A state graph consists of the states and one state \( s \) is connected to another state \( s' \) if there is a transition such that \( s \xrightarrow{t} s' \). A path of the model is a finite sequence \( s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_n \). A state \( s \) is reachable if there is a path that starts in \( s_I \) and ends in \( s \). We assume that the state graph contains finitely many states. By definition, \( s_I \) is reachable.

The user can distinguish auxiliary transitions. A protocol path can be obtained from a path \( \sigma \) by removing all auxiliary transitions from \( \sigma \) and the states where they are executed. For example, consider transition \( t \) in Figure 1(a) with \( s_2 \xrightarrow{t} s_3 \). Transition \( t \) can be considered as auxiliary if the user is not interested in the transition that switches context between processes \( p_1 \) and \( p_2 \). In this case, a protocol path from \( s_1 \) to \( s_4 \) is \( s_1 \xrightarrow{t_1} s_3 \xrightarrow{t_2} s_4 \) where the following paths exist: \( s_1 \xrightarrow{t_1} s_2 \) and \( s_3 \xrightarrow{t_2} s_4 \).
Properties. The protocol can specify properties. We assume that these properties are state properties.\footnote{State properties correspond to invariants in general verification terms.} We will justify this assumption in Section 4. A state property is a predicate over the local states of the processes. A state property $f$ holds in a model of a message-passing protocol if $f$ is true in every reachable state.

3.2 MP-State Algorithm

The algorithm describing the core idea behind MP-State is displayed in Algorithm 1. The notations that are used throughout the algorithm are shown in Table 1. For simplicity, the algorithm assumes that the set of pending messages and local states of each process are available in the initial state and after executing every transition. This assumption can be implemented, for example, by using messaging abstraction [24].

In principle, MP-State modifies a naive depth-first search algorithm between lines 1-15. The first modification is selective hashing in line C2. Instead of adding every newly visited state $s$ to the set of visited states ($reached$), a state $s'$ is added that only contains values of pending messages and local states in $s$.

The second modification is selective push-on-stack in line C1. Instead of pushing every newly visited state $s'$ onto the search stack, $s'$ is only pushed if the transition enabled in $s'$ is not an auxiliary transition. For soundness, we check that this transition is the only enabled transition in $s'$. Otherwise, the search may miss some states that are reachable from $s'$.

3.3 MP-State Properties

We now introduce the main properties of MP-State algorithm. First, we prove that selective push-on-stack explores the same state graph as the unreduced search. This implies that state properties are preserved by selective push-on-stack. The benefit of using selective push-on-stack is that it is able to reduce the time overhead of model checking, as shown in our evaluation (Section 4).

We use the following notations: given a model $MP$ of a message-passing protocol, $Unreduced(MP)$ is obtained from Algorithm 1 by removing the condition statements from lines C1 and C2. $Selective-Push-On-Stack(MP)$ is obtained from Algorithm 1 by removing the condition statement from line C2.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$protocol_transition(t)$</td>
<td>is true iff $t$ is not an auxiliary transition</td>
</tr>
<tr>
<td>$v \in s$</td>
<td>returns variable $v$ of state $s$</td>
</tr>
<tr>
<td>$pending_messages(v)$</td>
<td>is true iff $v$ is the variable representing the pending messages</td>
</tr>
<tr>
<td>$local_state(v)$</td>
<td>is true iff $v$ is the variable representing the local state of a process</td>
</tr>
<tr>
<td>$stack_push(s)$</td>
<td>adds state $s$ onto stack</td>
</tr>
<tr>
<td>$stack_pop()$</td>
<td>returns the topmost item from stack</td>
</tr>
<tr>
<td>$enabled(s)$</td>
<td>set of enabled transitions in $s$</td>
</tr>
<tr>
<td>$next(enabled(s))$</td>
<td>non-deterministically returns one transition from $enabled(s)$</td>
</tr>
<tr>
<td>$s(v)$</td>
<td>returns the value of $v$ in $s$</td>
</tr>
<tr>
<td>$s'_append(s(v))$</td>
<td>appends $s(v)$ to state $s'$</td>
</tr>
</tbody>
</table>

Table 1: Summary of notations used in Algorithm 1
function $MP$-State($MP$)
1    Stack stack $\leftarrow \emptyset$;
2    Set reached $\leftarrow \emptyset$;
3    State $s \leftarrow s_I$;
4    stack.push($s$);
5    while stack $\neq \emptyset$ do
6        while enabled($s$) $\neq \emptyset$ do
7            Transition $t \leftarrow \text{next}(\text{enabled}(s))$;
8            enabled($s$) $\leftarrow \text{enabled}(s) \setminus \{t\}$;
9            State $s' \leftarrow s$;
10           $s' \leftarrow \text{Selective_Hashing}(s')$; //Selective Hashing
11           if $s' \not\in$ reached then
12              reached $\leftarrow$ reached $\cup \{s'\}$;
13              if protocol_transition(\text{next}(\text{enabled}(s'))) $\lor |\text{enabled}(s')| > 1$ then
14                 stack.push($s'$);
15              s $\leftarrow$ s';
16              stack.pop();
17        function Selective_Hashing(State $s$)
18           State $s' \leftarrow \emptyset$;
19           forall $v \in s$ do
20              if pending_messages($v$) $\lor \text{local}_\text{state}(v)$ then
21                  $s'$.append($s(v)$);
22           return $s'$;

Algorithm 1: $MP$-State algorithm for message-passing protocol $MP$

Theorem 1 (State graph preservation). Given a model $MP$ of a message-passing protocol, Selective-Push-On-Stack($MP$) explores the same state graph as Unreduced($MP$).

Proof. See in Appendix.

The following theorem proves that $MP$-State is sound for state properties. In other words, $MP$-State can be used for the verification of state properties without missing bugs and, also, without falsely concluding the truth of a state property.

Theorem 2 (Soundness). Given a model $MP$ of a message-passing protocol and a state property $f$, $f$ holds in the state graph explored by Unreduced($MP$) if and only if $f$ holds in the state graph explored by $MP$-State($MP$).

Proof. See in Appendix.

The meaning of the following property is that $MP$-State is able to witness the reachability of all reachable combinations of local states and pending messages. More formally, every reachable state $s$ is represented by an equivalent state $s'$ in the $MP$-State
search and a search can return a protocol path leading to \( s' \). The equivalence of states is meant in terms of selective hashing: states \( s \) and \( s' \) are equivalent, and we write \( s =_{sh} s' \), if and only if \( \text{Selective Hashing}(s) = \text{Selective Hashing}(s') \). Note that the debugger of the protocol is given all relevant information as state properties are independent of the values of auxiliary variables (excluded by selective hashing); and protocol paths are valid paths except that they exclude auxiliary transitions.

**Theorem 3 (Debugging).** If state \( s \) is reachable in a model of a message-passing protocol MP, then MP-State(MP) visits a state \( s' \) such that \( s =_{sh} s' \) and a protocol path leading to \( s' \) is on the search stack when \( s' \) is visited.

**Proof.** See in Appendix.

### 3.4 Implementation

We implement MP-State as an extension of MP-Basset [4], which is an efficient software model checker for message-passing systems written in Java. The state space search and state matching of MP-State, as in MP-Basset is implemented by JPF [27]. The source of MP-State is available online [30]. Next, we detail the implementation of the two techniques of MP-State:

**Selective hashing.** JPF’s stateful optimization consists of (i) a serializer, which is used to serialize the states, and (ii) a hash function, which takes the output of the serializer as input to hash the given state. In JPF, the serializer contains two data structures, namely, buffer and queue. The buffer is used to store the values of all the variables of a state. Initially, the queue contains the references of the topmost classes of the program. The reference fields of these classes are added to the queue to be recursively processed and the primitive field values are added to the buffer. Afterwards, the output of the serializer, which is the buffer, is passed as an input to the hash function. JPF uses Jenkin’s hashing function [17] to compute the hash value of the given buffer.

We modify JPF’s serializer to implement our selective hashing technique. We observe that the queue (and buffer) of JPF’s serializer is unstructured. As a result, we cannot select the fraction of local states and pending messages in a state. Therefore, we implement a structured queue (and buffer). We do so by finding the pointers to the classes of (a) each process and (b) each pending message. We can then serialize these classes one by one and pass the resulting data to the hash function.

**Selective push-on-stack.** In JPF, concurrent transitions (transitions enabled in the same state) are specified using choice generators. Depending on the interactions between processes (which are Java threads), they are obtained automatically by JPF (using a coarse over-approximation of concurrency) or they are registered by the user.

MP-State borrows the concept of transitions from MP-Basset where transitions of the protocol are registered using special choice generators. In our implementation of selective push-on-stack, these are the choice generators that constitute protocol (not auxiliary) transitions. As a result, transitions such as the ones that do context switching between threads are considered as auxiliary transitions. For soundness, our implementation checks that auxiliary transitions are not concurrent with other transitions, as in Algorithm 1 (line C1).
4 Evaluation of MP-State

In this section, we evaluate MP-State with representative fault-tolerant message-passing protocols. We measure the gain of MP-State compared to the state-of-the-art and highly optimized model checker MP-Basset [4, 5]. The evaluation compares model checking time and memory (the number of visited states) for MP-Basset and MP-State.

**Overall evaluation strategy.** Firstly, to show the efficiency of stateful model checking, we evaluate and compare the stateless and stateful model checking results in MP-Basset. As the stateful optimization is shown to be efficient, we can use it as the base case for measuring the efficiency of MP-State. We evaluate MP-State for selective hashing and selective push-on-stack to compare their added benefits. Partial-order reduction (POR) [8] is a widely-used optimization of model checking and is implemented by MP-Basset. Therefore, we also evaluate MP-State combined with POR to see if it improves POR.

4.1 Target Protocols and Properties

Our evaluation is based on the following protocols: Paxos consensus [21], a regular register protocol in the style of ABD [1], and Zab atomic broadcast [18]. We argue that these protocols constitute a representative and practical selection of fault-tolerant large-scale protocols. Firstly, these are all crash-tolerant protocols. The crash fault-model is widely used, also because a large and practical class of non-crash faults can be transformed into crash faults, as shown by a recent result [9]. Secondly, Paxos, regular register, and Zab are conceptual and/or known to be practically relevant. For example, Paxos algorithm is in the core of commercial replication services [26], or the Zab protocol is part of Yahoo’s Zookeeper open-source library used in different real deployments [28]. As the implementation of none of the protocols is available to us, we use our prototype Java implementation in each case.

In our evaluation, we use the following settings of the above protocols. We also discuss the properties that are subject to our model checking experiments with MP-State. In addition to the protocols and their specified properties, we fault inject each protocol and/or its properties to evaluate the debugging feature of MP-State.

- **Paxos.** In Paxos, every process can be one or more of the following types: proposer, acceptor, and learner. We evaluate MP-State with the following Paxos instances. The first instance, Paxos (6), has two proposers, three acceptors, and one learner. Every proposer makes at most one proposal. The second instance, Faulty-Paxos (6), is like Paxos (6) but an acceptor (wrongly) accepts all proposals. The third instance, Faulty-Paxos2 (7) contains three proposers and a subtle bug: an acceptor does not remember the highest numbered proposal it has ever accepted but only the last proposal it has accepted. We model check Paxos for agreement, the main safety property of consensus, which can naturally be expressed as a state property. We remark that the bug of Faulty-Paxos2 (7) cannot be found with Paxos (6) because at least three proposals are needed for the bug to violate agreement.

5 Although Zookeeper is an open-source project, the code of Zab cannot be extracted as a stand-alone protocol.
• **Register.** The register protocol implements a single-writer register. The *regularity* property of the register guarantees that a read operation returns a value not older than the one written by the latest preceding write operation. We evaluate three instances of the register: Register (5), Register (6) and Register (7) with three, four and five fault-prone storing objects, respectively, and one reader. The writer (the reader) initiates at most one write (read) operation. In addition, we fault inject the regularity property, which we call wrong regularity, by requiring that a read operation that completes after a write has to return the value written by the write even if the two operations are concurrent. As regularity is a safety property, it can be expressed as a state property.

• **Zab.** Even the smallest meaningful Zab instance turns out to be infeasible (time-out after 192 hours) for exhaustive search in our experimental setup. Therefore, instead of proving that Zab implements atomic broadcast, we show that Zab is not live. In fact, Zab does not guarantee that every request is eventually delivered and we challenge the model checker to find a counterexample of liveness. We remark that, arguably [18], finding a counterexample manually involves considerable expertise in the protocol. Zab distinguishes between leader and follower processes. We evaluate two instances of Zab, both with three followers, Zab (6) and Zab (7) with three and four leaders, respectively. Although liveness in general cannot be expressed as a state property, the provided counterexamples prove the violation of liveness [24].

### 4.2 Experimental Setup

We run our experiments in a Deterlab testbed [29] on machines with 2GHz Dual Xeon processors and 2GB memory, running on Ubuntu v.10.04. We compare the execution times and total number of visited states of different reduction types for each protocol.

• **MP-Basset stateless/stateful.** We evaluate stateless model checking to compare it with stateful model checking to see if the overhead of stateful model checking is compensated by the gain in time. Note that, the search always terminates due to experimenting with only acyclic examples.

• **MP-State without selective push-on-stack.** MP-State without selective push-on-stack experiments show the added benefit of push-on-stack technique exclusively. This technique does not achieve state reduction. The aim of this experiment is to see whether the selective push-on-stack technique achieves time reduction.

• **MP-State.** MP-State experiments apply selective hashing and selective push-on-stack techniques. We check if MP-State improves efficiency compared to the stateful MP-Basset search.

• **POR.** POR experiments are our base case to compare with the experiments where MP-State and POR are used together. We use POR wherever it is applicable. For example, MP-Basset’s implementation of POR does not apply for Zab (see more details in Section 4.3 about the assumptions made by POR and MP-State).

---

6 For space reasons, we do not display the results of these experiments for fault-injected cases.

7 MP-Basset implements different POR algorithms: static and dynamic. We use static POR for our experiments, because it is known to be more efficient than dynamic POR for fault-tolerant message-passing protocols [5].
• **MP-State and POR.** The purpose of this type of experiment that combines MP-State and POR is to see if MP-State is efficient with POR.

We use the following notations to display our results (Table 2). We write OK if the model checker proves that the property holds for the given instance of the protocol, otherwise a counterexample (CE) is returned. In buggy instances, the search is stopped after finding the first bug, hence the search is non-exhaustive. We write N/A (not available) if POR is not available for the experiments or the reduction percentage is not available due to timeout in the experiment. For exhaustive searches that end with time-out (192 hours), the value in the states column indicates the number of visited states at the time when the search stops.

4.3 **Reduction Results**

The results of our experiments are shown in Table 2. Our main observations are as follows:

• **Stateful outperforms stateless search.** The stateful search finishes earlier than the stateless one in all exhaustive and fault-injected experiments. In some cases (e.g. Paxos(6)), stateful search terminates where the stateless search is infeasible (given our timeout). In other cases, stateful model checking reduces the search time by up to 94% compared to stateless model checking.

• **MP-State improves efficiency.** MP-State is highly efficient as shown by the exhaustive search results, reducing the total number of visited states by up to 57% and model checking time by up to 54%. It also finds bugs up to 69% faster than stateful model checking.

• **Selective push-on-stack time efficient.** Selective push-on-stack reduces model checking time by up to 17%.

• **MP-State efficient with POR.** When MP-State is used with POR, MP-State reduces model checking time and memory by up to 39%, compared to the experiments with only POR.

**Disclaimer.** The reduction achieved by POR can be significantly more than by MP-State. For example, POR reduces model checking time by 98% for Paxos (6), whereas MP-State achieves a reduction of 54%. This is true only given the assumptions made by POR. In fact, POR is based on the assumption that the execution of certain transitions is commutative [10]. The soundness of POR can only be guaranteed if this assumption is verified. In general, verifying the assumption of commutative transitions is as hard as model checking. MP-State, in contrast to POR, makes no assumptions about the commutativity of transitions (see also in Section 5). Note that the simple static analysis in MP-Basset’s POR implementation [5] is not applicable for Zab to verify the assumptions required by POR. MP-State is still applicable in this case and it achieves a time reduction of up to 69%.

**Further insights.** We observe that the reduction achieved by MP-State is constant as the number of processes increases. In other words, the more states the bigger the subset of them can be reduced by MP-State (selective hashing). In fact, the time reduction
<table>
<thead>
<tr>
<th>Protocol (num. of processes)</th>
<th>Result</th>
<th>Property</th>
<th>Reduction</th>
<th>States</th>
<th>Time</th>
<th>Time reduction *</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paxos (6)</td>
<td>OK</td>
<td>Agreement</td>
<td>MP-Basset stateless</td>
<td>&gt; 202,756,996</td>
<td>&gt; 192h</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>13,044,613</td>
<td>22h19m</td>
<td>91% (over SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State without SPoS</td>
<td>5,606,047</td>
<td>11h10m</td>
<td>54% (over SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>5,606,047</td>
<td>10h22m</td>
<td>98% (over SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>191,081</td>
<td>23m43s</td>
<td>39% (over POR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State + POR</td>
<td>117,369</td>
<td>14m22s</td>
<td>36% (over POR)</td>
</tr>
<tr>
<td>Faulty-Paxos (6)</td>
<td>CE</td>
<td>Agreement</td>
<td>MP-Basset stateless</td>
<td>&gt; 71,914,839</td>
<td>&gt; 192h</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>96,802</td>
<td>10m3s</td>
<td>94% (SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>3,050</td>
<td>36s</td>
<td>94% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>2,786</td>
<td>31s</td>
<td>14% (POR)</td>
</tr>
<tr>
<td>Faulty-Paxos2 (7)</td>
<td>CE</td>
<td>Agreement</td>
<td>MP-Basset stateless</td>
<td>&gt; 140,997,799</td>
<td>&gt; 192h</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>&gt; 66,651,310</td>
<td>&gt; 192h</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>129,533</td>
<td>21m18s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>124,976</td>
<td>19m29s</td>
<td>9% (POR)</td>
</tr>
<tr>
<td>Register (5)</td>
<td>OK</td>
<td>Regularity</td>
<td>MP-Basset stateless</td>
<td>21,582</td>
<td>1m59s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>3,376</td>
<td>27s</td>
<td>66% (SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>4,965</td>
<td>34s</td>
<td>66% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>1,936</td>
<td>24s</td>
<td>29% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State + POR</td>
<td>1,483</td>
<td>20s</td>
<td>17% (POR)</td>
</tr>
<tr>
<td>Register (5)</td>
<td>CE</td>
<td>Wrong regularity</td>
<td>MP-Basset stateless</td>
<td>46,009</td>
<td>6m14s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>4,580</td>
<td>54s</td>
<td>86% (SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>4,642</td>
<td>51s</td>
<td>90% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>4,642</td>
<td>48s</td>
<td>6% (POR)</td>
</tr>
<tr>
<td>Register (6)</td>
<td>OK</td>
<td>Regularity</td>
<td>MP-Basset stateless</td>
<td>1,451,439</td>
<td>1h7m</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>94,348</td>
<td>10m49s</td>
<td>85% (SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>56,222</td>
<td>8m14s</td>
<td>24% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>4,642</td>
<td>51s</td>
<td>90% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State + POR</td>
<td>4,642</td>
<td>48s</td>
<td>6% (POR)</td>
</tr>
<tr>
<td>Register (7)</td>
<td>OK</td>
<td>Regularity</td>
<td>MP-Basset stateless</td>
<td>187,564,795</td>
<td>&gt; 192h</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>35,692,316</td>
<td>&gt; 192h</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>2,986,657</td>
<td>8h21m</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>1,656,212</td>
<td>6h3m</td>
<td>28% (POR)</td>
</tr>
<tr>
<td>Zab (6)</td>
<td>CE</td>
<td>Liveness</td>
<td>MP-Basset stateless</td>
<td>46,009</td>
<td>6m14s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>4,580</td>
<td>54s</td>
<td>86% (SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>1,876</td>
<td>26s</td>
<td>38% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Zab (7)</td>
<td>CE</td>
<td>Liveness</td>
<td>MP-Basset stateless</td>
<td>88,681</td>
<td>10m21s</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-Basset stateful</td>
<td>8,198</td>
<td>2m4s</td>
<td>80% (SL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MP-State</td>
<td>3,132</td>
<td>38s</td>
<td>69% (SF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POR</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 2: Evaluation results of MP-State with/without selective push-on-stack (SPoS) compared with MP-Basset with/without stateful and partial-order reduction (POR) optimizations. Time reduction is computed with respect to base cases MP-Basset stateless (SL), stateful (SF), and stateful with partial-order reduction (POR). * State reduction is not shown as it is proportional to time reduction.

of “MP-State + POR” is 18%, 33%, and 28%, for the register with 5, 6, and 7 processes, respectively. One reason of this trend can be in the majority voting mechanism
that register (similarly to Paxos and Zab) uses for fault-tolerance. The majority of voters contains 2, 3 and 3 processes for Register (5), (6) and (7), respectively. A larger majority means more “equivalent” states for selective hashing because the writer has more choice in contacting different voters to observe the same voting result. This explains the improved reduction from 18% to 33%; and also the more or less constant reduction of 33% and 28%. Note that MP-State experiments (without POR) show a slightly different trend for the register: 25% and 23% for Register (5) and (6) (timeout for Register (7)). We speculate that there are collisions on the outputs of the hash function due to the large number of states in these experiments. As a result, MP-State cannot achieve the expected reduction.

Finally, as MP-State uses the same scheduling of the transitions as MP-Basset’s stateful model checking, MP-State explores the same state graph as MP-Basset modulo the reduction achieved by the latter search. As a result, the fault-injected experiments can also be considered as reductions.

5 Related Work

There exist many software model checkers and they come in different flavors. These include explicit model checkers for C programs such as Verisoft [11] and SPIN [16], or for Java [27], symbolic execution engines such as DART for C [13] or KLEE for low-level (bit)code [6], and dedicated solutions for message-passing systems such as Modist [26] or Mace [19]. Selective push-on-stack is inherently related to depth-first search and, as such, it can be implemented in any explicit state model checker (like Verisoft, SPIN, Modist, or Mace). On the other hand, selective hashing is not restricted to explicit state model checking as it can be used to decrease the number of variables needed for a symbolic encoding of the state. Our implementation of MP-State is specific to JPF [27] and it has to be tailored to the stateful search implementation of the model checker in use.

Existing model checkers for message-passing systems can benefit from using MP-State. We remark that model checkers that assume input models of the systems written in simplified programming languages (like the C-free segment of Promela, the input language of SPIN) would not gain much using selective hashing because the models usually already extract the protocol-specific variables of the system’s state.

To the best of our knowledge, all known reduction approaches that work with depth-first search, such as symmetry reduction (SR) [23], partial-order reduction (POR) [10], or dynamic interface reduction (DIR) [15], can be directly combined with MP-State without affecting soundness. Reductions of stateless model checking such as symmetric transitions [12] or dynamic partial-order reduction [14] would only benefit from selective push-on-stack. The benefit of MP-State compared to most existing reductions is that it makes no assumption about the transitions of the systems. In fact, POR assumes the commutativity of executing transitions, SR depends on symmetric execution patterns, DIR needs to be tailored depending on how the execution of different processes can interleave. The only assumptions made by MP-State is that (1) the protocol is specified in our very general system model and (2) the property of the protocol is a state property.
6 Conclusion and Future Work

We have proposed MP-State, an improvement of explicit-state software model-checking of general message-passing programs. Our evaluation of MP-State with various fault-tolerant message-passing protocols shows extensive reduction both in time and state space. We show that MP-State improves POR and unlike POR, MP-State requires no assumptions about the transitions of the system.

MP-State is a sound reduction for state properties. State properties, however, cannot express the general class of liveness properties. An extension of MP-State to cover the usual liveness properties of practical classes of message-passing protocols is future work. Another interesting extension of MP-State is towards symmetry reduction [23]. Most fault-tolerant message-passing protocols show natural symmetries in their (replicated) processes. Therefore, the structured queue (and buffer) of our implementation of selective hashing (see Section 3.4) can be used to detect and exploit such symmetries. Concretely, we plan to design static analysis for the detection and a canonicalization function for the exploitation of symmetries. Intuitively, once process-symmetries are detected and verified, the queue can be canonicalized into a representative state by permuting local states and the senders/ recipients of pending messages.

References

27. http://babelfish.arc.nasa.gov/trac/jpf/
Appendix: Proofs

We introduce the following notations. Given state property \( f \) and state \( s \), \( f(s) \) is a predicate that is true iff (if and only if) \( f \) holds in \( s \). Given paths \( \sigma = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_n \) and \( s_n \xrightarrow{t_{n+1}} s_{n+1} \xrightarrow{t_{n+2}} \ldots s_I \), the path \( s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots s_I \) can be written as \( \sigma = s_n \xrightarrow{t_{n+1}} s_{n+1} \xrightarrow{t_{n+2}} \ldots s_I \).

**Lemma 1.** Given the model MP of a message-passing protocol, if a state \( s' \) is visited in MP-State(MP) or Selective-Push-On-Stack(MP) and the condition of line 11 holds, then every \( t \in enabled(s') \) is executed in \( s' \).

**Proof.** Let \( t \) be a transition in \( enabled(s') \). If \( t \) is next(\( enabled(s') \)), then \( t \) is executed in the next iteration of the while loop in line 6. Otherwise, if \( enabled(s') > 1 \) (cf. condition of line C1), \( s' \) is pushed onto the stack and every transition \( t' \neq t \in enabled(s') \) is executed when \( s' \) is backtracked in the depth-first search. Note that \( s' \) is guaranteed to be backtracked after each transition executed in \( s' \) because the state graph is finite.

**Theorem 1 (State graph preservation).** See in Section 3.3.

**Proof.** The proof is indirect. Let \( M \) and \( M_Q \) be the state graphs explored by the unreduced (UR) and the selective push-on-stack search (SPoS), respectively. The initial state \( s_I \) is in both \( M \) and \( M_Q \). As \( reached \) is initially empty and so \( s_I \notin reached \), Lemma 1 implies that every successor of \( s_I \) is explored in \( M_Q \). As in every visited state \( reached \) contains the same states in UR and SPoS, the condition of line 11 is true in UR if it is true in SPoS. Therefore, it follows inductively that a path starting from \( s_I \) exists in \( M \) iff it exists \( M_Q \).

The following properties support soundness of MP-State.

**Lemma 2.** (1) Given two states \( s, s' \) and state property \( f \), \( s =_{sh} s' \) implies that \( f(s) \iff f(s') \). (2) In addition, given transition \( t \), \( s =_{sh} s' \) implies that \( t \in enabled(s) \iff t \in enabled(s') \). (3) Given \( s, s' \) and \( t \), if \( s =_{sh} s' \), \( s \xrightarrow{t} s_A \), and \( s' \xrightarrow{t} s'_A \), then \( s_A =_{sh} s'_A \). (4) Given states \( s, s', s'' \), \( s =_{sh} s' \) and \( s'' =_{sh} s'' \), if \( s =_{sh} s'' \).

**Proof.** (1) holds because a state property is a predicate over the local states of a state and, given any state \( s \), selective hashing selects the local states of all processes in \( s \). (2) holds because the guard of \( t \) is a predicate of the local states and pending messages, as selected by selective hashing. (3) holds because \( t \) is deterministic and its execution only depends on the state selected by selective hashing. (4) is because \( s(v) = s'(v) \) and \( s(v) = s''(v) \) imply \( s'(v) = s''(v) \).

**Theorem 2 (Soundness).** See in Section 3.3.

**Proof.** The proof is indirect. Let \( M \) and \( M_Q \) be the state graphs explored by the unreduced search and MP-State, respectively. Direction \( \Rightarrow \) holds because every state of \( M_Q \) is also in \( M \). The proof of direction \( \Leftarrow \) is illustrated in Figure 2. Assume that \( f \) holds in \( M_Q \) and there is a reachable state \( s_n \) in \( M \) where \( f \) does not hold. Let
\[ \sigma = s_1 \overset{t_1}{\rightarrow} s_2 \overset{t_2}{\rightarrow} \ldots s_n \text{ be a path leading to } s_n, \sigma \text{ is not in } M_Q, \text{ otherwise } f \text{ would not hold in } M_Q. \] We also know that } n > 1 \text{ because the initial state is explored by MP-State. Therefore, there must be } 2 \leq i < n \text{ such that } t_i \text{ is not executed in } s_i \text{ in } M_Q. \text{ Lemma 1 implies that the condition of line 11 of MP-State does not hold when } s_i \text{ is visited. This means that there is a state } s' \text{ reachable via a path } \sigma' \text{ in } M_Q \text{ such that } s' = \text{sh } s_i \text{ and } s' \not\in \text{reached} \text{ when visited in MP-State. If } i = n, \text{ then } \neg f(s') \text{ from Lemma 2(1). Therefore, } i < n. \text{ From Lemma 2(2), we know that } t_i \in \text{enabled}(s_i) \cap \text{enabled}(s'). \text{ Furthermore, } t_i \text{ is executed in } s' \text{ (Lemma 1) and it holds that } s_{i+1} = \text{sh } s'_1 \text{ where } s' \overset{t_i}{\rightarrow} s'_1 \text{ (Lemma 2(3)). Let } 0 < j_1 \leq n - i \text{ be the highest natural number such that } \sigma' \overset{t_{j_1}}{\rightarrow} s'_1 \overset{t_{j_1+1}}{\rightarrow} \ldots s'_{j_1} \text{ is a path in } M_Q. \text{ We know from Lemma 2(1-3) that } s'_{j_1} = \text{sh } s_{i+j_1}. \text{ Therefore, } j_1 = n - i \text{ would imply a contradiction (Lemma 2(1)). Since } j_1 \text{ is the highest such index, Lemma 1 implies that the condition of line 11 of MP-State does not hold when } s'_{j_1} \text{ is visited. So there must be a state } s^{j_1} \text{ reachable via a path } \sigma^{j_1} \text{ in } M_Q \text{ such that } s^{j_1} = \text{sh } s'_{j_1} \text{ and } s^{j_1} \notin \text{reached} \text{ when visited in MP-State. From Lemma 2(4), we know that } s^{j_1} = \text{sh } s_{i+j_1}. \text{ Let } 0 < j_2 \leq n - i - i_1 \text{ be the highest natural number such that } \sigma^{j_1} \overset{t_{j_1+j_2}}{\rightarrow} s'_1 \overset{t_{j_1+j_2+1}}{\rightarrow} \ldots s'_{j_2} \text{ is a path in } M_Q. \text{ We know from Lemma 1 and Lemma 2(1-3) that such a path exists and that } s'_{j_2} = \text{sh } s_{i+j_1+j_2}. \text{ Since } j_2 \text{ is the highest such index Lemma 1 implies that the condition of line 11 of MP-State does not hold when } s'_{j_2} \text{ is visited. So there must be a state } s^{j_2} \text{ reachable via a path } \sigma^{j_2} \text{ in } M_Q \text{ such that } s^{j_2} = \text{sh } s'_{j_2} \text{ and } s^{j_2} \notin \text{reached} \text{ when visited in MP-State. From Lemma 2(4), we know that } s^{j_2} = \text{sh } s_{i+j_1+j_2}. \text{ Continue the construction. Let } 0 < j_1, j_2, \ldots, j_k \text{ be
natural numbers such that $i + j_1 + j_2 + \ldots + j_k = n$. The construction ends because $1 \leq k \leq n - i$. Inductively, we have that $s_n =_{sh} s_{j_k}^{j_k-1}$. Since $s_{j_k}^{j_k-1}$ is reachable in $MQ$ and from Lemma 2(1), we have a contradiction.

**Theorem 3 (Debugging).** See in Section 3.3.

**Proof.** The proof is indirect. Let $M$ and $MQ$ be as in Section 2. Assume that $s$ is reachable in $M$ and $s'$ is not reachable in $MQ$. The construction in the proof of Theorem 2 (Figure 2) proves the existence of a reachable state $s_{j_k}^{j_k-1}$ in $MQ$ such that $s_{j_k}^{j_k-1} =_{sh} s$, a contradiction. Now, assume that MP-State cannot return a protocol path leading to $s'$. Since MP-State is DFS, the sequence of states on the stack in $s'$ is a path $\sigma$ to $s'$ from the initial state. This is true if the condition of line C1 of Algorithm 1 is always true. Otherwise, a state in $\sigma$ can only be missing if the transition $t$ executed in the state is not a protocol transition (cf. line C1), a contradiction.