Efficient Verification of Distributed Protocols Using Stateful Model Checking

Habib Saissi, Péter Bokor, Can Arda Muftuoglu and Neeraj Suri
TU Darmstadt, Germany
Email: {saissi, pbokor, arda, suri}@cs.tu-darmstadt.de

Marco Serafini
Qatar Computing Research Institute, Qatar
Email: mserafini@qf.org.qa

***Abstract*—This paper presents efficient model checking of distributed software. Key to the achieved efficiency is a novel stateful model checking strategy that is based on the *decomposition* of states into a *relevant* and an *auxiliary* part. We formally show this strategy to be sound, complete, and terminating for general finite-state systems.**

As a case study, we implement the proposed strategy within Basset/MP-Basset, a model checker for message-passing Java programs. Our evaluation with actual deployed fault-tolerant message-passing protocols shows that the proposed stateful optimization is able to reduce model checking time and memory by up to 69% compared to the naive stateful search, and 39% compared to partial-order reduction.

I. Introduction

Software model checking (MC) [14], [16] is a practical branch of verification for checking the actual implementation of the system. The wide usability comes at the price of low scalability as the model checking of even simple single-process programs can take several hours (or go off-scale) using state-of-the-art techniques [23].

Verification complexity gets even worse for concurrent programs that run on loosely coupled processes. Our focus is on distributed protocols for various mission-critical (fault-tolerant) applications where rigorous verification is desired. Example applications include atomic broadcast [21], storage [12], diagnosis [29], etc. Although the verification of fault-tolerant distributed systems is known to be a hard problem due to concurrency and faults, MC has proven to be useful for debugging and verifying small instances of deployed protocols; recent approaches include MaceMC [22], CrystalBall [30], Modist [31], [18], Basset [25] and its extensions/optimizations [4], [5], [28].

In MC, the possible executions of a system are modeled in terms of a state graph, where states (i.e., nodes) can be thought of as snapshots of the entire system (e.g., state of the servers, clients, communication channels) and transitions (i.e., edges) model any event that may alter the system's state (e.g., lines of code, function blocks). For MC to be scalable, the size of the graph must be feasible to manage, a challenge that is often referred to as the *state explosion* problem. An efficient and simple approach is *stateful depth-first search* [10], where the state graph is abstracted by 1) a sequence of states (called stack) that corresponds to the last run of the system, and 2) a set of states that have been explored during the model checking (called visited states).

In this paper, we propose a general and sound approach to reduce the size of both the stack and the visited states for improved scalability of MC. Key to the proposed reduction is the concept of *decomposition* that we observe to be present in the implementation of real systems. For example, implementations of distributed systems are typically decomposed into different aspects or execution modes (i.e., runnable configurations of the system under verification) of the system such as synchronization, GUI, automatic execution, or logging. Despite the richness of implementations, the specifications subject to model checking very often consider only a subset of all these aspects. Roughly speaking, our reduction approach consists in utilizing decomposition so that only selected aspects are model checked against the specification without having to modify the implementation.

a) Our Theoretical Contributions: We propose a formal framework that characterizes decomposition by distinguishing between *relevant* and *auxiliary* state information. The decomposition is always with respect to a subset of all transitions of the system corresponding to the execution mode of interest. We show a use of this characterization for more scalable stateful depth-search, called *decomposition-based stateful search*, and prove the soundness of the proposed approach. The input of the framework (beside the specification of the system) is a sound decomposition. Although showing the soundness of a decomposition can be as hard as model checking itself, we argue that this can be done using suitable static analysis and we justify this claim by showing an implementation for general distributed systems implemented in Java.

b) Our Prototype Implementation: We implement the proposed decomposition-based stateful model checking within Basset [25], [4], an explicit-state model checker for general message-passing Java programs. We then apply our decomposition framework to optimize the verification of distributed message-passing algorithm. In this decomposition, the auxiliary part of the state stores the latest messages delivered by a process. This is utilized only for debugging, that is, to analyze runs where the desired properties of the system are violated This execution mode is irrelevant, say, for the fault-tolerance aspects of the system and the corresponding state information can be safely decomposed as auxiliary.

c) Our Evaluation: We use our prototype implementation to evaluate the proposed decomposition-based stateful search with various fault-tolerant message-passing protocols such as Paxos consensus [24], Zookeeper atomic broadcast [21], and distributed storage [1]. The decomposition-based stateful optimization improves on the naive stateful search both in terms of search time and memory by up to 69%. We also compare decomposition-based stateful search with partial-order reduction [13], an optimization known to be efficient for

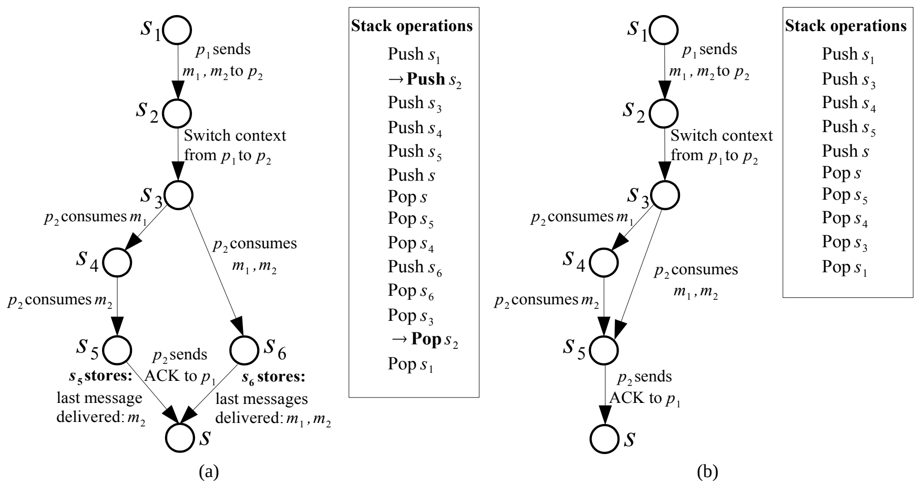

Fig. 1: (a) Naive depth-first search (DFS) and (b) decomposition-based stateful search.

fault-tolerant message-passing protocols [4], [5]. Our experiments show that the two optimizations, when used together, result in enhanced reductions achieving an improvement of 39% compared to settings with partial-order reduction only.

The paper is structured as follows. After a motivating example, we present in section III our decomposition framework and prove the correctness of the DBSS algorithm. In section IV and V, we discuss the implementation of DBSS within MP-Basset and evaluate our experimental results. Section VI discusses other related work, and we conclude with section VII.

II. Motivating Example

We give the intuition of the proposed reduction 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. After receiving m_1 and m_2, p_2 sends an acknowledgement message to p_1.

Figure 1(a) shows the state graph of the this example system as explored by a naive DFS and the corresponding operations of the search stack. Note that s_5 and s_6 are different states because they store different messages histories.

a) Decomposition: Suppose that the message history information is not subject to the verification. As a result, this part of the state is labeled as auxiliary. This decomposition is sound with respect to the transitions shown in Figure 1 because these transitions only depend on the non-auxiliary (i.e., relevant) part of the state. Note that although the system may contain additional transitions, in Figure 1 only those transitions are depicted that are relevant in target execution mode.

b) Selective Hashing: We propose a reduction framework for more scalable stateful depth-first search by making use of the decomposition of a system. Firstly, we introduce *selective hashing*, which modifies the naive search in that it only stores the relevant state in the set of visited states. 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 they only differ with respect to their message histories. The gain of selective hashing is that it directly reduces the size of the state graph that is explored by the model checker.

c) Selective Push-on-Stack: Secondly, we introduce *selective push-on-stack*, which is based on the observation that the transition system may have *single enabled* transitions, i.e., transitions that are exclusively enabled in a state. Since single enabled transitions are non-concurrent with any other transitions, states where these transitions are executed do not have to be used for backtracking, although they have to be remembered as visited states. Therefore, states with single enabled transitions do not have to be pushed onto the search stack. Consider the single enabled transition t from s_2 to s_3 in Figure 1[1]. 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. The application of selective push-on-stack to our example single-enabled transition leads us to the search stack in Figure 1(b), where s_2 is not involved in any stack operation. The information of visiting s_2 is stored in a different stack, which is returned when a counterexample is found. Note that selective push-on-stack visits the same states as the naive search but in shorter time with fewer stack operations.

III. General Reduction Framework

Our general model for decomposition is presented in Section III-A. The proposed verification approach and its properties are explained in Section III-B and Section III-C, respectively.

A. System Model

We adopt a general and abstract model of programs [2], [3]. The program maintains a global state and can execute transitions (e.g., line of codes) to reach other states. Formally, a

[1]Note that t is not the only single enabled transition in the example.

program is represented as a transition system $TS = (S, S_0, T)$ where:

- S is a finite set of possible states of the program.
- $S_0 \subseteq S$ is a set of initial states. For simplicity, we assume that there is a single initial state $s_I \in S_0$.
- $T = \{t \mid t \subseteq S \times S\}$ is a finite set of transitions.

A transition $t \in T$ is *enabled in* state $s \in S$ and we write $t \in enabled(s)$, if there is $s' \in S$ such that $(s, s') \in t$. Consequently, we define $enabled(s)$ as a set of transitions enabled in s. To simplify the discussion, we assume that transitions are deterministic, i.e given a state $s \in S$ and a transition $t \in T$ enabled in s, there is a single state $s' \in S$ such that $(s, s') \in t$. A *path* of the model is defined as a finite sequence $s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 ... \xrightarrow{t_{n-1}} s_n$, where $s_1, ..., s_n \in S$, $t_1, ..., t_{n-1} \in T$ and for all $1 \leq i < n$ it holds that $(s_i, s_{i+1}) \in t_i$. For convenience, we write $s_1 \xrightarrow{t_1 ... t_{n-1}} s_n$. A state s is called *reachable* if there exists a path $s_I \xrightarrow{t_1 ... t_n} s$.

a) Decomposition: Let $TS = (S, S_0, T)$ be a transition system, where T may be a subset of all transitions of the system corresponding to the execution mode under verification. We decompose a state $s \in S$ into two states: A "relevant" part $s_{rel} \in S_{rel}$ and an "auxiliary" part $s_{aux} \in S_{aux}$ such that $s = (s_{rel}, s_{aux})$. Whether a state fragment can be marked as relevant depends on the system and the property being checked. This should be done by a domain expert.

To formalize our approach, we introduce the function $h : S \rightarrow S_{rel}$, to extract the relevant part of states such that for a state $s = (s_{rel}, s_{aux}) \in S$ we have $h(s) = s_{rel}$. Given two states $s, s' \in S$, a transition $t \in T$ with $(s, s') \in t$ is said to be *single enabled* in s iff t is the only transition enabled in s. Intuitively, a single enabled transition is non-concurrent with any other transition.

Intuitively, we require that, given two states s and $s' \in S$ with the same relevant part, a transition t is enabled in s only if it is enabled in s'. Moreover, the execution of t in s and s' results in two states s_1 and s'_1 with the same relevant part. Formally, we characterize a sound decomposition as:

Definition 1 (Decomposition)**.** *Given a transition system $TS = (S, S_0, T)$ and a set S_{rel} of states, we say that TS can be decomposed along S_{rel} if:*

- (State decomposition) $S \subseteq S_{rel} \times S_{aux}$.
- (Transition decomposition (1)) $\forall s, s' \in S, t \in T : h(s) = h(s') \Rightarrow (t \in enabled(s) \Leftrightarrow t \in enabled(s'))$.
- (Transition decomposition (2)) $\forall s, s', s_1 \in S, t \in T : (h(s) = h(s') \wedge s \xrightarrow{t} s_1) \Rightarrow (\forall s'_1 \in S : s' \xrightarrow{t} s'_1 \Rightarrow h(s_1) = h(s'_1))$.

B. Decomposition-based Stateful MC

a) Preserved Specifications: We assume that the specification of the system is given in form of state properties. As state properties refer to the "global" state of the system, they constitute a key class of properties of various distributed systems [30], [31], [28], [18]. Formally, given a transition system $TS = (S, S_0, T)$, we define a *state property* f as $S \rightarrow \{true, false\}$. The state property *holds* for a transition system

```
function DBSS(TS, f)
1   Stack stack ← ∅
2   Set reached ← ∅
3   Stack CE-stack ← ∅
4   State s ← s_I
5   stack.push(s)
6   CE-stack.push(h(s))
7   while stack ≠ ∅ do
8       while enabled(s) ≠ ∅ do
9           Transition t ← next(enabled(s))
10          enabled(s) ← enabled(s) \ {t}
11          State $s' \xleftarrow{t} s$
12          //selective hashing
C1          if h(s') ∉ reached then
C2              reached ← reached ∪ {h(s')}
13              CE-stack.push(h(s'))
14              //selective push-on-stack
C3              if next(enabled(s')) is not single enabled s' then
15                  stack.push(s')
16              s ← s'
17          if ¬f(s) then
18              return s, CE-stack
19      s ← stack.pop()
20      while h(s) ≠ CE-stack.peek() do
21          CE-stack.pop()
22  return true
```

Algorithm 1: Decomposition-based stateful search (DBSS) algorithm for transition system TS and state property f.

if it returns true for every reachable state. Let $TS = (S, S_0, T)$ be a transition system that can be decomposed along a set of states S_{rel} and $f : S \cup S_{rel} \rightarrow \{true, false\}$ a state property. We say that f is *decomposed* if for all reachable $s \in S$ it holds that $f(s) = f(h(s))$. Intuitively, f depends solely on the relevant part of a state.

b) The Algorithm: Algorithm 1 shows the pseudo-code of the proposed decomposition-based stateful search (DBSS). The call next($enabled(s)$) non-deterministically returns one transition from $enabled(s)$. The calls *pop* and *push* respectively correspond to the usual stack operations of removing and adding an element to the stack. Calling *peek* returns the topmost element of the stack without removing it. In principle, the DBSS algorithm modifies the naive depth-first search (DFS) algorithm. The algorithm uses a $stack$ to remember the explored paths. The stack is also used for backtracking in case of branching. To avoid redundancy by visiting a state more than once, the set $reached$ stores the visited states. The use of $reached$ constitutes the fundamental optimization of the stateful search. The outer loop at line 7 assures that every visited state is checked for branching. The loop at line 8 guarantees that every enabled transition is explored.

The first modification to DFS is *selective hashing* in lines C1 and C2. Instead of remembering in $reached$ a new visited state s, the algorithm remembers only the relevant state part (C2). Two states with the same relevant parts are considered to be equivalent (C1). Note that, in contrast to the $reached$ set, the entire state is pushed on the stack. This corresponds to allowing the verification of unmodified implementations, where transitions can be meaningfully executed only in fullfledged states containing both the relevant and auxiliary parts. The second modification is *selective push-on-stack* in line C3. Instead of pushing every newly visited state s' onto the search stack, s' is only pushed if the transition enabled in s' is *not* a single enabled transition. Since the single enabled transition is the only enabled transition in s', no branches that possibly lead to new reachable states are missed. We add another stack to the algorithm, $CE\text{-}stack$, to keep track of the relevant part of the states that compose the explored path. $CE\text{-}stack$ also serves the purpose of keeping track of all executed transitions including those skipped from backtracking because of selective push-on-stack. If a bug is found, that is the condition in line 17 holds, we return the reached state s and $CE\text{-}stack$ as a counterexample path leading to the state violating the property. Otherwise, the state property holds and $true$ is returned.

c) Liveness Model Checking: Using the notations from the definition above, as DBSS explores a subset of TS' which behaves like TS, the algorithm can be modified to *generate* TS. Doing so, standard algorithms can be used to check liveness properties (e.g. written in temporal logics [10]) that cannot otherwise be expressed through state properties.

d) Analysis: By exploiting decomposition, the DBSS algorithm explores only a "relevant state graph". In other words, given a transition system TS' that can be decomposed along S, the transition system explored by DBSS simulates another transition system $TS = (S, S_0, T)$ which is subsumed by TS'. Intuitively, given some state property f, $DBSS(TS', f)$ explores a transition system that behaves like TS. As TS is subsumed by TS' (and is thus a smaller transition system), DBSS improves time and memory efficiency over DFS of TS'. This analytic claim will be substantiated by our experiments in Section V. Formally, we can define a subsumption relation between the two transition systems TS' and TS:

Definition 2 (Subsumption)**.** *Given two transition systems $TS = (S, S_0, T)$ and $TS' = (S', S'_0, T')$, we say that TS' subsumes TS and write $TS \subseteq TS'$ if:*

- *TS' can be decomposed along S and*
- $\forall s, s_1 \in S : (\exists t \in T : s \xrightarrow{t} s_1) \Leftrightarrow (\exists t' \in T', \exists s', s'_1 \in S' : s' \xrightarrow{t'} s'_1 \wedge h(s') = s \wedge h(s'_1) = s_1)$.

Note that there is no need of proving that the above subsumption relation indeed applies for TS' and the transition system explored by $DBSS(TS', f)$. The above discussion has been added to better highlight the source of reduction of DBSS.

C. Correctness of DBSS

We now show that the DBSS algorithm can be used for the verification of decomposed state properties without missing bugs and also without falsely concluding the truth of the property. Formally, we prove that the algorithm is sound, complete and terminating. The proofs of the theorems appear in the Appendix.

Theorem 1 (Soundness)**.** *Given a transition system TS and a decomposed property f, if $DBSS(TS, f)$ returns $true$, then f holds for TS.*

The algorithm returning $true$ is a guarantee that the verified program satisfies the property f.

Theorem 2 (Completeness)**.** *Given two transition systems TS' and TS such that $TS \subseteq TS'$ and a decomposed property f, if $DBSS(TS', f)$ returns s and $CE\text{-}stack$, then:*

- *$\neg f(s)$ and s is reachable in TS' and*
- *$CE\text{-}stack$ contains a path from $h(s_I)$ to $h(s)$ in TS where s_I is the initial state of TS'.*

In case a bug violating a property f is found, DBSS returns a state where f does not hold, and a path in the subsumed transition system leading to it. In practice, such a path is sufficient for debugging as the subsumed system contains all relevant state information.

Theorem 3 (Termination)**.** *$DBSS(TS, f)$ terminates for any transition system TS and state property f.*

The termination of the program follows directly the assumption that the transition system is finite-state and the algorithm is based on DFS.

IV. Implementing DBSS in JPF/MP-Basset

In this section, we present a general application and implementation of the conceptual reduction framework described in Section III. The following application instantiates, implements, and evaluate the decomposition-based reduction framework for general message-passing systems written in Java. A direct implication of our results is the enhanced scalability of model checking Java-based implementations of message-passing systems. We also discuss how our implementation can be used for symmetry reduction of replication-based (fault-tolerant) message-passing protocols.

We implement the proposed decomposition-based stateful search (DBSS) within the MP-Basset model checker for message-passing systems [4]. [2] The source of our implementation of DBSS/MP-Basset can be downloaded under [33]. In the core of MP-Basset, the model checker Java Pathfinder [32] (JPF) implements depth- and breadth first search of multi-threaded Java programs. MP-Basset builds upon JPF's architecture to enable writing and model checking message-passing Java programs. In essence, JPF consists of the *core* search engine and a *model*. Intuitively, the core is responsible for the search, whereas the model constitutes the Java program under verification. The model, in this case MP-Basset, runs in a separate Virtual Machine implemented by JPF. JPF itself is implemented in Java and it runs within the Java Virtual Machine of the host system. Our decomposition based approach extends JPF's core with selective hashing and push-on-stack. As the reduction is based on the decomposition of the system, this information is obtained from the JPF Virtual Machine, which contains all system-specific information. Communication between core and model is done via JPF's Model Java Interface (MJI). First in Section IV-A we will discuss how our decomposition model applies to the MPBasset case study.

[2] Our instrumentation would analogously apply for Basset [25], the precursor of MP-Basset.

Section IV-B (respectively, in Section IV-C) explains how the *hash* function (and *auxiliary* predicate) is implemented within the architecture of Basset/MP-Basset and JPF.

A. Decomposition

From Definition 2, $TS = (S, S_0, T)$ corresponds to the message-passing program as defined in Basset/MP-Basset's input language (a Java library for message-passing), whereas $TS' = (S \times S_{aux}, S'_0, T')$ is determined by the program executed by the JPF virtual machine. The argument that it is a sound decomposition implicitly follows from the (sound) implementation of the model checkers Basset [25] and MP-Basset [4]. Every Java method specified by the message-passing program can always be executed if the method's guard (a concept implemented by Basset/MP-Basset) is enabled. In Basset/MP-Basset, S_{aux} contains the set of messages processed by the last transition, as explained in the example in Section II. Note that S_{aux} might contain any data as long as the decomposition property can be shown. We (manually) verify that state properties are decomposed by checking if the property (Java assertion) only involves variables of the message-passing program and other variables for context switching required to satisfy transition decomposition.

B. Selective Hashing

The JPF's stateful optimization is implemented by serializing the state of the JPF Virtual Machine; the outcome of serialization is stored as a reached state. We explain this mechanism using code excerpts of JPF core (Figure 2). The serialization uses two data structures, a *(reference) queue* (line 19) and a *buffer* (line 20). The queue is an array containing references of objects in the JPF Virtual Machine. The buffer is an array of integers where each element holds the value of a primitive type. Initially, the queue contains references of the topmost classes of program. The serialization of the system state is done by calling the process method (line 3). The references in the queue are processed one-by-one (line 5) where the reference itself (line 29) and the content of the referenced object is added to the buffer (lines 37-55). Every object is a composition of primitive (e.g., int) and non-primitive types (e.g., HashMap). If it is a primitive type, its value is added to the buffer (line 53), otherwise the reference is added to the end of the queue for further processing (line 50). The serialization of the state terminates if every reference in the queue has been processed (line 4). Finally, JPF uses a hashing function (not depicted) to compute the hash value of the buffer.

a) Serialization Example: Consider the simple actor program in Figure 3 written in Basset's Java library. Actors correspond to processes in our general system model. The driver class is used to create the initial actors. In this example, two actors of class FooActor are created. For simplicity, no message is sent in this example. Consider the state of the system after executing the main function of the driver class. In this state, the process method of the serializer is called with refQueue=$[ref_{a1}, ref_{a2}]$ where ref_{a1} and ref_{a2} denote the references of the two actors. As a result of the

```
1   public class ReferenceQueue {
2       //...
3       public void process(ElementInfoProcessor proc) {
4           for (Entry e = markHead; e != null; ) {
5               proc.processElementInfo(e.refEi);
6               //...
7           }
8       }
9       public void processActorQueue(MPSerializer proc) {
10          for (Entry e = markHead; e != null; ) {
11              //...
12              processNamedFields(ei, ci, fields);
13          }
14      }
15  }
16  public class MPSerializer extends FilteringSerializer {
17  //FilteringSerializer implements ElementInfoProcessor {
18      //...
19      protected ReferenceQueue refQueue;
20      protected IntVector buf = new IntVector(4096);
21      public void processElementInfo(ElementInfo ei) {
22          Fields fields = ei.getFields();
23          ClassInfo ci = ei.getClassInfo();
24          //SELECTIVE HASHING
25          if (StringSetMatcher.isMatch(
26                  ci.getName(),
27                  includeClasses,
28                  excludeClasses)) {
29              buf.add(ci.getUniqueId());
30              actorQueue = new ReferenceQueue();
31              actorQueue.add(ei);
32              actorQueue.processActorQueue(this);
33              //...
34              //processNamedFields(ei, ci, fields);
35          }
36      }
37      protected void processNamedFields(
38              ElementInfo ei,
39              ClassInfo ci,
40              Fields fields){
41          FinalBitSet refs = getInstanceRefMask(ci);
42          //...
43          int[] values = fields.asFieldSlots();
44          for (int i = 0; i < values.length; i++) {
45              //...
46              int v = values[i];
47              if (refs.get(i)) {
48                      //...
49                      actorQueue.add(ei);
50                      //refQueue.add(ei);
51                      //...
52              } else
53                  buf.add(v);
54          }
55      }
56      //..
57  }
```

Fig. 2: Selective hashing via modified serializer in JPF. Text in bold shows our changes.

```
public class FooActor extends Actor{
    int id;
    FooClass fooClass;
    public FooActor(int id){
        this.id=id;
        fooClass=new FooClass(id);
    }
    class FooClass{
        int foo;
        public FooClass(int id){
            foo=id+10;
        }
    }
}
public class Driver extends TestDriver {
    public static void main(String[] args) {
        //...
        ActorName a1, a2;
        a1=PlatformUtil.createActor(FooActor.class,1);
        a2=PlatformUtil.createActor(FooActor.class,2);
        //...
    }
}
```

Fig. 3: Message-passing system with two actors.

serialization (before calling the hash function), the buffer will contain $[1, 2, 11, 12]$ (whereas refQueue will contain $[ref_{a1}, ref_{a2}, ref_{foo1}, ref_{foo2}]$ where ref_{foo1} and ref_{foo2} denote references of FooClass in a1 and a2, respectively).

b) Our Design of Selective Hashing: The heart of our implementation of selective hashing is an additional condition (lines 25-28) applied during serialization. This condition enforces the rule that a reference is only processed if it is selected for inclusion.[3] In our current setting (not depicted) the set of excluded classes is empty whereas the set of included classes consists of classes extending Actor and the class (called Cloud) holding the set of pending messages. A new reference queue is created for each such class and it is processed recursively (lines 30-32) similar to the original serializer. We remark that this mechanism cannot be implemented using the standard JPF API. Although include/exclude classes are supported by JPF, they are used to include/exclude *every* reference in the queue. Therefore, JPF's serializer makes no difference between an object reference within and outside an actor (or Cloud) class.

c) Our Structured Serialization: We now explain another benefit of our solution which relates to symmetry reduction [26], a promising optimization of model checking of distributed systems. Intuitively, most distributed systems are symmetric with respect to replicated processes, where replication may serve different goals such as fault-tolerance or enhanced performance. It has been shown that symmetry reduction can be extremely efficient in various practical applications of distributed systems [26], [6]. Unfortunately, symmetry reduction has not yet established itself as an efficient *software* verification technique. In fact, to the best of our knowledge, the only attempt to implement general purpose symmetry reduction for software verification was the SymmSpin extension of the Spin model checker [7], an implementation that is no longer maintained [37].

Our modified JPF serializer for selective hashing paves the way for implementing symmetry reduction for message-passing systems à la Basset/MP-Basset. We aim at process-based symmetries that arise from the free permutation of local process states.[4] JPF serializes the current state irrespective of the structure of the state. Therefore, we call JPF's serialization *unstructured*. The unstructured approach is in contrast to ours where actors (processes) and pending messages are serialized in isolation and appended to the final result. We call this *structured* serialization, which we apply for selective hashing. We observe that structured serialization can also be used for implementing symmetry reduction. The idea is that the output of structured serialization can be used to *canonicalize* (or normalize [20], [7]) the state, which corresponds to mapping each state into a unique state by permuting the local states of processes. Canonicalization is the common way to implement symmetry reduction [26] because it allows that only canonicalized states (and their successor states) need to be explored. Note that canonicalization is impossible using the unstructured serializer because the local state of a process is unknown to the serializer, as shown in the following example.

```
1   public class JVM {
2       //...
3       public boolean forward () {
4           //...
5           //SELECTIVE PUSH-ON-STACK
6           if (isBranchState()){
7               backtracker.pushSystemState();
8               updatePath();
9           }
10      }
11      private boolean isBranchState() {
12          if (getChoiceGenerator() != null &&
13          !(getChoiceGenerator() instanceof
14                          ThreadChoiceGenerator))
15              return true;
16          return false;
17      }
18      //..
19  }
```

Fig. 4: Selective push-on-stack with JPF's choice generators. Text in bold shows our changes.

d) Symmetry Example: Assume that the system in Figure 3 is symmetric with respect to the IDs of the actors. This means that another execution of the system where actor a1 and a2 are given IDs, respectively, 2 and 1 (and not 1 and 2 as in Figure 3) is indistinguishable by the property of interest (i.e., the property holds or fails in both executions). The result of unstructured serialization in these two execution examples would be $[1, 2, 11, 12]$ and $[2, 1, 12, 11]$, respectively. Let the first state be the canonicalized state. After serialization, it is impossible to find out that $[2, 1, 12, 11]$ can be canonicalized into $[1, 2, 11, 12]$ because the information that $\langle 1, 11\rangle$ and $\langle 2, 12\rangle$ constitute the local state of actor a1 and a2, respectively, is dismissed throughout serialization. The structured serializer, on the other hand, outputs $[1, 11, 2, 12]$ and $[2, 12, 1, 11]$ for the respective states and it is aware of the information of how states are structured along processes, i.e., $[\langle 1, 11\rangle, \langle 2, 12\rangle]$ and $[\langle 2, 12\rangle, \langle 1, 11\rangle]$. Therefore, our structured serializer makes the canonicalization of the states for process symmetries possible.

C. Selective Push-on-Stack

Our implementation of selective push-on-stack is based on JPF's mechanism for the systematic exploration of branching execution. Intuitively, the execution of the Java program can branch if, given a state of the program, methods of different threads can be executed concurrently. In JPF, *choice generators* are used to associate such methods with the state. 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. Every time a new state is visited by JPF, the forward method is called (see Figure 4) and the state is pushed onto the search stack (line 8). According to the proposed selective push-on-stack strategy, this push operation is done conditionally (line 6). The condition in our implementation is specific to Basset/MP-Basset where we know that a choice generator of type ThreadChoiceGenerator corresponds to single enabled transitions. This choice generator is responsible for switching context between actors (cf. example in Section II) and it never specifies branching the

[3]Our implementation utilizes Basset's StringSetMatcher method.

[4]There are efficient techniques to detect such symmetries [20], [6].

execution (i.e., the set of enabled transition in the current state consists exactly of one transition). Therefore, selective push-on-stack can be implemented simply by checking if the current choice generator is a ThreadChoiceGenerator (lines 12-16).

V. Evaluation with Fault-Tolerant Protocols

In this section, we evaluate DBSS with representative fault-tolerant message-passing protocols. We measure the gain of DBSS compared to the 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 DBSS.

a) Target Protocols and Properties: Our evaluation is based on the following protocols: Paxos consensus [24], a regular register protocol in the style of ABD [1] , and Zab atomic broadcast [21]. 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 in [11]. 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 [31], or the Zab protocol is part of Yahoo's Zookeeper open-source library used in different real deployments [35]. As the implementation of the protocols is not available to us[5], we use our prototype Java implementation in each case. In our evaluation, we use different settings of the above protocols (see description in Table I). In addition to the protocols and their specified properties, we *inject* faults in each protocol and/or its properties to evaluate the debugging feature of DBSS. Note that we injected subtle faults, e.g., liveness of Zab [21], [28] or safety of Faulty-Paxos2 [24], to challenge the model checker.

b) Experimental Setup/Reduction Types: We run our experiments in a Deterlab testbed [34] 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. For comparability, DBSS uses the same scheduling of the transitions as MP-Basset's naive search.

First, we evaluate stateful against stateless model checking. Note that the search always terminates for our acyclic examples. We then evaluate DBSS without selective push-on-stack[6]. DBSS without selective push-on-stack experiments show the added benefit of push-on-stack technique exclusively. We only expect time reduction for these experiments, as this reduction type cannot achieve memory reduction. The third reduction type measures the performance of DBBS, as explained in Section III-B. Finally, we apply stateful partial-order reduction (POR) alone (base case), then in combination with DBSS.[7] We use POR wherever it is applicable. For example, MP-Basset's implementation of POR does not apply for Zab (see more details later about the assumptions made by POR and DBSS).

[5]Although Zookeeper is an open-source project, the code of Zab cannot be extracted as a stand-alone protocol.

[6]In this reduction type, a new state s' is always pushed onto the stack even if the condition of line C3 in Algorithm 1 does not hold.

[7]MP-Basset implements different POR algorithms; we apply static POR for our experiments as it is more efficient than dynamic POR for the considered class of protocols [5].

We use the following notations to display our results in Table I. 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 fault-injected 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 (192 hours). For exhaustive searches that end with timeout, the value in the states column indicates the number of visited states at the time when the search stops.

c) Reduction Results: The results of our experiments are shown in Table I. 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 – only the reduction of stateful over stateless search is depicted in Table I. In some cases (e.g. Paxos(6)), the stateful search terminates where the stateless search is infeasible (given our time-out). In other cases, stateful model checking reduces the search time by up to 94% compared to stateless model checking.
- **DBSS improves efficiency.** DBSS 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 9% (see Register (5) experiment) – fault-injected cases with DBSS without selective push-on-stack are not displayed as they follow the same reduction trend as the exhaustive experiments.
- **DBSS efficient with POR.** When DBSS is used with POR, DBSS reduces model checking time and memory by up to 39%, compared to the experiments with only POR.

d) Assumptions by POR/DBSS: The reduction achieved by POR can be significantly more than by DBSS. For example, POR reduces model checking time by 98% for Paxos (6), whereas DBSS achieves a reduction of 54%. This is only true given the assumptions made by POR that the execution of certain transitions is commutative [13]. The soundness of POR can only be guaranteed if this assumption is verified. DBSS, in contrast to POR, makes no assumptions about the commutativity of transitions. For example, the simple static analysis in MP-Basset's POR implementation [5] is not applicable for Zab to verify the assumptions required by POR. DBSS is still applicable in this case and it achieves a time reduction of up to 69%.

e) Scalability: We observe that the reduction achieved by DBSS changes with the number of processes. In fact, the time reduction of "DBSS + POR" is 18%, 33%, and 28%,

Protocol (# of processes)	Description	Result	Property	Reduction type	States	Time	Time reduction*
Paxos (6)	2 proposers each issuing $\leq$ 1 proposal, 3 acceptors, and 1 learner	OK	Agreement	MP-Basset stateful	13,044,613	22h19m	N/A
				DBSS without SPoS	5,606,047	11h01m	51% (over SF)
				DBSS	**5,606,047**	**10h22m**	**54% (over SF)**
				POR	191,081	23m43s	98% (over SF)
				DBSS + POR	**117,369**	**14m22s**	**39% (over POR)**
Faulty-Paxos (6)	Paxos(6) setting + One acceptor accepts all proposals (instead of those without prohibiting promise)	CE	Agreement	MP-Basset stateful	96,802	10m3s	94% (SL)
				DBSS	**70,543**	**8m20s**	**17% (SF)**
				POR	3,050	36s	94% (SF)
				DBSS + POR	**2,786**	**31s**	**14% (POR)**
Faulty-Paxos2 (7)	Paxos(6) setting with 3 proposers** + One acceptor remembers last accepted proposal (instead of highest numbered accepted proposal)	CE	Agreement	MP-Basset stateful	>71,914,839	>192h	N/A
				DBSS	**>66,651,310**	**>192h**	**N/A**
				POR	129,533	21m18s	N/A
				DBSS + POR	**124,976**	**19m29s**	**9% (POR)**
Register (5)	3 base objects, 1 reader (single writer)	OK	Regularity	MP-Basset stateful	89,041	7m31s	14% (SL)
				DBSS without SPoS	59,306	6m20s	16% (SF)
				DBSS	**59,306**	**5m39s**	**25% (SF)**
				POR	10,896	1m5s	86% (SF)
				DBSS + POR	**8,590**	**53s**	**18% (POR)**
Register (5)	Register(5) setting + Read finishing after concurrent write to return written value (instead value of last preceding write)	CE	Wrong regularity	MP-Basset stateful	4,965	34s	66% (SL)
				DBSS	**3,376**	**27s**	**21% (SF)**
				POR	1,936	24s	29% (SF)
				DBSS + POR	**1,483**	**20s**	**17% (POR)**
Register (6)	Register (5) setting with 4 base objects	OK	Regularity	MP-Basset stateful	2,269,797	5h22m	90% (SL)
				DBSS without SPoS	1,475,845	4h31m	16% (SF)
				DBSS	**1,475,845**	**4h8m**	**23% (SF)**
				POR	96,641	11m1s	97% (SF)
				DBSS + POR	**57,187**	**7m26s**	**33% (POR)**
Register (6)	Register (5) setting with 4 base objects	CE	Wrong regularity	MP-Basset stateful	94,348	10m49s	85% (SL)
				DBSS	**56,222**	**8m14s**	**24% (SF)**
				POR	4,642	51s	90% (SF)
				DBSS + POR	**4,642**	**48s**	**6% (POR)**
Register (7)	Register (6) setting with 5 base objects	OK	Regularity	MP-Basset stateful	51,465,807	>192h	N/A
				DBSS	**35,692,316**	**>192h**	**N/A**
				POR	2,986,657	8h21m	N/A
				DBSS + POR	**1,656,212**	**6h3m**	**28% (POR)**
Zab (6)	3 leaders, 3 followers	CE***	Liveness****	MP-Basset stateful	4,580	54s	86% (SL)
				DBSS	**1,876**	**26s**	**38% (SF)**
				POR	N/A	N/A	N/A
Zab (7)	4 leaders, 3 followers	CE	Liveness****	MP-Basset stateful	8,198	2m4s	80% (SL)
				DBSS	**3,132**	**38s**	**69% (SF)**
				POR	N/A	N/A	N/A

TABLE I: Evaluation results of DBSS 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. **At least three proposals needed to violate agreement for Faulty-Paxos2. ***Exhaustive search infeasible for smallest meaningful Zab instance. ****Zab is not live in general [21]; liveness is encoded as state property [28].

for the register with 5, 6, and 7 processes, respectively. One reason of this trend can be in the majority voting mechanism that the 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 DBSS 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.

VI. Related Work

a) Model Checkers.: Mainstream software model checkers include explicit-state checkers for C programs such as Verisoft [14] and Spin [19], for Java programs [32], symbolic execution engines such as DART for C [16] or KLEE for low-level (byte)code [8], and dedicated solutions for message-passing systems such as Modist [31] or Mace [22]. 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 and it can also be used to decrease the number of variables needed for a symbolic encoding of the state.

Some model checkers (such as Mace [22]) offer the user an interface to exclude certain state information from the representation of the state. As a result, similar to selective hashing, the excluded state information is not considered by stateful model checking. It is, however, left to the user to guarantee the correctness of model checking. Our notion of decomposition formalizes a sufficient condition of correctness, which can be applied by users of these model checkers.

b) Reductions: Broadly-studied and intuitive reductions are partial-order (POR) [13] and symmetry reductions (SR) [26]. Figure 1 demonstrates that DBSS 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 [26]) that would preserve the transition relation: 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 "asymmetric" because of s_4.

To the best of our knowledge, all known reduction approaches that work with depth-first search, such as SR, POR, or dynamic interface reduction (DIR) [18], can be directly combined with DBSS. Reductions of stateless model checking such as symmetric transitions [15] or dynamic partial-order reduction [17] would only benefit from selective push-on-stack. The reduction achieved by DBSS is based on the assumption of a sound decomposition. Other reductions are also based on (other) assumptions: 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.

In a recent brief announcement [27], we sketched a preliminary outline of the idea of DBSS.

VII. Conclusion and Future Work

We have proposed decomposition-based stateful search (DBSS) as an improvement of explicit-state software model-checking. Given a sound decomposition we showed the correctness of DBSS. Also, we have built a proof-of-concept implementation based on the Java Pathfinder search engine for general message-passing Java programs. Our evaluation of DBSS with various representative fault-tolerant message-passing protocols shows extensive reduction both in time and state space. We also show that DBSS is able to improve over existing partial-order reductions (POR). The approach can be automated and applied to general implementations using static analysis to infer a sound decomposition of the system as a pre-processing phase before running DBSS.

Appendix

For convenience, given a transition system TS and a decomposed property f, we may refer to $DBSS(TS, f)$ by writing $DBSS(TS)$. We introduce the following terminology to ease the following discussion. A state s is said to be *explored* and transition is *fired* if there is a state s' such that line 11 of DBSS is executed.

First, we show that assuming that a state s is not yet explored, all transitions in $enabled(s)$ are fired by DBSS.

Lemma 1. *Given a transition system $TS = (S, S_0, T)$, if a state $s' \in S$ is explored in $DBSS(TS)$ and the condition of line C1 holds, then every $t \in enabled(s')$ is fired in s'.*

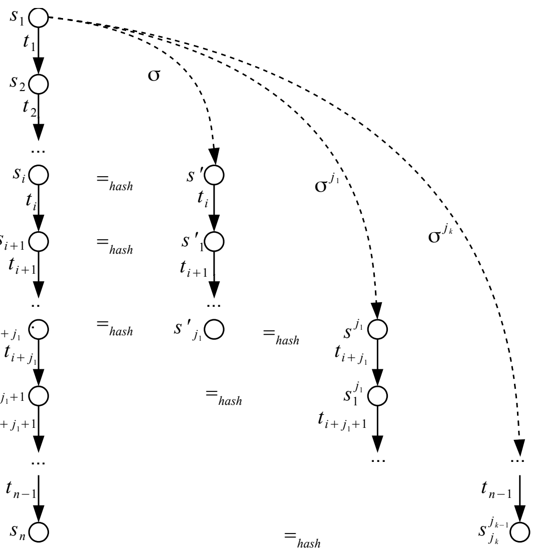

Fig. 5: Illustration of proof of Lemma 2.

Proof: Let t be a transition in $enabled(s')$. If t is next($enabled(s')$), then t is fired in the next iteration of the while loop in line 8. Otherwise, if $enabled(s') > 1$ (cf. condition of line C3), s' is pushed onto the stack and every transition $t' \neq t \in enabled(s')$ is fired when s' is backtracked in the depth-first search. Note that s' is guaranteed to be backtracked after each transition fired in s' because the state graph is finite. ■

We prove that if a state is reachable in TS, there is at least a state explored by $DBSS(TS)$ with the same relevant information. We will use the following notation for convenience: Given a path $\sigma = s_1 \xrightarrow{t_1 \ldots t_{n-1}} s_n$ and $s_n \xrightarrow{t_n \ldots t_{l-1}} s_l$, the path $s_1 \xrightarrow{t_1 \ldots t_{l-1}} s_l$ can be written as $\sigma \xrightarrow{t_n \ldots t_{l-1}} s_l$.

Lemma 2. *Given a transition system $TS = (S, S_0, T)$, $\forall s \in S$, if s is reachable in TS, then there exists a state $s' \in S$ such that $h(s) = h(s')$ and s' is explored by $DBSS(TS)$.*

Proof: The proof is indirect and is illustrated in Figure 5 using the following notation: Given two states $s, s' \in S$, we write $s =_h s'$ if and only if $h(s) = h(s')$. Indirectly, we assume the following: Let $s_n \in S$ be a reachable state in TS so that there is no other state s'' explored by $DBSS(TS)$ with $h(s_n) = h(s'')$. Let $\sigma' = s_1 \xrightarrow{t_1 \ldots t_{n-1}} s_n$ be a path leading to s_n in TS. We know that $n > 1$ because the initial state $s_I = s_1$ is explored by the algorithm. Consequently, there must be $2 \leq i < n$ such that t_i is not fired in s_i. Lemma 1 implies that the condition of line C1 does not hold when s_i is explored. This means that there is a state s' reachable via a path σ explored by $DBSS(TS)$ such that $h(s') = h(s_i)$ and $h(s') \notin$ *reached* when explored. $i < n$ since otherwise s_n would be explored. From Definition 1 (transition decomposition (1)), we also know that $t_i \in enabled(s_i)$ and $t_i \in enabled(s')$. Furthermore, t_i is fired in s' (Lemma 1) and it holds that $h(s_{i+1}) = h(s'_1)$ where $s' \xrightarrow{t_i} h(s'_1)$ (transition decomposition (2)). Let $0 < j_1 \leq n - i$ be the highest natural number such that

$\sigma \xrightarrow{t_i} s'_1 \xrightarrow{t_{i+1}\ldots t_{j_1-1}} s'_{j_1}$ is explored by $DBSS(TS)$. Because of transition decomposition, we know that $h(s'_{j_1}) = h(s_{i+j_1})$. Therefore, $j_1 = n - i$ would imply a contradiction, as this would mean that there is a state s'' such that $h(s_n) = h(s'')$ which is explored. Since j_1 is the highest such index, Lemma 1 implies that the condition of line C1 does not hold when s'_{j_1} is explored. So there must be a state s^{j_1} reachable via a path σ^{j_1} explored by $DBSS(TS)$ such that $h(s^{j_1}) = h(s'_{j_1})$ and $h(s^{j_1}) \notin \mathit{reached}$ when explored. Because of transitivity, we know that $h(s^{j_1}) = h(s_{i+j_1})$. Let $0 < j_2 \leq n - i - j_1$ be the highest natural number such that $\sigma^{j_1} \xrightarrow{t_{i+j_1}} s^{j_1}_1 \xrightarrow{t_{i+j_1+1}\ldots t_{j_2-1}} s^{j_1}_{j_2}$ is a path explored by $DBSS(TS)$. We know from Lemma 1 and the decomposition definition that such a path exists and that $h(s^{j_1}_{j_2}) = h(s_{i+j_1+j_2})$. Since j_2 is the highest such index, Lemma 1 implies that the condition of line C1 does not hold when $s^{j_1}_{j_2}$ is explored. So there must be a state s^{j_2} reachable via a path σ^{j_2} explored by $DBSS(TS)$ such that $h(s^{j_2}) = h(s^{j_1}_{j_2})$ and $h(s^{j_2}) \notin \mathit{reached}$ when explored. We know that $h(s^{j_2}) = h(s_{i+j_1+j_2})$. Continue the construction. Let $0 < j_1, j_2, ..., j_k$ be natural numbers such that $i + j_1 + j_2 + ... + j_k = n$. The construction ends because $1 \leq k \leq n - i$. Inductively, we have that $h(s_n) = h(s^{j_{k-1}}_{j_k})$. Since $s^{j_{k-1}}_{j_k}$ is explored by $DBSS(TS)$, we have a contradiction. ■

Theorem 1 (Soundness)**.**

Proof: Assume that $DBBS(TS, f)$ returns $true$. This means that for every state s explored by $DBSS(TS, f)$, we have $f(s)$. Now we suppose that there exists a reachable state $s' \in S$ such that $\neg f(s')$. From Lemma 2 we know that there exists a state $s'' \in S'$ explored by $DBSS(TS, f)$ such that $h(s'') = h(s')$. Since f is decomposed, we have $\neg f(s'')$ which is a contradiction. ■

Theorem 2 (Completeness)**.**

Proof: $DBSS(TS', f)$ returns a state s and $CE\text{-}stack$ when the condition in line 17 is satisfied. This means that we have $\neg f(s)$. Since s is explored by DBSS it is trivially reachable in TS'.

Now we prove that $CE\text{-}stack$ contains a path leading to $h(s)$ in TS. Since DBSS is a DFS, the sequence of states in the $CE\text{-}stack$ when exploring s is a path σ from $h(s_I)$ to $h(s)$. The path exists in TS because of Definition 2. ■

Theorem 3 (Termination)**.**

Proof: Assuming that the calls in the algorithm terminate, we have to check whether the two loops at line 8 and 7 terminates. The loop at 8 terminate because in each iteration one element is removed from $enabled(s)$ (line 10) and the number of enabled transitions is finite. In line 7, no more states are pushed to the stack once every state has been explored and therefore included in $reached$. Since the number of states is finite and after each iteration one element is removed from the stack (line 19), the loop terminates after a finite number of iterations. Note that the loop at line 20 is guaranteed to terminate since the set of relevant parts of states in $stack$ is included in $CE\text{-}stack$. ■

References

[1] H. Attiya et al. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.

[2] H. Attiya, et al. *Distributed Computing*. John Wiley and Sons, 2004.

[3] P. Bokor, et al. On Efficient Models for Model Checking Message-Passing Distributed Protocols. *FORTE*, pp. 216–223, 2010.

[4] P. Bokor, et al. Efficient Model Checking of Fault-Tolerant Distributed Protocols. In *DSN*, pp. 73-84, 2011.

[5] P. Bokor, et al. Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction. In *Proc. ASE*, pp. 113–122, 2011.

[6] P. Bokor, et al. Role-Based Symmetry Reduction of Fault-Tolerant Distributed Protocols with Language Support. In *Proc. ICFEM*, pp. 147-166, 2009.

[7] D. Bonacki, et al. Symmetric Spin. *Journal on Softw. Tools for Techn. Transfer*, 4(1):92–106, 2002.

[8] C. Cadar, et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, pp. 209–224, 2008.

[9] E. M. Clarke, et al. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods System Design*, 9(1-2):77–104, 1996.

[10] E. Clarke, et al. *Model Checking*. MIT Press, 2000.

[11] M. Correia, et al. Practical Hardening of Crash-Tolerant Systems. In *USENIX ATC*, pp. 453–466, 2012.

[12] S. Ghemawat et al. The Google File System. In *SOSP*, pp. 29–43, 2003.

[13] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer, 1996.

[14] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *POPL*, pp. 174–186, 1997.

[15] P. Godefroid. Exploiting Symmetry when Model-Checking Software. In *Proc. FORTE*. pp. 257–275, 1999.

[16] P. Godefroid, et al. DART: Directed Automated Random Testing. In *PLDI*, pp. 213–223, 2005.

[17] C. Flanagan, et al. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, pp. 110–121, 2005.

[18] H. Guo, et al. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP*, pp. 265-278, 2011.

[19] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2004.

[20] C. N. Ip, et al. Better verification through symmetry. *Formal Methods Sys. Design*, 9(1-2):41–75, 1996.

[21] F. P. Junqueira, et al. Zab: High-Performance Broadcast for Primary-Backup Systems. In *DSN*, pp. 245-256, 2011.

[22] C. Killian, et al. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, pp. 243-256, 2007.

[23] V. Kuznetsov, et al. Efficient State Merging in Symbolic Execution. In *PLDI*, pp. 193-204, 2012.

[24] L. Lamport. The Part-time Parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, 1998.

[25] S. Lauterburg, et al. A Framework for State-Space Exploration of Java-Based Actor Programs. In *ASE*, pp. 468–479, 2009.

[26] A. Miller, et al. Symmetry in Temporal Logic Model Checking. *ACM Computing Surveys*, 38(3), 2006.

[27] C. A. Muftuoglu, et al. Brief announcement: MP-State: State-Aware Software Model Checking of Message-Passing Systems. In *SSS*, 2012.

[28] C. A. Muftuoglu, et al. Scalable Verification of Distributed Systems Implementations via Messaging Abstraction. In *SOSP WiP section*, 2011.

[29] M. Serafini, et al. Application-Level Diagnostic and Membership Protocols for Generic Time-Triggered Systems. *IEEE Trans. TDSC*, 8(2):177–193, 2011.

[30] M. Yabandeh, et al. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, pp. 229–244, 2009.

[31] J. Yang, et al. MODIST: Transparent MC of Unmodified Distributed Systems. In *NSDI*, pp. 213–228, 2009.

[32] http://babelfish.arc.nasa.gov/trac/jpf/

[33] http://www.deeds.informatik.tu-darmstadt.de/home/homepages/peter/mp-basset/

[34] http://www.isi.deterlab.net/

[35] http://hadoop.apache.org/zookeeper/

[36] http://www.macesystems.org/wiki/macemc

[37] http://www.win.tue.nl/ lhol/SymmSpin/