Local Model Checking
EPFL Technical Report LPD-REPORT-163963

Rachid Guerraoui and Maysam Yabandeh
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

Current approaches to model checking distributed systems reduce the problem to that of model checking centralized systems: *global* states involving all nodes and communication links are systematically explored. The frequent changes in the network element of the global states lead however to a rapid state explosion and make it impossible to model check any non-trivial distributed system. We explore in this paper an alternative: a *local* approach where the network is ignored, a priori: only the local nodes' states are explored and in a separate manner. The set of valid system states is a subset of all combinations of the node local states and checking validity of such a combination is only performed a posteriori, in case of a possible bug. This approach drastically reduces the number of transitions executed by the model checker. It takes for example the classic *global* approach several minutes to explore the interleaving of messages in the celebrated Paxos distributed protocol even considering only three nodes and a single proposal. Our *local* approach explores the entire system state in a few seconds. Our local approach does clearly not eliminate the state exponential explosion problem. Yet, it postpones its manifestations till some deeper levels. This is already good enough for online testing tools that restart the model checker periodically from the current live state of a running system. We show for instance how this approach enables us to find two bugs in variants of Paxos.

1 Introduction

At each step of model checking a centralized system, (i) one of the traversed states is selected, (ii) an enabled event is executed on that state, and (iii) the resulting state is added to the list of traversed states. The user-specified invariants are checked against the traversed states after each step and the set of these states grows exponentially with the *depth* of the exploration, i.e., the length of the sequence of enabled events considered. Current approaches to model checking distributed systems [9, 10, 20, 21, 17] reduce the problem to that of model checking a centralized system (Figure 1). The sets explored are *global* states comprising the *local* states of the nodes involved in the distributed system, i.e., the *system* state, as well as the *network* state involving the exchange of messages.

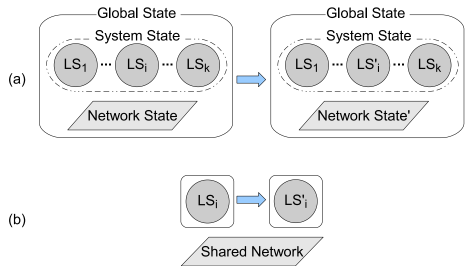

Figure 1: State transition in model checking distributed systems. In (a) the classic global approach, the model checker creates the entire state space of the global states, whereas in (b) our proposed local approach, the network element is eliminated from the stored states and the model checker keeps track of only node local states.

The exponential state space explosion problem manifests itself very quickly in this *global* approach, which makes the model checking of distributed systems practically ineffective. This is because the global state changes following any small change into a node local state or the network state. Consider for instance the celebrated Paxos protocol [12], in the simple setting with three nodes where exactly one proposes at the start, i.e., no contention: it takes the global model checking approach 1514 s (running on a 3.00 GHz Intel(R) Pentium(R) 4 CPU with 1 MB of L2 cache) to explore the interleaving of messages.

The starting point of this paper is a couple of simple, complementary observations: (1) in the global model checking approach, the invariants are checked on each traversed global state, although these invariants are typically specified only on the system states, i.e., the invariants do not involve the network states [10, 20, 21, 17]; [1] (2) for checking invariants that are defined on system variables, visiting the system part is a priori sufficient. Focusing on these states only, and ignoring the network states, significantly reduces the exploration space in comparison to the classic approach where each system state is typically repeated in multiple global states that differ only in the network part.

We present in this paper a *local* model checking approach, which essentially consists in keeping track of the traversed local nodes' states separately by ignoring the network, a priori. Combined, these states are sufficient for invariant checking. The approach is most effective on protocols that involve frequent changes into the network, i.e., the nodes have lots of parallel network activities. For the Paxos example state space with one proposal, our approach explores the entire system state in a few seconds. We show that our approach is *complete* in the sense that any violation of a system state invariant that could be detected by the global approach could be detected by our local approach. Two important remarks are however in order.

First, the combination of node states does not induce system states that are all *valid*: the fact that we ignore the network element, a priori, means that some combinations of node states might not occur in a real run. In other words, although complete, checking invariants on the retrieved system states is *unsound* since it could report a violation on an invalid system state. We address this problem by, a posteriori, verifying every preliminary violation report to make sure the sequence of events leading to the corresponding system state could also happen in a real run. An invariant violation is then reported to the user only if passes this test. If the number of preliminary violations is low enough, which turns out to be the case in our experiments with Paxos, the performance penalty of verifying them becomes negligible.

Second, although our local approach is several orders of magnitude faster than the classic model checking approach, the state explosion problem is not eliminated. (The cost of invalid states created by our approach, although low at the start, will anyway eventually dominate in the general case.) Yet, we believe this can, to a large extent, be addressed by *online* model checking tools where the model checker is run for just a short period (a few seconds): in this case, our approach is efficient enough to search till depths of 20$\sim$30 for the Paxos example state space.

[1]In testing, invariants are used to express the high-level properties of the system. Including the in-flight messages in invariants, although possible in theory, makes defining the invariants too complicated in practice.

In global model checking approach, visiting the system states is part of the exploration process: the new global states (which involve the system states) are explored by running enabled events on the previously visited global states. Therefore, skipping a system state makes the exploration incomplete. In contrast, our local approach separates the exploration of transitions from the creation of system states. This makes it possible to ignore all system states on which the user-specified invariants can inherently not be violated: for instance, the Paxos invariant stipulates that no two decisions should be different and all undecided states can systematically be eliminated.

Summary of Contributions.

- We introduce a new, local approach to model checking distributed systems. Instead of keeping track of global states, we eliminate the network element from the model checking states and keep only track of node local states. Our approach optimistically eliminates the overhead of ensuring soundness of every visited state and instead verifies soundness only on the states that violate the invariants.
- Our approach decouples exploration algorithm from system state space creation. This feature opens the door for optimizations that skip some system states without, however, hurting the completeness of exploration. We benefit from this aspect in our experiments by skipping the system states that could not violate the Paxos invariant.
- Having the exploration, system state creation, and soundness verification decoupled, the model checking process can be embarrassingly parallelized to benefit from the ever increasing number of cores.
- We present an efficient implementation of our approach and we show how this approach tracks bugs in two variants of Paxos, known to be one of the most complex distributed algorithms.

The rest of the paper is organized as follows. § 2 illustrates our approach through a simple example. The background is recalled in § 3. § 4 presents our approach. After presenting the evaluation results in § 5, we contrast local model checking approach with related work in § 6 and conclude the paper in § 7. We prove the correctness of our soundness verification procedure in the appendix.

2 Local Model Checking: A Primer

Here we use a simple example to highlight the difference between global model checking and our local approach. The example we consider here does not attempt to illustrate the performance improvements obtained by our approach but aims at explaining the main idea. The exam-

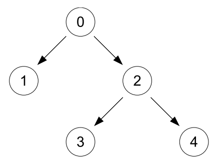

Figure 2: A simple distributed tree algorithm. Each node forwards the message to its children.

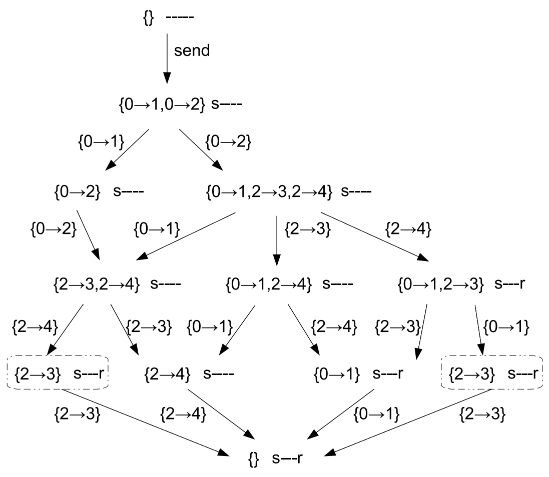

Figure 3: The global state space of the example tree in Figure 2 as explored by a global model checking approach. The network element of the global state is represented by the set of in-flight messages. Each arrow depicts a transition in the model checker from one global state to another. The label besides each arrow indicates the event that triggers the transition. Although the global states inside the rectangles are duplicates, they are not joined into one state, for simplicity of presentation.

ple system is a simple distributed tree structure, depicted in Figure 2. Node 0 initiates a message for Node 4 and changes its state to `sent`. Each node, upon receiving a message, forwards it to its children. Node 4 changes its state to `received` upon receiving the message.

At each step of global model checking, the model checker transitions from a global state to another by running an enabled event, such as handling a message. The global state contains the network state besides the system state, i.e., the local state of all the nodes. The global state space of the example system is depicted in Figure 3. The initial state of each node is denoted ”-”. The system state is shown by concatenating the five states of Nodes 0 to 4. The state of Node 0 and 4 is changed to ”s” and ”r” after the sending and receiving of the message, respectively. Each change into the network element causes creation of a new global state. As one can observe, the number of system states covered by this global state space is much less than its size.

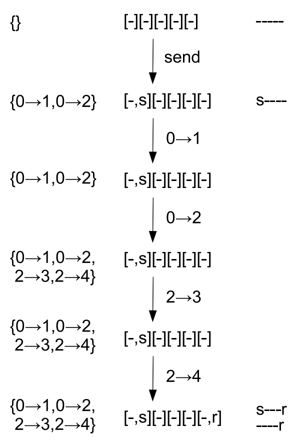

Figure 4: Local model checking approach on the example tree in Figure 2. The first column indicates the changes into the shared network element. The middle column shows the set of states of Node 0 to 4. The first event is the local event of Node 0 that generates the message. The generated message is then added to the shared network element. At each step, an event is selected and is executed on all states of the destination node. The resultant states are added to the list of visited node states if they have not been visited before. The last column shows the new system states created after each step.

Figure 4 illustrates our local approach on the same example system. Here, the network element, i.e., the non-essential part for invariant checking, is separated from the model checking state. Instead, we keep a shared network component that receives the generated messages by all the transitions in the model checking. Observe that the messages added to the network are not removed by the executed transitions. This is necessary for the completeness of the search, because each message must be received by all the states of the destination node, including the node states that will be explored later.

The last column of the figure depicts the new system states created after each step. The system states are created temporarily for the sake of being checked against the user-specified invariants. Observe that, in total, only 4 system states are created in contrast with the 12 global states of Figure 3. Moreover, the last system state, i.e., ”----r” is invalid since Node 4 could not receive the message before it is sent by Node 0. After an invariant is violated on a system state, we run a *soundness* verification phase to ensure the validity of the system state.

3 Preliminaries

We present here a simple model of a distributed system and a basic model checking algorithm based on depth-

basic notions:
N – node identifiers
S – node states
M – message contents
$N \times M$ – (destination process, message)-pair
$C = 2^{N \times M}$ – set of messages with destination
A – internal node actions (timers, application calls)

global state : $(L, I) \in G,\ \ G = 2^{N \times S} \times 2^{N \times M}$
system state (local nodes' states) : $L \subseteq N \times S$
(function from N to S)
in-flight messages (network) : $I \subseteq N \times M$

behavior functions for each node :
message handler : $H_M \subseteq (S \times M) \times (S \times C)$
internal action handler : $H_A \subseteq (S \times A) \times (S \times C)$

transition function for distributed system :

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H_M}{\begin{array}{ll} \text{before:} & (L_0 \uplus \{(n, s_1)\}, I_0 \uplus \{(n, m)\}) \leadsto \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I_0 \uplus c) \end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H_A}{\begin{array}{ll} \text{before:} & (L_0 \uplus \{(n, s_1)\}, I) \leadsto \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I \uplus c) \end{array}}$$

Figure 5: A simple distributed system model

first search. The model is later altered in § 4 to explain local model checking algorithm. We then explain the short run in online model checking, where the model checker can benefit from our local model checking approach.

3.1 System Model

Figure 5 describes a simple model of a distributed system, taken from [20].

System state. The *global state* of the entire distributed system encompasses (1) the system state, i.e., local states of all nodes, and (2) in-flight network messages. We assume a finite set of node identifiers N (e.g., corresponding to IP addresses). Each node $n \in N$ has a local state $L^n \in S$. A node state encompasses node-local information, such as explicit state variables of the distributed node implementation, the status of timers, and the state that determines application calls. A network state corresponds to the set of in-flight messages, I. We represent each in-flight message by a pair (N, M) where N is the destination node of the message and M is the remaining message content (including sender node information and message body).

Node behavior. Each node in the system runs the same state-machine implementation. The state machine has two kinds of handlers: (i) a message handler executes in response to a network message; (ii) an internal handler executes in response to a node-local event such as a timer and an application call. We represent message handlers by a set of tuples H_M. The condition $((s_1, m), (s_2, c)) \in H_M$ means that, if a node is in state s_1 and it receives a message m, then it transitions into state s_2 and sends the set c of messages. Each element $(n', m') \in c$ is a message with target destination node n' and content m'. $((s_1, a), (s_2, c)) \in H_A$ represents the handling of an internal node action $a \in A$. An internal node action handler is analogous to a message handler, but it does not consume a network message.

System behavior. The behavior of the system specifies one step of a transition from one *global state* (L, I) to another global state (L', I'). We denote this transition by $(L, I) \leadsto (L', I')$ and describe it in Figure 5 in terms of handlers H_M and H_A. [2] The handler that sends the message, inserts the message directly into the network state I, whereas the handler receiving the message simply removes it from I. To keep the model simple, we assume that transport errors are particular messages, generated and processed by message handlers.

Observations. The following observations can be derived from the definitions of H_M and H_A in Figure 5: (i) Except the node in which the event is executed, the state of other nodes, i.e., L_0, is untouched. This implies that to execute an event on node n, we require only the state of node n; (ii) To execute H_M with message m on node n, the only required part from the network state is tuple (n, m): the rest of the network state, i.e., I_0, is untouched. These observations indicate that the entire global state of the system is not required to execute a handler in the model checker.

3.2 Global Model Checking

Global model checking is based on a standard search algorithm such as bounded depth-first search (B-DFS) for tracking invariant violations in the transition system captured by relation $\leadsto$ of Figure 5. The search starts from a given global state, which, in the standard approach, is the initial state of the system. By executing enabled handlers (H_M and H_A) on the traversed global states, the search systematically explores reachable global states at larger and larger depths and checks whether the states satisfy the given `invariant` condition.

Soundness. B-DFS is sound in the sense that all violation reports could also occur in a real run of the system. In other words, there is no *false positive* in the reported bugs. Moreover, all traversed states are valid and

[2] $\uplus$ in the handler definition means disjoint union.

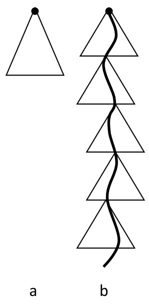

Figure 6: The covered state space in model checking by (a) a model checker started from the initial global state, and (b) an online model checker that restarts periodically from the current live system state. The curved line represents the states explored by the running system.

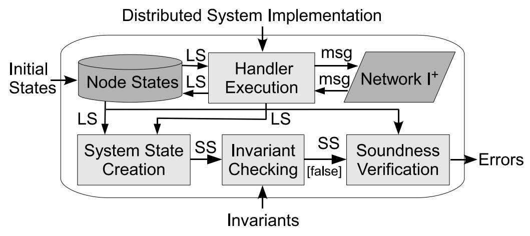

Figure 7: In our local approach, the handler execution works only on node states and produces new node states. Local and system states are denoted ”LS” and ”SS”, respectively. The messages are not removed from the shared network component after execution. The soundness verification checks the validity of a system state, only after an invariant violation is reported.

could also be created in a real run. The sufficient part for soundness, however, is only the reported violations to the developer. We will show later that our local model checking is also sound, even though some system states created a priori might be invalid.

Completeness. An exploration algorithm is complete if, given enough time and space, it can explore all system states. In other words, completeness is satisfied if there is no *false negative* in bug reporting. Although B-DFS is complete, due to an inherently limited time budget, in practice it can explore only a small fraction of the state space of complex algorithms.

3.3 Online Model Checking

Due to the state space explosion problem, a model checker of a distributed system cannot explore deeper than certain steps in a limited time budget. For example, even in the very small state space experiment of Figure 10, where only one node proposes once, the model checker cannot explore more than 15 events within a minute. An online model checker is, on the other hand, restarted periodically from the live state of a running system. As a consequence, the model checker has a chance to explore more relevant states at deeper levels, instead of getting stuck in the exponential explosion problem at some very shallow depths.

Figure 6 illustrates the use of a model checker in parallel with a running system. As one can see, an online model checker does not require solving the exponential explosion problem completely; it is rather sufficient to explore till a depth that is useful for testing purposes.

4 Local Model Checking

The architecture of our local model checking approach is depicted in Figure 7. In this approach, the model checker keeps track of node states separately: set LS^n contains all the traversed states of node n. This is enough to recreate the *system states* upon which the invariants are checked. After a preliminary violation report on a system state, the validity of the system state is checked by a soundness verification module. If the system state is confirmed to be valid, the error is then (and only then) reported to the developer.

Instead of keeping a separate network state for each global state, we keep one single network state I^+ that contains all generated messages during the model checking (Figure 7). The execution of handlers must change to work with the shared network state I^+ (Figure 8). In the new handlers, H'_M and H'_A, the network state of the input global state is replaced with the new shared network state, I^+. Furthermore, the received message, (n, m), is not removed from I^+ after the execution of handler H'_M. In other words, the content of I^+ is always increasing. It is not hard to see that the altered handlers preserve the completeness of the search: for each Transition $(L_p, I_p) \rightsquigarrow (L_q, I_q)$ in H_M, there exist a corresponding Transition $(L_p, I_p^+) \rightsquigarrow (L_q, I_q^+)$ in H'_M. We discuss soundness later in this section and we prove it in the appendix.

Recall from $\S$ 3 that, to execute a handler on node n, the only required state is the state of node n, i.e., LS^n. Therefore, the stored node states are enough to execute the handlers and we do not need to recreate the system state for that. To execute network handlers, however, we require also message (n, m) from the network (we do not need the whole network state.). As shown in Figure 7, the handler execution module receives input only from node states and the shared network module.

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H'_M}{\begin{array}{ll} \text{before:} & (L_0 \uplus \{(n, s_1)\}, I^+ \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I^+ \uplus \{(n, m)\} \uplus c) \end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H'_A}{\begin{array}{ll} \text{before:} & (L_0 \uplus \{(n, s_1)\}, I^+) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I^+ \uplus c) \end{array}}$$

Figure 8: The altered handlers in local model checking.

```
1  proc findBugs(liveState, invariant)
2    $LS$ = emptySet(); $I^+$ = emptySet();
3    foreach $n \in$ N
4      $LS^n$ = $LS^n \cup$ {liveState$^n$};
5    while ( ! StopCriterion )
6      if ( $\exists((s, e), (s', c)) \in H'_M$ where $LS^n_s \in LS^n$, $(n, e) \in I^+$ ||
7           $\exists((s, e), (s', c)) \in H'_A$ where $LS^n_s \in LS^n$)
8        addNextState($n, s, s', e, c, LS$);
9        checkSystemInvariant($n$, $s'$, liveState, $LS$, invariant);
10
11 proc addNextState($n, s, s', e, c, LS$)
12   $I^+$ = $I^+ \cup c$;
13   $LS^n$ = $LS^n \cup s'$;
14   $LS^n_{s'}$.predecessors.add($s, e$);
15
16 proc checkSystemInvariant($n$, $s'$, liveState, $LS$, invariant)
17   foreach $ss$ : system state
18     where $\forall n_k.\ ss^{n_k} \in LS^{n_k}$
19     if ( ! invariant($ss$) )
20       if ( isStateSound(liveState, $ss$) )
21         reportBug($ss$); // a bug found
22
23 proc isStateSound(liveState, state)
24   //obtain all sequences following predecessor pointers
25   foreach $h$ : list of event sequences where
26     $h^n \in$ (state$^n$.predecessors)$^*$ // $^*$ is closure operator
27     if ( isSequenceValid(liveState, $h$) )
28       return true;
29   return false;
30
31 proc isSequenceValid(liveState, $h$)
32   state = liveState;
33   while ($\exists n$, nextState where state $\overset{h^n.first()}{\rightsquigarrow}$ nextState)
34     state = nextState;
35     $h^n$.popFirst();
36   return $h$ == $\emptyset$;
```

Figure 9: Local model checker algorithm.

4.1 Algorithm

Figure 9 presents our algorithm. Variable LS in Figure 9 refers to the set of all visited node states, i.e., (n,s), where n is the node index and s is the node state. Procedure `findBugs` takes the live state of the system as input, to initialize Variable LS at Lines 3-4. As in global model checking , the search terminates upon exceeding some bounds, such as running time or search depth (Line 5).

Handler execution. At each step of the model checking, an enabled handler, either network or local, is executed. For network handlers, the algorithm at each step checks all network messages in Variable I^+. To obtain the enabled network events, for each message e of node n in I^+, all the currently visited states of node n are considered (Line 6). The corresponding network handler is then executed (Line 8) and Procedure `addNextState` is called on the resultant state, s', and the set of new network messages, c. Note that the messages that are added to network I^+ in this round of the loop (i.e., c in Figure 8) will be considered on the node states in the next round.

As in the global model checking approach, the node local events, such as timers and application calls, are defined based on the node local states. In other words, the value of node state LS^n_s determines which of the local events are enabled. To obtain the enabled local events, we look at all visited node states and retrieve their local events (Line 7).

In Procedure `addNextState`, the set of new network messages is added to the shared network, I^+ (Line 12). If the state of node n has changed, it is added to set LS (Line 13). Variable $predecessors$ keeps track of all the last immediate node states as well as the executed events on them that led to the current node state (Line 14). We need more than one pointer in Variable $predecessors$, since the same node state might be reached by executing different sequences of events.

Creating system states. The invariants are defined on system states. Since we do not store the system states, they must be temporarily created for the sake of invariant checking, which is performed by Procedure `checkSystemInvariant`. The procedure is called after each change to LS. Each system state ss is created by combining the node states of different nodes in LS. (We will explain in § 4.2 an optimization that prevents revisiting system states.)

The only purpose of system state creation is to verify the user-specified invariant in on them. Therefore, we can design invariant-specific system state creation to bypass the system states that could not possibly violate the invariant. In other words, if $in' \Rightarrow in$ and $in'(ss)$ is false, verifying $in(ss)$ is not necessary. In order for this to be useful, in' should be cheaply verifiable. One way to achieve that is to decompose in' into some locally verifiable properties. For example, the Paxos invariant specifies that no two nodes should choose different values. In system state creation, therefore, we can ignore the node states in which no value is chosen yet. If the invariant is defined on node states separately, the invariant-specific system state creation can also bypass the system states in which none of node states have violated the invariant. For example, in RandTree distributed tree structure, one invariant specifies that in all node states the children and siblings must be disjoint sets.

Soundness verification. Since taking all combinations of node states could result into some invalid system states, the preliminary violation of an invariant could be unsound. Procedure `isStateSound`, therefore, verifies validity of the system state upon which an invariant is violated. Variable $predecessors$ in each node state s' contains all the last immediate node states that led to s'. Following these pointers, we obtain the set of event sequences that could lead to s'. If a system state is valid, then there exists at least one valid combination of its node states' event sequences.[3] Lines 25-26 loop on all these combinations and invoke Procedure `isSequenceValid` on each. The number of paths could exponentially increase with sequence size, which is the major cost in soundness verification.

Procedure `isSequenceValid` receives n event sequences (h^i, $i \in N$) corresponding to n nodes in the system. The procedure then looks for a valid total order for execution of the events, in which an event is executed only after it is enabled. For example, to execute a network handler that receives message m from node s, the message must first be generated by an event in s. At each step, the procedure verifies whether any of the events on top of the h^i stacks are enabled (Line 33). The first enabled event is greedily selected for execution based on the definition of handlers in Figure 5 (the events are executed similar to a real run of the distributed system.). The loop continues until there are no enabled events on top the h^i stacks. Afterward, the fact that h is empty (Line 36) indicates that the set of sequenced events in h was possible to run and hence its corresponding system state is valid.

Procedure `isSequenceValid` returns true if and only if the corresponding input system state is valid. Appendix A provides formal arguments for the above statement. Intuitively, since an event in not popped out from h unless it is a valid, enabled event, the feasibility of executing all events implies that the system state is valid. It actually does not matter which enabled event is selected for the next step, since the demanded order by the sequences will be eventually enforced by receiving only the messages that are already generated.

[3]Each event sequence must deterministically lead to the same node state. If the event handler implementation is dependent on some non-deterministic values, those values must be recorded as part of the event, to be replayed deterministically on a re-execution of the event.

4.2 Implementation Details

Local model checking can be used for testing programs in all languages, including C++. Basically, any of existing stateful global model checking tools could be instrumented to run our proposed algorithm. Our prototype implementation of the local model checking approach, denoted LMC, uses MaceMC [10], a model checker for distributed system implementations in the Mace language [9]. Mace programs are basically structured C++ implementations, in which the boundary of handlers and the protocol messages need to be specified. This helps Mace automatically generate the code for serialization and deserialization of the protocol state, and simplifies the definition of events in the model checker.

We use CrystalBall [20, 19] for online running of the model checker, in parallel with a live distributed system. The model checker is then periodically restarted from the taken snapshot. It is worth noting that LMC improves the performance of model checking anyway, independent of CrystalBall. For testing of complex programs, however, we use the online model checking approach to restart the model checker before exponential explosion manifests.

We changed MaceMC to work only on one global object of the network simulator, i.e., I^+. To change the network handler implementations from H_M to H'_M (Figure 8), we changed the network simulator not to remove a message after its delivery. MaceMC automatically generates specific functions for (de)serializing a module state in the service. We added specific functions to save and restore the whole service stack. This is required for multi-layer services such as 1Paxos [18] (one of the protocols we check), which uses Paxos as its lower layer module. To efficiently check for duplicate states, we use the hashes of the serialized states. For each node n, the hashes of the traversed states are kept in a `set` structure. The serialized state itself is stored in a `deque` structure to benefit from its efficiency in random access.

Each message keeps track of the number of node states on which it has been executed. Therefore, in each round, each message is checked only on the newly added states, by jumping over the old states. Instead of the actual event, its hash is added into the predecessor pointers. These hash values will be checked against the hash values of the enabled events, later when we verify the soundness of the system state.

Test driver. The test in model checking a service is generally driven by an application sending requests to the service. In Paxos for example, an application sending propose requests to the service is the test driver of the model checker. The more complex the test driver, the larger the generated state space is. A careful design of the test driver could greatly impact the efficiency of model checking. In our Paxos experiments, the test driver proposes values for a particular index. The index is selected from recent chosen proposals, where not all the nodes have learned the proposal yet. Otherwise, a new index is used for the proposal.

System states. To avoid revisiting system states, checking invariants on system states is performed only after visiting a new node state, which implies the possibility for creating new system states. For each new node state (n,s), the system states are created by iterating over the states of all the nodes except node n and loading them. This is because the combinations of the previously visited states of node n and the node states of the other nodes have already been verified in previous rounds. It is worth noting that this optimization could make the model checking incomplete because the handler execution that has not produced a new node state could still change the pointers in $predecessors$, which means the possibility of a valid event sequence for a previously rejected system state. To address this issue we could cache the system states in which an invariant is violated and reverify them after the changes into LS that affect them.

Beside the general approach for system state creation, we also implemented an invariant-specific variation, denoted LMC-OPT, optimized for the Paxos main invariant. In this variation, we map the node states to the values that are chosen in them. Because most of the node states have not chosen any value, lots of them will not be included in this mapping. When creating system states, we thus select only the node states that at least two of them are mapped to different values. This optimization helps avoid the creation of lots of redundant system states and consequently omits their corresponding invariant checking and soundness verification steps.

Soundness verification. Procedure `isStateSound` uses pointers in Variable $predecessor$ to find event sequences that could lead to the input node states. For the sake of simplicity in implementation, we ignore the *self-references* in following the pointers in $predecessor$. Although in theory this could make the exploration incomplete, in practice the search in the limited time budget is incomplete anyway and benefiting from the simplicity is, hence, preferable. Moreover, after the soundness verification on a system state is finished, some more pointers could be added into $predecessor$ by the process of local model checking. Therefore, a complete exploration should invoke soundness verification after each change into a $predecessor$. However, an efficient implementation of that would be complex since it should check only for the newly added pointers. For the sake of simplicity in implementation, we invoke soundness verification only after a new node state is visited.

Procedure `isSequenceValid`. The validity of a set of sequenced events could in general be checked by executing them in a simulator (the same way the global model checking approach transitions from one global state to another). If no event from the sequences is enabled in the simulator, it indicates that sequence of events is not valid. Although using the simulator simplifies the implementation, initializing the simulator at each run of the soundness verification is expensive since it involves loading the test driver.

For efficient implementation of soundness verification module, we take advantage of the following observation. The role of the simulator in executing event e on node n is to (i) updates the state of node n, (ii) remove the message m from the network if e is a network event for delivery of message m, and (iii) add the set c of messages, resulting from the execution of e, to the network.

The consumed message by a network event is specified by its corresponding hash in the node event sequence, which was given as a part of the input to the procedure. The set of the generated messages by an event execution can also be remembered by keeping the hashes of the generated messages in $predecessor$. In this manner, the input to Procedure `isSequenceValid` is the set of sequenced events as well as the set of generated messages by each event. The execution of event e in Procedure `isSequenceValid` can then be simplified as follows:

1. A local event e is always enabled. A network event e is enabled if the hash of the required message is found in the set of generated message hashes, net.
2. If event e is enabled, then pop it out from the sequence. If event e is a network event, remove the hash of the corresponding message from set net.
3. After popping out event e, add its generated message hashes to set net.

The above implementation simplifies Procedure `isStateSound` to some integer comparison operations and therefore makes checking the validity of a set of sequenced events very efficient.

Local assertions. LMC checks for the system invariants defined on the system state. The source code could still be instrumented by some local assertions by which the developers have benefited in earlier stages of testing. The violation of the local assert statements in the process of local model checking could imply that either (i) the node state is invalid, perhaps because of delivering an unexpected message, or (ii) there is a bug in the system under test. Checking the latter case necessitates (i) creating all the system states by combining the node state with all states from other nodes, and (ii) checking the validity of those states by invoking soundness verification. This approach is very expensive since it involves lots of invocation of soundness verification.

In general we could ignore violation of a local assert since a protocol bug will eventually manifest itself by violating a system invariant. Alternatively, we can discard the node state on which the assertion is violated assuming that the assert violation implies the invalidity of the node state. In the applications we tested, the assert statements were mostly used to exclude the receipt of unexpected messages, i.e., the case that could be caused by conservative message delivery policy of LMC, which delivers the message to all the node states of the destination. We, therefore, benefited from the local assert violations by discarding the corresponding node states.

Local events. The presented algorithm in § 4 is complete in the sense that, given enough time and space, it explores all possible states. In practice, however, we have a short time budget to check the reachable states from a given current state. Therefore, the developers might be interested to favor some events to be explored first in the search. Hence, in each round we put a bound on the number of local events that each node can execute; after finishing the round, the bounds are increased and the model checking is started from scratch. This approach is in spirit similar to B-DFS search, where the search depth is increased at each step.

Duplicate messages. In general, a node could infinitely issue duplicates of the same message. For example, in the verified Paxos implementations, the same Chosen message will be sent over and over to the proposer that insists for an already chosen value. To favor the main protocol messages in the limited time of search, we have put a limit on the number of duplicate messages sent from a source to a destination node. This limit is set to zero for the results reported in this paper. Note that the duplicate messages can be postponed to be processed later, after processing some main protocol messages.

As we explained, to ensure completeness, the messages are never erased from the network object, I^+. However, if node state $s \overset{m}{\rightsquigarrow} s'$ where m is a network event, execution of m on s' is redundant since m is already executed in the sequence. To avoid redundant executions, we keep the history of the messages that has been executed to obtain the state: a network event is considered on a state only if it is not in the history of the state. After executing message m on node state s that results into node state s', we apply the two following rules to maintain the history: (i) $s'.history = s.history$, (ii) $s'.history.addLast(m)$. Thus, message m will never be executed on node state s' as well as its descendants. Maintaining history gets complicated if state s' already exists since we need to maintain separate histories for different sequences that lead to s'. We have simplified the implementation by applying rule (i) only if the state does not exist. Since the run of LMC in the limited time budget is not complete anyway, we decided to favor simplicity over completeness here.

4.3 Scope of Applicability

In contrast with global model checking that validity of each traversed state is ensured, local model checking optimistically allows visiting invalid states and verifies the validity of a state only after it violates an invariant. If we have a few preliminary violations, the optimistic approach of local model checking performs better since it does not pay for ensuring validity of every single visited state. Otherwise, the cost of soundness verification dominates. For example, in online model checking, if a run of the model checker is revealing a bug in the protocol, it is likely to see lots of violation reports caused by both valid and invalid event sequences. Perhaps, one solution could be running both local and global model checker in parallel and use the result of the one that finishes sooner.

By eliminating the network element from the model checking state, local model checking reduces the explored state space since each system state is repeated in multiple global states that are different only in the network part. The larger the network state space is, the more space and time is saved by eliminating it. Local model checking is, therefore, most effective for the protocols that are chatty, i.e., exchange lots of messages to service a request. Otherwise, if the nodes rarely communicate, the change into the network is rare and therefore there is not much to be saved by local model checking.

In contrast with global model checking, local model checking considers interleaving of parallel network events only when they turn out to be dependent. LMC, therefore, avoids lots of unnecessary event interleaving. For example, upon receipt of the Accept message, the nodes in Paxos broadcast some Learn messages in parallel, which enables LMC to perform much better than global model checking. The more parallel network activities in the system, the more effective LMC is. For example, we could not expect much from LMC in a chain system in which each node simply forwards the input message to the next.

The current implementation of LMC assumes a best-effort, lossy network, i.e., IP. The protocols that use UDP can, therefore, be directly model checked with LMC. Although, TCP could be considered as part of the protocol stack, in practice this is not efficient, and TCP is usually simulated in the model checker. To do so, LMC implementation should be also augmented to benefit from the fact that reordered messages in a connection will eventually be rejected by TCP and could, hence, be ignored, saving some unnecessary handler executions in the model checker.

5 Evaluation

We evaluate in this section the performance of our local model checking approach compared to a classic global one. We also illustrate the ability of our tool, LMC, in finding bugs in Paxos and its variant, 1Paxos.

We use Paxos as a complex distributed testbed to evaluate the performance of the proposed local model checking approach. In usual implementations of Paxos, each node implements three roles: proposer, acceptor, and learner. Multiple proposers can concurrently propose values for the same index. The Paxos invariant (also known as the Paxos safety property) stipulates that no two nodes will choose different values for the same index. A proposition (i.e., proposing a value for an index) starts by broadcasting Prepare messages to the acceptors. The acceptors respond by a PrepareResponse message. After receiving it from a majority of acceptors, the proposer broadcasts an Accept message to the acceptors. The value in the Accept message is the value returned by the PrepareResponse message with the highest proposal number, which reflects the accepted values from previous proposals, if there is any. Each acceptor then broadcasts a Learn message to the learners. A value is chosen by the learners after receiving the Learn message from a majority of acceptors.

For benchmarking purposes, we use a state space of Paxos running between three nodes, in which one node proposes a value once and the others react to this proposal by communicating using Paxos messages. The long chain of messages following each proposal could be received in a variety of orders, which all must be considered by a model checker. For each experiment, we report on evaluation of 3 algorithms: (i) B-DFS (explained in § 3), (ii) LMC-GEN, which is the non-optimized, general version of our local model checker (LMC), and (iii) LMC-OPT, which is a version of our local model checker optimized for the Paxos main invariant according to § 4.2. The experiments are run on a 3.00 GHz Intel(R) Pentium(R) 4 CPU with 1 MB of L2 cache.

5.1 LMC Speedup

Here we evaluate the speedup in model checking that we can get by our tool, LMC. Figure 10 presents the results for the example state space, in which only one node proposes a value. This state space is relatively small and yet effective in finding bugs when it is explored through an online model checker. The depth of the state space is 22 events (three initialization, one propose local event, three Prepare messages, three PrepareResponse messages, three Accept messages, and nine Learn messages). LMC explores also longer sequences of events (up to 25) since it could also explore some invalid sequences of events. [4] The elapsed time is depicted in a logarithmic scale to illustrate exponential state space explosion problem. In B-DFS, the exponential explosion starts from the very early steps, which makes the exploration take 1514 s. The growth in LMC-OPT is much less steep, which allows it to finish the model checking in just 189 ms ($\sim$8,000 times faster than B-DFS).

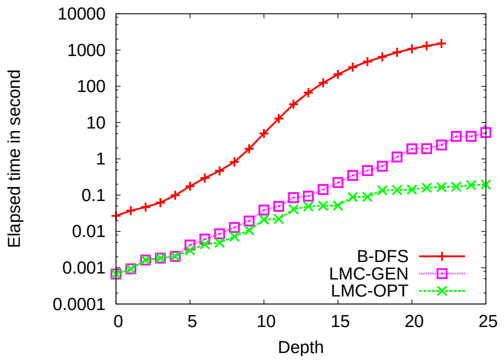

Figure 10: The elapsed time in model checking Paxos where only one out of three nodes proposes a value.

The growth in LMC-GEN, although still much more gentle than B-DFS, is steeper than LMC-OPT. The exploration finishes in 5.16 s which is still $\sim$300 times faster than B-DFS. The extra delay is due to the creation of the system states out of the explored node states, which in LMC-OPT is optimized to be performed only after a different value is chosen. Figure 11 depicts the number of explored states. The number of created system states in LMC-GEN, although much less than B-DFS, is much more than the total number of node states, denoted LMC-local in the figure. LMC-OPT, on the other hand, drops the number of created system states to zero since there is no bug in the Paxos implementation to lead to any preliminary violations. (LMC-OPT creates a system state only if it is likely to invalidate the invariants.)

The total number of performed transitions in B-DFS is 157,332. LMC drops this to 1,186, which is $\sim$132 times less. This is because a LMC transition from state s to state s' in node n, is redundantly executed several times in global model checking approach (once for each global state that encompasses s and its network event is enabled).

This state space of Paxos is very useful in online model checking, where we expect the model checker to seek for a bug in the time budget of less than a minute. Both LMC-OPT and LMC-GEN can finish this state space in this duration and LMC-OPT can continue for more complicated state spaces where there is some time left (as we explained in § 4.2, the model checker, in favor

[4] The invalid sequences will be eventually rejected by soundness verification phase if they violate some invariants.

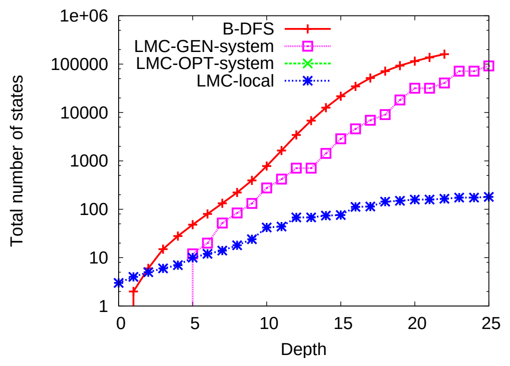

Figure 11: The number of explored states. The number of system states explored by LMC-OPT is zero and is, hence, not plotted in the figure.

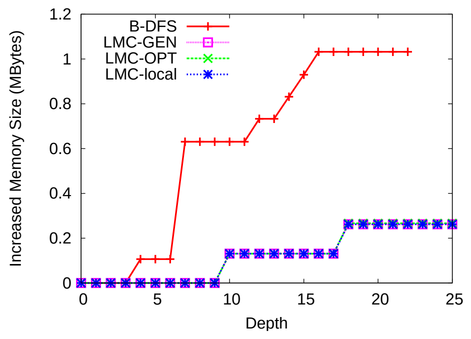

Figure 12: The consumed memory. The numbers for all configurations of LMC are close together and are, hence, overlapped in the figure.

of time, starts with small state spaces by gradually increasing the number of allowed local events.). This is in contrast to B-DFS that will not go further than depth 12 within a minute.

5.2 LMC Scalability Limits

We showed that LMC manages to finish a valuable state space in less than a few seconds. This is already good enough for practical applications such as online model checking that restarts the model checker every few seconds. From the theoretical point of view at least, it is interesting to find the scalability limits of LMC, i.e., the point where the postponed exponential explosion problem eventually manifests and makes LMC ineffective for the rest of the exploration. To this aim, we choose a much bigger state space, where two separate nodes propose two values. The depth of the state space is 41 events, which is two times the events in one error-free proposal. (LMC explores also longer sequences of events, up to 68, since it could also explore invalid sequences of events.)

Due to exponential explosion problem, neither B-DFS nor LMC could finish the state space, even after hours of running. Within this duration, B-DFS explores till 20 steps (out of maximum depth of 41) and LMC searches till 39 steps (out of maximum depth 68). The major contributor to the slowdown of LMC is the expensive task of soundness verification. The number of different event sequences that must be considered for checking validity of a system state exponentially increases with the search depth. In the above example that the search depth of LMC is 39, each invocation of soundness verification induces $\sim$10 s into the algorithm. Invocations of soundness verification are much less in the smaller state space in which only one node proposes a value.

5.3 LMC Memory Requirements

Figure 11 depicts the very fact that the number of node states explored by LMC is much less than the total number of system or global states. Because LMC keeps track only of node states, and the system states are created only temporarily, LMC is expected to require very low memory footprint. Figure 12 verifies this expectation by depicting the memory footprints of different algorithms. LMC-local denotes the run of LMC-OPT in which the creation of system states is disabled. The difference between LMC-local and LMC-OPT (resp. LMC-GEN) indicates the memory overhead of system state creation as well as soundness verification. Although there is a marginal overhead for system states, the memory eventually returns to the system by reusing the deleted objects. The consumed additional memory by all algorithms is less than 1 MB which can totally fit into the L2 cache. However, the exponential trend in memory consumption of B-DFS, promises the ineffectiveness of B-DFS for deeper searches. LMC in contrast uses the memory very efficiently ($\sim$200 KB in total) and this amount grows linearly by increase in search depth.

5.4 LMC Overheads

Here we break down the overheads that limit the scalability of LMC. LMC has two major overheads: (1) creation of system states out of traversed node states, and (2) verifying soundness of the preliminary violations. The precise load of each overhead depends on the particular system under test. Figure 13 illustrates the overheads of LMC-OPT in the buggy implementation of Paxos, for which the corresponding bug is reported in § 5.5. In LMC-system-state the soundness verification phase is disabled and in LMC-explore the creation of system states is eliminated.

The difference between LMC-system-state and LMC-

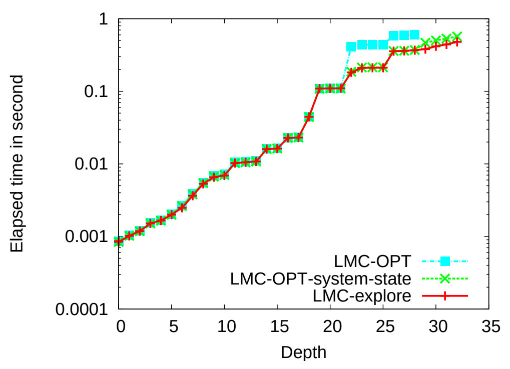

Figure 13: The overheads of LMC in model checking Paxos in which a bug is injected.

explore captures the overhead of creating the system states and checking the invariant on them. The overhead is zero until 21 steps since the unnecessary system states are bypassed by the optimization in LMC-OPT. Afterwards, the overhead increases with the depth search, because as the exploration moves forward, more node states are explored and hence more combinations of them must be considered for system state creation. The difference between LMC-OPT and LMC-system-state reveals the overhead of soundness verification. (LMC-OPT did not go further than 28 steps, the level at which the injected bug is rediscovered.) This overhead is the major contributor to the exponential increase in model checking time. The reason is that not all combinations of node states are valid, and the more node states are traversed, the more invalid system states will be checked. On the other hand, since the injected bug is close to manifest in this run of the model checker, the number of invalid combinations of node states that violate the invariant increases. LMC-OPT triggers the soundness verification for 773 times, and each call takes 45 ms in average. Overall, 427,731 different event sequences were checked by the soundness verification module.

5.5 Testing Paxos

In this section, we report on our experiments in injecting a bug into a Paxos implementation and then running our prototype to verify its ability to detect the bug. The bug we injected was reported in a previous implementation of Paxos [13]: once the leader receives the PrepareResponse message from a majority of nodes, it creates the Accept request by using the submitted value from the last PrepareResponse message instead of the PrepareResponse message with highest round number. The installed invariant is the original Paxos invariant: no two nodes can choose different values.

Every one minute, the online model checking framework takes the live system state of a running Paxos application and use that to initialize the next run of LMC. The application encompasses three nodes, each node proposes its Id for a new index and then sleeps for a random time between 0 and 60 s. The nodes communicate using UDP and 30% of non-loopback messages are randomly dropped to allow rare states to be also created.

The bug was detected after 1150 seconds. The run of LMC that detected the bug was initialized with the following live state: for index k_i, node N_1 has proposed value v_1, nodes N_1 and N_2 have accepted this proposal, but due to message losses only N_1 has learned it. Starting from this system state, LMC detected in 11 s a violation of the Paxos invariant in the following scenario: N_2 proposes a new value v_2 but its Prepare messages is not received by N_1. N_2 responds by a PrepareResponse message containing value v_1, because this value was accepted by N_2 in the previous round. However N_3, since had not accepted any value for index k_i, responds back by the same value proposed by N_2, v_2. Receipt of PrepareResponse of N_3 triggers the bug, and N_2 broadcasts an Accept message for v_2 instead of v_1. Eventually this leads to choosing value v_2 in N_2, which is different from the value chosen by N_1, i.e., v_1.

5.6 Testing 1Paxos

In this section, we report on running our prototype to find bugs on a variant of Paxos, denoted 1Paxos [18]: this is an efficient variation of Multi-Paxos [2] that uses only one acceptor. Upon failure, the active acceptor is replaced with a backup acceptor by the global leader. Therefore, it is necessary that the acceptor and leader roles to be assigned to two separate nodes. To uniquely identify the global leader and the active acceptor, 1Paxos uses a separate consensus protocol referred to as PaxosUtility [18]. The global leader and the active acceptor are identified by the last LeaderChange and AcceptorChange entries in the PaxosUtility, respectively. In this experiment, we have implemented PaxosUtility using Paxos itself. 1Paxos is more complex than Paxos for it comprises more logic. Here we use the same setup that was used for testing Paxos, with the difference that the application instead of proposing a value triggers the fault detector with the probability of 0.1 to stress the fault tolerance mechanisms of 1Paxos. In 225 s, the tool found one new bug in 1Paxos that we report in the following.

The bug was created because of the wrong usage of the ”++” operator; if the operator is used after the operand, the returned value is the original value and not the increased one. The developer had made this mistake in the initialization function, where the leader is set to the first node of the members and the acceptor is set to the second. The used command was

```
acceptor = *(members.begin()++);
```

which makes the acceptor be the same node as the leader. The bug is of course fixed by putting the ”++” operator before the operand, i.e.,

```
acceptor = *(++members.begin());
```

During the live run, node N_3 attempts to be the leader by inserting a LeaderChange entry into the PaxosUtility. At this moment, it obtains from the PaxosUtility the correct value of the active acceptor, which is N_2. After N_3 becomes leader, it proposes value v_3 for index k_i, which is accepted by the acceptor, i.e., N_2. N_2 then broadcasts a Learn message, which is received by N_3 as well as itself. At this point the live system state, in which all nodes except N_1 have chosen value v_3 for the index k_i, is taken to be used by LMC.

Starting from the above system state, LMC highlights the following scenario that violates the Paxos invariant: N_1, which still assumes it is the leader, proposes value v_1 for index k_i to the acceptor. Since N_1 considers itself to be the leader, according to the protocol, it does not refer to PaxosUtility to get the acceptor Id. Therefore, N_1 uses its current value, which is set to N_1, i.e., its own Id, due to the initialization bug described above. N_1 accepts the proposal and sends a Learn message to N_1. Upon receiving the loopback message, N_1 assumes value v_1 as chosen for index k_i. This violates the Paxos invariant since other nodes have chosen a different value, i.e., v_3.

6 Related Work

Cartesian abstraction. This is an abstraction-based verification technique where an overapproximated variant of the program is model checked, instead of the original one [1]. Due to overapproximation, the reported bugs are not sound, which makes the technique mainly useful for correctness proving, benefiting from the completeness of the search. Malkis et al. [14] achieved thread-modular model checking [5, 15] using a Cartesian abstract interpretation of multi-threaded programs. Each thread state consists of the thread local variables plus the global variables. For each thread, the model checker separately explores possible valuations of the thread local variables as well as the global variables. The approximation comes from the fact that the valuations of the global variables by a thread are also used by other threads, ignoring the causal order for obtaining them. Again, the unsoundness, stemmed from the approximation, makes the technique inappropriate for testing purposes. In contrast, our reported bugs are sound and this is ensured by keeping track of the events executed for obtaining a node state and checking the validity of the combination of these histories after a preliminary invariant violation report.

We also make use of the Cartesian product of independently explored node states to obtain the system states. Cartesian abstraction is essential here in our approach in order to create the system states and check (system-wide) invariants against them. In contrast, previous works benefited from the Cartesian abstraction by not creating system states; skipping the system states is possible since the invariants in multi-threaded programs are just thread-local assert statements and could be verified on a local state of a thread without having the rest of the system state. [5] Our local model checking approach employs the Cartesian abstraction in a different way: namely, to explore the system state space without exploring the global state space.

In [8], Cartesian Abstraction is used on top of boolean abstraction of threads to find race conditions in multi-threaded programs. After boolean abstraction, each thread is represented by a long boolean expression over global and local variables including an artificially added variable for line number. A race condition is also represented by a boolean expression over the line numbers in which the threads read and write the global variables. Race conditions are detected by taking conjunction of the thread boolean expressions with race conditions. Therefore, there is no need for system state creation. This approach cannot be applied on general system invariants that would express a relation between local variables of multiple threads. The approach applies a heuristic on the detected races to eliminate some of the false positives.

One could indeed generalize the Cartesian abstract interpretation presented in [14] to distributed systems, by using the network as the global object. However, the network would still be part of the model checking states, concatenated to the local states. In our approach, we exclude the network element from the model checking state and use only a shared network element.

Monotonic abstraction. Monotonic abstraction [16] of the network has been used in verification of security protocols since it accounts for the maximal knowledge learned by attacker. Dolev-Yao’s model [4] is one such model, in which the attacker remembers all messages that have been intercepted or overheard. The shared network object in our local model checking approach is essentially an application of a monotonic abstraction since the delivered messages are not removed from the network. The shared monotonic network is key to ensuring the completeness of the search by applying the generated messages also on future generated node states.

Online model checking. CrystalBall [20, 19] is a

[5]There is an ongoing research to convert a system-wide invariant to a set of thread-local assert statements, which has shown good results on small multi-threaded programs [3].

framework that implements the online model checking scheme. To be effective in practice, the online model checker must be fast enough to explore till a reasonable depth in the period between two restarts (typically a few seconds). CrystalBall uses a heuristic, namely *Consequence Prediction*, which prunes the local events of an already visited node state. As a heuristic, Consequence Prediction is incomplete and could, hence, miss some bugs due to false negatives. In contrast, our local model checking approach offers a complete search accompanied with proofs. Furthermore, complex distributed systems such as Paxos, often generate lots of network messages on which Consequence Prediction does not have any effect. For instance, in the used Paxos state spaces throughout this paper, we consider only the interleaving of the resulting network messages after some proposals. Therefore, Consequence Prediction, which does not prune the network messages, would not offer any improvement over B-DFS.

Stateful vs stateless search. To avoid loops created by exploring duplicate states, it is necessary to keep track of the states visited by a model checker. Obtaining a hash of the system state requires touching the whole state once, which can be nontrivial for large states. (Although stateless approaches [7] avoid this cost by not keeping track of traversed states, visiting duplicate states can make them very inefficient.) Thanks to Mace [9], a language upon which we implemented our tool, the relevant state of the protocol is specified by the developer and it is, hence, straightforward for MaceMC [10] to obtain its hash.

Partial order reduction. Since stateless approaches are not able to avoid loops, specific techniques are required to tackle the exponential explosion problem. Partial Order Reduction (POR) techniques [6] prune the state space of a concurrent system to avoid some unnecessary interleaving of events. The performance of B-DFS (used in our benchmarks) could potentially improve by implementing such technique. However, we expect the improvement would be marginal because of frequent changes into the global state; transmitting any message would change the network state and consequently the global state. Moreover, lots of redundancies avoided by POR-based techniques are already avoided by duplicate state detection in B-DFS. To the best of our knowledge, no study has compared the performance of stateful searches with POR-based techniques in distributed systems.

Furthermore, any application of a POR-based technique to model checking distributed systems would be incomplete since it would not account for system-wide invariants. For a set of local states, POR explores only one valid combination of them among all possible valid Cartesian products. It is useful in multi-threaded programs since the assert statements are defined on thread-local states and visiting a local state once in a combination with any other local states is enough. In contrast, we test the system against system-wide invariants such as Paxos main invariant, which could be held in one combination of node local states and violated in another.

7 Concluding Remarks

We introduce a novel, *local* approach to model checking distributed systems. Essentially, the underlying idea is to remove the network state from the global state when model checking, and focus on the remaining system state, which is the usual required part for invariant checking. The system state is itself built temporarily out of node states, and these are maintained separately. Although complete, the approach is not sound in the sense that some system states could be invalid, i.e., could not have been produced by an actual run of the system. We check the soundness of the system state, a posteriori, only if an invariant is violated.

By removing the network from the global states, our local model checking approach creates much less system states than in the global approach. In addition, and in contrast with the latter approach, in which visiting the system states is an inherent part of the exploration process, local approach separates the exploration of transitions from the actual creation of system states. This makes it possible to exploit the specificities of the user-specified invariants and a priori eliminate all system states on which these invariants cannot be violated.

Clearly, the state exponential explosion problem is not eliminated in our approach, and it indeed eventually manifests, especially because of invalid system states. Yet the problem is postponed and this makes our local approach an adequate match for online model checking that restarts the model checker periodically. Using online model checking augmented with our local approach, we found a previously reported bug in a traditional Paxos implementation, as well as a new bug in a recent variant of Paxos. Both bugs have been identified by focusing on a simple, arguably common case, namely the case with no contention for which distributed protocols are typically optimized and hence error-prone.

For future works, one can think of methods to automatically prune the system states according to a given invariant. In addition, the low memory consumption of our approach brings potentials for techniques that trade memory for CPU, gaining more speedup.

8 Acknowledgments

We thank Viktor Kuncak for invaluable comments. We are also thankful to our shepherd Alex Snoeren and the

anonymous reviewers for their excellent feedback.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*, 2001.

[2] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007.

[3] A. Cohen and K. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 2009.

[4] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on information theory*, 29(2), 1983.

[5] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Model Checking Software*. Springer, 2003.

[6] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[7] P. Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, 1997.

[8] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, 2003.

[9] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[10] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Com. of the ACM*, 21(7), 1978.

[12] L. Lamport. The part-time parliament. *TOCS*, 1998.

[13] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[14] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC*. Springer, 2006.

[15] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-Modular Counterexample-Guided Abstraction Refinement. In *SAS*, 2010.

[16] J. Mitchell. Multiset rewriting and security protocol analysis. In *Rewriting Techniques and Applications*, 2002.

[17] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 2002.

[18] M. Yabandeh, L. Franco, and R. Guerraoui. One Acceptor is Enough. Technical report, EPFL, 2010.

[19] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[20] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM TOCS*, 28(1), 2010.

[21] J. Yang and et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

A Appendix

We argue here that the soundness verification module of LMC, implemented by Procedure `isSequenceValid` of Figure 9, reports a given system state as valid if and only if the corresponding set of sequenced events is valid.

Theorem 1: If Procedure `isSequenceValid` returns `true` then the input system state is valid.

This is trivial since the procedure executes the sequence of events of each node in order and only if these events are enabled. Therefore, the resultant global state could also occur in a real run, i.e., it is valid.

Theorem 2: If the input set of sequenced events is valid, then Procedure `isSequenceValid` returns `true`.

We use here partial and total orders. Basically, the input sequences of events (one for each node state) define a partial order on the events. Considering all nodes, there might be more than one way to combine their event sequences, i.e., obtaining a total order between the events of all the nodes. A total order must also respect event causality. For example, an event in node n_1 that receives a message from node n_2 cannot be executed until the message is generated by an event in node n_2.

The partial order between the events is defined by the happens-before relation [11] as follows.

Definition: Let e_i^n be the ith event of node n. Relation happens-before, $\rightarrow$, between two events is defined by the following rules: (i) $e_i^n \rightarrow e_{i'}^n \Leftrightarrow i \le i'$, (ii) e_i^n generates the network message used by $e_{i'}^{n'} \Rightarrow e_i^n \rightarrow e_{i'}^{n'}$.

Notice that happens-before is antisymmetric, i.e., $e \rightarrow e', e' \rightarrow e \Rightarrow e = e'$. The total order of events executed by Procedure `isSequenceValid` could be obtained by sequencing the events in the time order that they are executed by the procedure. The following lemma expresses Theorem 2 using the notion of total order. The set of all total orders is denoted TO. Each total order $TO_i \in TO$ is a sequence of a subset of events that respects the partial orders. The projection of node n on total order TO_i is denoted TO_i^n.

Lemma 1: $\exists TO_i \in TO.\ \forall e.\ TO_i.e \notin TO \Rightarrow \neg\exists TO_j.\ \mid TO_j \mid > \mid TO_i \mid$.

Lemma 1 indicates that, if the size of a total order, obtained by any given algorithm, cannot be increased, then no algorithm could obtain a bigger total order. In other words, since Procedure `isSequenceValid` is finished by checking the possibility of increasing the size of the created total order, if the result is not a total order that includes all the input events, then there is no such total order. We prove Lemma 1 by contradiction. Assume there exists such TO_j. Then we have

$$\exists k \ge 1, n_m \in nodes(0 < m \le k).\ \mid TO_j^{n_m} \mid > \mid TO_i^{n_m} \mid \quad (1)$$

Let $e_f^{n_m} = TO_j^{n_m}[\mid TO_i^{n_m} \mid +1]$, where Operator $[i]$ gives the ith element of the sequence. In other words, $e_f^{n_m}$ denotes the first new event of node n_m in total order TO_j that is not included in total order TO_i. Observe that $e_f^{n_m}$ could not be a local event since it would then have been enabled in TO_i as well.

Let $s(e)$ be the event that has generated the message handled by network event e. Let $n(e)$ be the node of event e. We use sm to denote $s(e_f^{n_m})$. We have $sm \notin TO_i^{n(sm)}$, otherwise sm would have made $e_f^{n_m}$ enabled in TO_i. Therefore we have $e_f^{n(sm)} \to sm$, and consequently $e_f^{n(sm)} \to e_f^{n_m}$. This leads us to the following equation:

$$\forall n_m (0 < m \leq k).\, \exists n_{m'} (0 < m' \leq k).\\ e_f^{n'_m} \to e_f^{n_m}, e_f^{n_m} \not\to e_f^{n'_m}$$

This is obviously a contradiction since it implies a first element while at the same time demands a smaller element than that.