Efficient Model Checking of Fault-Tolerant Distributed Protocols

Péter Bokor†, Johannes Kinder†, Marco Serafini‡ and Neeraj Suri†
†Technische Universität Darmstadt, Germany
{pbokor,kinder,suri}@cs.tu-darmstadt.de
‡Yahoo! Research, Barcelona, Spain
serafini@yahoo-inc.com

***Abstract*—To aid the formal verification of fault-tolerant distributed protocols, we propose an approach that significantly reduces the costs of their model checking. These protocols often specify atomic, process-local events that consume a *set* of messages, change the state of a process, and send zero or more messages. We call such events *quorum transitions* and leverage them to optimize state exploration in two ways. First, we generate fewer states compared to models where quorum transitions are expressed by single-message transitions. Second, we refine transitions into a set of equivalent, finer-grained transitions that allow partial-order algorithms to achieve better reduction. We implement the *MP-Basset* model checker, which supports refined quorum transitions. We model check protocols representing core primitives of deployed reliable distributed systems, namely: Paxos consensus, regular storage, and Byzantine-tolerant multicast. We achieve up to 92% memory and 85% time reduction compared to model checking with standard unrefined single-message transitions.**

I. Introduction

Message-passing is a broadly used communication and programming paradigm in the design of reliable distributed systems [6], [32], [10], [28]. However, given the complexity resulting from concurrency and faults, message-passing systems are prone to subtle bugs [29], [23], [35], [37]. Consequently, a variety of formal techniques is advocated for ascertaining protocol correctness. A widely used formal technique for finding bugs or proving their absence is model checking [12], i.e., the automated and exhaustive exploration of the system's state space. The continuing main limitation of model checking is that the size of the full state space (and the corresponding time of exploration) is intractably large even for small systems, i.e., state space explosion.

An effective measure against state space explosion is abstraction [12], the separation of the conceptual, *protocol-level* state space from the low-level implementation (Figure 1). An implementation of protocol-level constructs defines a "one-to-many" mapping between protocol and implementation-level states and transitions. Once the correctness of the implementation (i.e., the mapping in general) is verified, a new protocol can be checked on the reduced protocol-level state space only [23]. If the implementation is not proven correct, the properties that hold at protocol-level are still valuable, e.g., to justify the conceptual design, but they do not transfer to the implementation.

Research supported in part by EC FP7 INDEXYS, Loewe TUD CASED, and DFG GKMM.

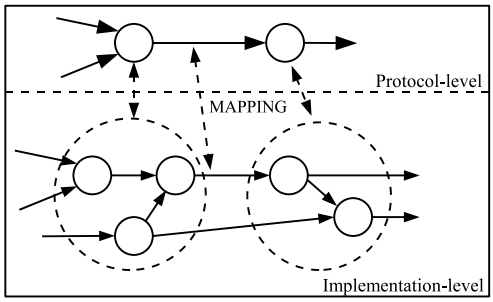

Figure 1. Illustration of protocol and implementation-level states.

Another generic state space reduction technique is *partial-order reduction* (POR) [12]. POR assumes that the system is defined in terms of transitions, i.e., atomic operations that change the state of the system. In message passing systems, for example, transitions are the sending or receiving of messages. The idea of POR is that the sequential execution of "independent" transitions leads to the same state irrespective of the relative order of the transitions and, often, the intermediate states do not impact the properties of interest. Therefore, it suffices to explore a representative execution order of such transitions.

Our overall goal is to minimize the size of protocol-level models and to perform space and time efficient state exploration of these models. A general pattern in the message-passing computation model is that a transition consumes multiple messages by a single execution. We call such transitions *quorum transitions*. Generally speaking, a quorum transition can process multiple messages, change the state of the process that executes it, and send new messages, in a single indivisible step. We show that quorum transitions not only enable a natural specification of a class of protocols, they also yield succinct protocol-level models and allow better POR performance.

As an example of quorum transitions, consider systems that guarantee reliability under the assumption that the number of faulty processes lies below a given threshold and

each correct (non-faulty) process executes an instance of the same replicated service [4], [6], [10], [38]. The threshold assumption implies that a set of messages from a large enough subset (or quorum) of processes contains at least one message from a correct process. Therefore, a common technique in such systems is that the execution of an event is triggered when a *set* of messages from a quorum (e.g., a majority) of processes is received.

Exploiting the characteristics of quorum transitions, we make the following contributions:

- We argue for *quorum transitions* to be modeled at the protocol-level. Otherwise, a quorum transition must be modeled via a sequence of transitions, each of them processing a *single* message, which generates a large number of (implementation-level) states (Section II). Although the implementation of quorum transitions can be complex, its correctness has to be verified only once.
- We observe that, maybe surprisingly, the definition of transitions, which depends on the programming style and language, can greatly affect the reduction achieved by POR. We introduce the concept of *transition refinement*, which exploits this observation to tune POR for better performance. Transition refinement splits a transition into multiple sub-transitions such that (a) the behavior of the system remains the same and (b) POR algorithms can detect more independent transitions. In particular, we define two transition refinement strategies: *quorum-split* and *reply-split* (Section III).
- We implement a POR-based model checker called *MP-Basset* that supports quorum transitions and the quorum and reply-split strategies. MP-Basset is built upon Basset, an existing model checker for actor programs, and upon its input language ActorFoundry [23]. Our protocol specification language is highly expressive, allowing the execution of arbitrary Java code that respects the message-passing computation model (Section IV).
- We evaluate MP-Basset based on diverse protocol examples with a range of fault semantics, namely (a) Paxos, a fundamental crash-tolerant consensus protocol [20], (b) a message-based regular storage implementation [3], and (c) a Byzantine-tolerant multicast protocol [26]. As the protocol properties are preserved by POR, our verification results are sound. While Paxos-similar protocols and storage implementations are already seeing deployment in various commercial settings [10], [38], [40], ready-to-use Byzantine tolerant libraries are also available [41]. Our experiments show that the proposed approach can be highly efficient with savings (verification memory and time) of *more than one order of magnitude* compared to models with unsplit single-message transitions. In addition, the proposed approach is also suitable for fast debugging especially of "subtle" bugs (Section V).

II. Message-Passing Models with Quorum Transitions

In this section, we briefly review the message-passing computation model [4]; we use simplified but equivalent semantics, which does not distinguish delivery and sending transitions and is better suited for model checking [8]. We then introduce MP, a Java-like language for specifying message-passing protocols. Finally, we show how quorum transitions affect the size of protocol-level models.

A. The Message-Passing Computation Model

Syntax. The system consists of n *processes* communicating via directed *channels*, which are (unordered) sets of *messages* from a set M. For processes i, j, $c_{i,j}$ represents a channel from process i to j and is called the outgoing channel of process i and incoming channel of j. Each process i assumes a set S_i of *local states*. Initially, every process i is in some initial state from S_i, and all channels are empty.

A *message passing protocol* is specified by defining a set T_i of *transitions* for each process i. Intuitively, a transition $t \in T_i$ can consume zero or more messages from the incoming channels of i, change the local state of i, and send multiple messages. If it can consume more than one (respectively, at most one) message, t is called a *quorum* (respectively, *single-message*) transition. t is associated with a predicate (or *guard*) g_t, whose truth value depends only on a set of incoming messages and i's local state. In addition, t is associated with $ls_t : S_i \times 2^M \rightarrow S_i$, the local state transition function of t. Intuitively, ls_t returns the new local state of process i depending on the current local state and a set of incoming messages. If the guard is true (i.e., t is *enabled*) in the current local state of i for a set of messages X in the incoming channels of i, the transition t can be *executed*. After executing t, all messages in X have been removed from the incoming channels of i, its local state may have been updated via ls_t, and messages may have been added (i.e., *sent*) to the incoming channels of other processes.

Note that transitions can be non-deterministic. For example, if transition t is enabled for messages $\{m_1\}$ and $\{m_2\}$, then t non-deterministically consumes either m_1 or m_2.

Semantics. The semantics of a message passing protocol is given by a *state graph*, i.e., pairs of states forming directed edges. Formally, a state graph (often referred to as Kripke structure [12]) is a tuple (S, S_0, Δ), where S is the set of states, S_0 is the set of initial states, and $\Delta \subseteq S \times S$ is a set of state pairs. A *state* $s \in S$ is a vector with all channel contents and the local state of each process. We denote the contents of channel $c_{i,j}$ and the local state of process i in s by $s(c_{i,j})$ and $s(i)$, respectively. Every transition t is a relation such that $t \subseteq S \times S$. For every $s, s' \in S$ and $t \in T_i$, $(s, s') \in t$ iff $g_t(X, s(i))$ is true for some subset X of the union of all incoming channels of i in s and s' is identical to s except

```
@guard
boolean READ_REPL(READ_REPL[] messages) {
  // guard: replies from a majority of N acceptors
  return messages.length==(Math.ceil((double)(N+1)/2));
}

@message
void READ_REPL(READ_REPL[] messages) {
  ... // select highest READ_REPL message among messages
  WRITE write=new WRITE(propNo, readReplHighest.val);
  for (ActorName w : acceptors)
   send(w, write);
}
```

Figure 2. MP syntax: Quorum transition in Paxos.

```
@message
void READ_REPL(READ_REPL message){
  cnt++;
  ... // stores READ_REPL if it is the highest seen
  if (cnt>=(Math.ceil((double)(N+1)/2))){
   cnt=0;
   WRITE write=new WRITE(propNo, readReplHighest.val);
   for (ActorName w : acceptors)
    send(w, write);
  }
}
```

Figure 3. MP syntax: Single-message Paxos transition.

for the following: (1) the messages in X are removed from the input channels of i, (2) $s'(i) = ls_t(s(i), X)$, and (3) zero or more messages are added to every outgoing channel of i. In this case, we say that t is *executed in* s *with* X and write $s \xrightarrow{t(X)} s'$. Now, $(s, s') \in \Delta$ iff there is a transition t such that $(s, s') \in t$.

For later use, we define $senders(X)$ to be $\{j \,|\, m \in X \wedge m \in s(c_{j,i})\}$, i.e., the set of processes that have sent a message in X. If there is $s \xrightarrow{t(X)} s'$ such that $|senders(X)| > 1$, then t is a quorum transition. Otherwise, t is a single-message transition.

Properties. Properties of a state graph can be defined using temporal logics [12]. These properties are interpreted over *paths*, i.e., sequences of states starting in an initial state such that each state is connected to the next state in the sequence. For example, a simple but useful class of properties are *invariants*, which define a state-local predicate that must hold in every state of any path. A *counterexample* is a path that violates the property. The property is true if there exist no counterexamples.

B. MP: A Language Implementation

We have implemented a language called *MP* (from message-passing) which allows specifying protocols in the message-passing computation model. MP extends the input language of the Basset model checker [23] with quorum transitions. MP inherits from Basset the ability to specify expressive guards and transitions in native Java. The only restrictions compared to full-fledged Java (in both Basset and MP-Basset) are imposed by the message-passing computation model, e.g., transitions cannot change the local state of other processes.

Figure 2 shows an example of a quorum transition from the Paxos consensus protocol [20] written in MP. In this transition a proposer defines its behavior on receiving a READ_REPL message from a quorum of acceptors.[1] By convention, the type of the message (here READ_REPL) must match with the name of the transition. The transition (annotated by @message) can only be executed if its guard (annotated by @guard) is true. In this example, the guard requires that the quorum contains a majority of the `N` acceptors. In the body of the transition the proposer sends the "highest" among the READ_REPL messages to all acceptors. Again, the name of the transition (and the corresponding guard) determines the type of messages this transition can consume. The argument of the transition (and guard) is an array of this message type, which stores the messages consumed by the transition. In accordance with the message-passing model, the order of elements in the array is arbitrary. It is guaranteed by the implementation that, given the current state of the process and the input array of messages, the guard function returns true before any execution of the transition.

A transition can change the local state of a process and send messages. Messages are sent using the `send` operation, which takes the recipient and the message as arguments. For example, the READ_REPL transition sends the same WRITE message to every acceptor (Figure 2).

C. MP without Quorum Transitions

Arguably, implementing quorum transitions is more complex than single-message transitions. We now show that the extra effort can pay off, given that the use of only single-message transitions can inflate the size of the state space.

Consider a language where only *single-message* transitions are allowed [23], i.e., a transition cannot consume multiple messages. In such a language the transition of Paxos shown in Figure 2 cannot be directly defined. We can describe a Paxos-like protocol by "simulating" READ_REPL via single-message transitions. Figure 3 shows such a transition: it receives a READ_REPL message, increments `cnt` to count the number of messages, and, if a majority of acceptors have sent a READ_REPL message, sends the WRITE message (with the same content as in Figure 2). In this case, the counter is reset, and the process of collecting READ_REPL messages starts over.

State space issues. A significant drawback of expressing quorum transitions with single-message transitions is that they can be interleaved with other transitions. For example, given the single-message READ_REPL and another transition of Paxos that can be co-enabled in states

[1]The original Paxos protocol is defined in terms of four phases 1a, 1b, 2a, and 2b, which we call READ, READ_REPL, WRITE, and ACCEPT, respectively. In the following discussion we assume basic familiarity with Paxos. As space constraints preclude us from fully detailing the protocol operations, we point the reader to [20], [21] and also pages 8-9.

where READ_REPL is executed, a model checker executes READ_REPL and this other transition in different orders in each of these states.

In general, consider a message-passing protocol P and transitions $t_1, ..., t_k$ that are enabled in some state s. Depending on the order of execution, the number of different states resulting from executing $t_1, ..., t_k$ is at most $k!k$.

Let t be a quorum transition that is enabled in s for a set X and $s \xrightarrow{t(X)} s'$ for some s'. Assume that P' is a message-passing protocol that is specified via single-message transitions only and s' is reachable from s in P'. The shortest path from s to s' in P' contains at least $|X| = l$ transitions, because any transition in P' can consume a single message. If we assume that these transitions are enabled in s, then the number of states is at most $(k + l)!(k + l)$, which is at least $(k + l)^2$ times more states than $k!k$ in P. This matches with the intuition that the larger the quorum the bigger the gain of using quorum transitions. We know that $k \leq |T|$ where $|T|$ is the number of all transitions in P. For the smallest meaningful instance and a "reasonable" specification of Paxos $(|T| + l)^2 = 169$. If we assume that $l \leq n$, i.e., t consumes at most one message from each process, then the state graph of P' can have $(|T| + n)^2$ times more states than P.

III. Transition Refinement

In this Section, after recalling the basics of partial-order reduction (Section III-A), we introduce and formalize transition refinement and prove that it preserves the soundness of POR (Section III-B). After the general definition, we introduce quorum-split, a message-passing example of transition refinement (Section III-C). Finally, we discuss reply-split, a useful and general quorum-split strategy (Section III-D).

As an early note, we emphasize that transition refinement preserves the underlying state graph of the system. It is essentially a "renaming" of the transitions that never affects the truth of properties. Note that replacing quorum transitions with single-message transitions does not have this property. This is because a single edge of the state graph representing a quorum transition can be divided into a path of lower-level single-message transitions.

A. Preliminary: Basic POR Terms

In practice, *transition systems* are used to specify how the system changes its state. Every transition t is a *set* of pairs (s, s') and makes the system proceed from state s to s'. Intuitively, a transition groups "similar" state changes of the system. Formally, a transition system is a tuple (S, S_0, T) where S (S_0) is the set of (initial) states and T is the set of transitions where $t \subseteq S \times S$ for every $t \in T$. Note that message-passing protocols define a transition system with $T = \cup_{i=1}^{n} T_i$ (see Section II-A).

Model checkers *generate* a state graph based on an input transition system in order to reason about the properties. Formally, $(s, s') \in \Delta$ in the generated state graph iff there is a transition t such that $(s, s') \in t$. The idea of reduction is to generate a *reduced* state graph that contains fewer edges than the original (unreduced) one.

POR is a reduction technique that is based on the observation that the execution of certain *independent* transitions leads to the same state irrespective of the order in which the transitions are executed (e.g., t_1 and t_2 in Figure 4(a)). In the Paxos example, t_1 and t_2 can be the READ_REPL transitions of different proposers. Therefore, it suffices to execute these transitions in one, representative order (e.g., $t_2 t_1$) if the states that are missed (s_1 in this example) are irrelevant for the truth of the property. In this example, the reduced state graph consists of $(s, s_2), (s_2, s_{12})$ and (s_2, s_3).

POR is defined via *global* terms through a sufficient set of paths that must be contained in the reduced state graph. In general, it is as hard to exactly determine this set as to explore the unreduced state graph. Practical POR algorithms therefore *over-approximate* this set of paths to respect property preservation. We now discuss two general classes of POR implementations that can benefit from transition refinement differently. Both classes implement POR by limiting (if possible) the set of enabled transitions that are executed in every reachable state. Such a sufficient subset of the set of all enabled transitions is called *stubborn set* [31].[2] In the example of Figure 4(a), $\{t_2\}$ is a stubborn set in s. Note that stubborn sets only preserve *deadlocks*, i.e., they guarantee that all states without enabled transitions will be explored. The preservation of general temporal logic properties (a class of properties called *stuttering equivalent*) requires additional constraints [31].

Static POR. The first class of implementations is called *static* (SPOR) because the stubborn set is computed in every state s when s is visited [15], [31], [12]. This means that the search is blocked for the time of the stubborn set computation. The main challenge of SPOR is to *guess* "future" paths, i.e., paths starting from s (note that these paths have not been explored yet). If there is a path among these future paths that is in the sufficient set defined by POR but that is not in the reduced state graph, then additional transitions must be added to the stubborn set. A common technique to guess future paths is the concept of *can-enabling* transitions [15]. For example, transition t_2 can enable t_3 in Figure 4(a). Therefore, we know that there is a path $t_2 t_3$ starting from s. As we will illustrate, transition refinement can affect how transitions can enable each other.

Dynamic POR. In *dynamic* POR (DPOR) the stubborn set in s is computed during the search [13]. In other words, the stubborn set is computed "on-the-fly" while the successors of s are visited. In fact, instead of guessing all future paths, DPOR *explores* some of these and defines the stubborn

[2]There are other notions of sufficient subsets such as ample [12], persistent [15], cartesian [16], monotonic [18], etc. The following discussion similarly applies to these alternate approaches.

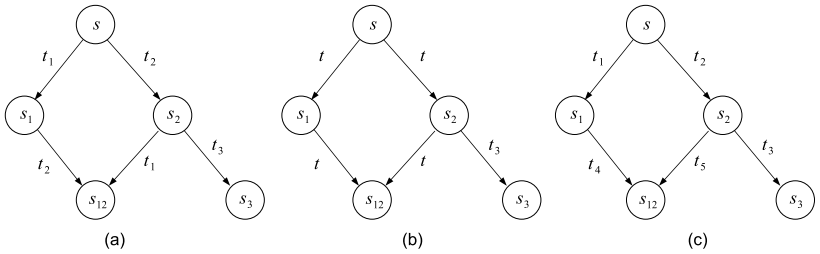

Figure 4. (a) Independent transitions t_1 and t_2. (b) Unrefined transition t: no reduction possible. (c) Caveat: transition refinement disables reduction.

sets later. An important limitation of DPOR is that it is unsound with *stateful* model checking, where the model checker maintains a set of visited states. Therefore, DPOR can only support stateless search, where the model checker cannot know if a state has been visited before and therefore its successor states are visited again.[3]

A commonality of SPOR and DPOR is that their performance can be greatly influenced by the choice of the *seed transition* [5] (also called start transition [31]), the first transition added to the stubborn set. Intuitively, the size of the stubborn set grows with the number of transitions that are dependent on the seed transition. In practice, heuristics are used to select seed transitions [5], [24].

B. General Transition Refinement

The ability of POR to achieve reduction strongly depends on the definition of transitions. Consider the transition system in Figure 4(b), which generates the *same* state graph as the system in Figure 4(a). Although the same reduction can be used in principle, POR is unable to realize a reduction in this case. This is because there is a single non-deterministic transition t enabled in s that must be executed in all possible ways. Therefore, to leverage POR, t can be refined into t_1 and t_2 as shown in Figure 4(a).

Transition refinement is a transformation of a transition system into another one such that the underlying state graph remains unchanged. Note that the following general definition does not require that the original transition set contains fewer transitions than the refined one.

Definition 1: Given transition systems TS and TS', TS is a *transition refinement* of TS' if TS and TS' generate the same state graph.

Transition refinement might affect the POR-reduced state graph but not the truth of any property. This is because POR is based on transitions, whereas properties are evaluated in the state graph. The property preservation result of POR directly implies that transition refinement also preserves temporal logic.

Theorem 1: Let TS_1 and TS_2 be two transition systems and φ a temporal logic property preserved by POR. Then, if TS_2 is a transition refinement of TS_1 and TS_1^R and TS_2^R denote the partial-order reductions of TS_1 and TS_2, then φ holds in TS_1^R iff it holds in TS_2^R.

Proof: Assume that φ holds in TS_1^R but not in TS_2^R. From the property preservation of POR we know that φ holds in state graph generated by TS_1. Furthermore, since TS_2 is a transition refinement of TS_1 we know that they generate the same state graphs. Therefore, φ also holds in state graph generated by TS_2. Again, from the property preservation of POR we know that φ holds in TS_2^R, a contradiction. The reverse can be proven similarly. ■

Benefit in practice. Transition refinement is not only theoretically useful but can also assist practical POR implementations in computing smaller stubborn sets. Firstly, the refinement of non-deterministic transitions into a set of deterministic ones can be beneficial (equally for SPOR and DPOR) if the refined transitions are independent. Unfortunately, a non-deterministic transition in practice often contains choices that are not independent. However, transition refinement can also help SPOR to guess future paths more accurately. Using the example of can-enabling transitions, a refined transition can enable fewer transitions. For an example, transition t in Figure 4(b) can enable t_3 because t_3 is disabled in s and, after the execution of t, enabled in s_2. Therefore, a static POR algorithm in s might falsely conclude that there is a future run $s \xrightarrow{t} s_1 \xrightarrow{t_3} s'$ (where s' denotes some state). However, if t is refined into t_1 and t_2 (Figure 4(a)), then t_2 can enable t_3 but t_1 *cannot* enable t_3. Therefore, the POR algorithm does not consider $s \xrightarrow{t_1} s_1 \xrightarrow{t_3} s'$ as a possible future run.

Caveat. Unregulated transition refinement can have undesired effects. As the runtime of POR algorithms increases with the number of all transitions [31], transition refinement can slow down the overall model checking time. Even worse, overly refined transition can even have a negative effect in terms of memory reduction. To see that, consider another transition system (Figure 4(c)) that also generates the same state graph. Here, every change of the system state is triggered by a new transition. Clearly, this transition system can be obtained via refining the transitions of one of the previous examples. Again, although reduction would be possible, POR sees no pair of transitions that could be executed in

[3]We remark that stateful optimizations of DPOR exists [34] but only at a price of increased memory and time overhead.

```
@guard
boolean READ_REPLij(READ_REPL[] messages) {
   return messages.length==((Math.ceil((double)(N+1)/2))
        && messages_are_sent_by_ij);
}

@message
boolean READ_REPLij(READ_REPL[] messages) {
  WRITE write=new WRITE(propNo, readReplHighest.val);
  for (ActorName w : acceptors)
   send(w, write);
}
```

Figure 5. Quorum-split READ_REPL with three acceptors.

different orders. Despite these warning scenarios, we will show that transition refinement is effective in practice.

C. Quorum-split: Refined Quorum Transitions

The idea of refining quorum transitions is to define a new transition for each set of processes from which the original quorum transition can consume a message. For example, consider the READ_REPL transition of Paxos in Figure 2. This transition is executed by a proposer process and it can only be enabled if a majority of all acceptor processes has sent a READ_REPL message to this proposer. If there are three acceptors 1, 2, and 3, then READ_REPL can be executed with messages from acceptors 1 and 2, 1 and 3, and 2 and 3. Therefore, the transition can be refined into three transitions READ_REPLij for every unordered pair i and j of acceptors (Figure 5). The transition READ_REPLij behaves exactly as READ_REPL except that it can only consume messages from acceptors i and j. We call this refinement strategy *quorum-split*.

Intuitively, READ_REPLij tells more about the possible state transitions of the system than the unsplit READ_REPL. In fact, we know that READ_REPLij only consumes messages from acceptors i and j. This additional information can be used by POR algorithms to achieve better reduction (we will show examples in Section III-D).

Formal definition. Transition refinement must not alter the system behavior. Thus, we define conditions under which a quorum-split can be performed and yields a valid transition refinement. We start by defining a special class of quorum transitions where the number of sender processes is fixed.

Definition 2: A transition t is an *exact quorum transition* with a threshold q_t iff $s \xrightarrow{t(X)} s'$ implies $|senders(X)| = q_t$ for all $s, s' \in S$ and sets of messages X.

Next, we formally define quorum-split.

Definition 3: Given a message-passing protocol P and an exact quorum transition t with threshold q_t, a *quorum-split* of P via t is an MP protocol P' derived from P by replacing t with transitions $t_1, t_2, \ldots, t_m$, for $m = \binom{n}{q_t}$, such that $s \xrightarrow{t_k(X)} s'$ iff $s \xrightarrow{t(X)} s' \wedge senders(X) = Q_k$, where Q_k is the k^{th} of the m sets of process IDs of size q_t.

Note that the definition of quorum-split also allows single-message transitions (with quorum-size one). In fact, every single-message transition is an exact quorum transition. We now state that quorum-split is a transition refinement.

```
@message
void READ(READ message) {
  highestPropNo=READ.propNo;
  READ_REPL read_repl=new READ_REPL(acceptedProp);
  send(message.sender,read_repl);
}
```

Figure 6. READ transition in Paxos.

Theorem 2: Let P be an MP protocol, TS the transition system of P, t an exact quorum transition in P with threshold q_t, P' a quorum-split of P via t, and TS' the transition system of P'. Then, TS' is a transition refinement of TS.

Proof: Let S and T be the set of states and transitions in TS and T' the set of transitions in TS'. Assume that TS and TS' generate different state graphs with sets of state pairs Δ and Δ', respectively. Assume that there is a $(s, s') \in \Delta$ such that $(s, s') \notin \Delta'$. Let $t' \in T$ be a transition such that $(s, s') \in t'$. Let X be a set of messages such that $s \xrightarrow{t'(X)} s'$. If $t' \neq t$, then $t' \in T'$ and thus $(s, s') \in \Delta'$, a contradiction. Since t is an exact quorum transition, it must be that $|senders(X)| = q_t$. P' is a quorum-split of P via t, so there is a $t_k \in T'$ such that $s \xrightarrow{t_k(X)} s'$ where $Q_k = senders(X)$, a contradiction. The reverse can be shown similarly. ■

Note that in principle every transition t can be split by adding a new transition t_Q for *every* subset Q of processes. However, this would mean adding 2^n extra transitions for *every* t (n is the number of all processes), which can worsen the time overhead of the POR algorithm.

Implementation. Quorum-splits can be performed automatically by conservatively analyzing the guards of quorum transitions. If the guard of a quorum transition t specifies an exact quorum size q_t (as in the example in Figure 2), then refining t for *each* set of processes of size q_t is guaranteed to be a transition refinement.

The number of new transitions can be further reduced by ruling out a process i that never sends messages consumed by t, i.e., if t is executed in a state with some X, then i cannot be in $senders(X)$. For example, learner processes in Paxos send no messages at all. Or, a proposer process sends no message to another proposer. The automatic detection of all possible $senders(X)$ sets can be done using simple patterns, otherwise we conservatively assume that i can be in such a set.

D. Splitting Reply Transitions

We discussed in Section III-B how transition refinement can benefit from POR. We now present some implications of quorum-split for static POR algorithms.

As refined transitions are dependent on fewer transitions than their unrefined counterpart, the SPOR algorithm can more accurately approximate future paths. In fact, a quorum

transition t *can be enabled* by (possibly) any process. However, if t_k is the quorum-split version of t, then t_k can be enabled only by transitions that are executed by processes in Q_k. For example, consider the transition READ_REPL in Figure 2. For acceptor processes 1 and 2 this transition is split into READ_REPL12 according to Figure 5. Now, READ_REPL can be enabled by every acceptor whereas READ_REPL12 can be enabled by transitions of acceptors 1 and 2 but not by acceptor 3.

Reply transition. We observe that the quorum-split of some special transitions can yield even more reduction. The idea is that the split version of these special transitions *can enable* fewer transitions than the original one. We observe that many protocols define *reply transitions* where a process receives one or more messages and sends messages only to the senders of these messages (e.g., acknowledgement).

For example, consider the READ transition of Paxos written in MP (Figure 6, guard is not depicted). Before a new value can be proposed in Paxos, the proposer process asks all acceptors about the values they have previously seen by sending a READ message to every acceptor. If an acceptor receives such a message, it executes a reply-transition to send a READ_REPL message to this proposer. Formally, we have the following.

Definition 4: Given an MP protocol and a process i, $t_i \in T_i$ is a *reply transition* if for all $s, s' \in S$ and for all subsets X of messages: $s \xrightarrow{t_i(X)} s'$ implies that $\{j \mid s(c_{i,j}) \subset s'(c_{i,j})\} = senders(X)$.

We call the quorum-split of reply-transitions *reply-split*. The additional benefit of reply-split is that the split transition t_k can only enable transitions executed by processes in Q_k. For example, the reply-split of READ for proposer 1 can only enable transitions of this proposer.

Implementation. For example, it is possible to automatically detect that a transition t is a single-message reply-transition if the recipient argument of any send operation appearing in t is `message.sender`, where `message` is the message consumed by t and `message.sender` is the sender of this message (Figure 6).

IV. The MP-Basset Model Checker

We implemented a tool called *MP-Basset* to model check protocol-level specifications written in MP. The architecture of MP-Basset is illustrated in Figure 7. The intuition is that the inclusion of a box denotes that it is "subsumed" by the outer boxes, e.g., JavaPathfinder (JPF) runs within the Java Virtual Machine (JVM). MP-Basset is built upon Basset [23], a model checker that supports a subset of ActorFoundry [19].[4] Therefore, protocols using only single-message transitions (the common subset of MP and ActorFoundry) are supported by both MP-Basset and Basset.

[4]Basset also supports a subset of Scala Actors, an actor programming language within Scala [25].

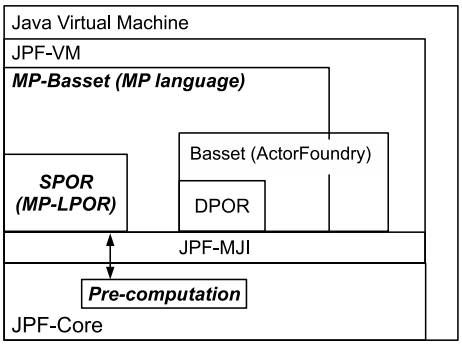

Figure 7. MP-Basset architecture illustration.

While Basset supports a wide range of DPOR algorithms, MP-Basset implements a new SPOR algorithm called MP-LPOR [9]. The novelty of MP-LPOR is two-fold: (1) it uses *pre-computation* to decrease the time overhead of SPOR and (2) it specifies *independent transitions* in the message-passing model. Despite these special characteristics, MP-LPOR is essentially an SPOR algorithm as discussed in Section III-A. Therefore, we expect that transition refinement can improve the reduction achieved by MP-LPOR.

In the following sections, we present the architecture of MP-Basset and key design issues. The complete source code and installation instructions of MP-Basset are available online [43].

A. Leveraging JPF & Basset

MP-Basset extends Basset, which runs as a Java application within the JPF model checker [39]. While JPF is written in Java itself and is executed by the JVM, the execution of target Java programs is "modeled" within JPF's model layer. We call this layer JPF-VM to refer to its functional similarity with the host JVM. Basset is an ordinary Java program that runs within the JPF-VM. JPF defines a gateway called Model Java Interface (JPF-MJI) between the modeled program and the core of JPF (JPF-Core). JPF-Core implements the search (model checking) functionalities of JPF such as the computation of concurrently enabled transitions in each state. By default, JPF assumes a fine-grained interleaving of Java threads. In order to prevent JPF from exploring unnecessary interleavings, Basset uses JPF-MJI (a) to impose the concurrency of the message-passing computation model, e.g., the execution of a transition is an atomic event, and (b) to implement different DPOR algorithms. In Basset, the model checking of an actor program written in ActorFoundry starts with creating the processes of the input actor program and sending an initial message. Then, JPF explores the state space of the program corresponding to interleavings defined by (a) and (b).

MP-Basset utilizes Basset's core architecture and implements quorum transitions by extending Basset's concept of "enabled message" into "enabled set of messages". More

precisely, set X of messages in the current state s is "enabled" if there is a transition t and a state s' such that $s \xrightarrow{t(X)} s'$. Note that computing these sets is time-expensive; in worst case they compose the powerset of all pending messages, which is an exponential overhead compared to the single-message case. Therefore, using quorum transitions can only reduce verification time if the space-reduction can compensate for the increased time overhead.

Example. Consider a state s where some process has three pending messages m_1, m_2 and m_3 in its input buffers. In order to find the enabled sets of messages, MP-Basset generates *every* set X in the powerset of $\{m_1, m_2, m_3\}$ to check if X is enabled for some transition t, i.e., the guard g_t is true for X in s. These are 2^3 sets compared to only three messages that need to be considered in a model of single-message transitions. Intuitively, this is the price we pay for the memory gain with quorum transitions as discussed in Section II-C.

B. Efficient Design of MP-Basset

We observe the following issues that are important with respect to the design of MP-Basset:

- *Executing code within MP-Basset.* Due to the indirection that MP-Basset runs in JPF-VM, any piece of code executed in MP-Basset is slower than in native Java. Fortunately, most message-passing protocols define simple code. However, other computation-intensive functionality such as stubborn set computation can be ineffective if executed within MP-Basset.
- *State size.* The larger the state of the modeled program the less the *throughput* of JPF, i.e., number of visited states per time unit. Reasons for this include that the time for hashing, storing, and state comparison increases with the size of the state.
- *MJI overhead.* Communication through JPF-MJI is expensive and tedious because JPF-MJI calls are implemented via Java methods that can only pass primitive type parameters. Therefore, the conversion (serialization and de-serialization) of complex types is required, which comes at the price of increased invocation time and additional code.

These issues necessitate tuning of how and where MP-Basset is implemented in the JPF architecture. It turned out to be efficient to compute enabled message sets entirely within MP-Basset (without JPF-MJI calls). However, the efficient design of MP-LPOR was more elaborate. It is possible for MP-LPOR to compute independent transitions before model checking as MP-LPOR uses a notion of independency that is unconditional, i.e., it is *not* a function of the system state. This information is queried (and not re-computed) repeatedly during the search. We perform pre-computation *outside* MP-Basset, i.e., through JPF-MJI calls, for two reasons. First, even if pre-computation is a one-time cost, it takes considerably longer when executed within the modeled program. Second, the pre-computed data is state unconditional, thus, it need not be stored in the state (an expensive measure as explained above).

An obvious approach to pre-compute and query independent transitions through JPF-MJI calls would be to serialize, pass, and de-serialize transitions as primitive types. To avoid this expensive and tedious task, we instantiate an exact *copy* of each transition within JPF-Core. As a result JPF-MJI calls can simply address transitions and the result of the queries (whether or not two transitions are independent) is passed through primitive boolean types. Note that this solution is only possible because the set of all transitions is fixed.

V. Evaluation

In this Section, we first briefly discuss the protocols we selected for analysis and how they are modeled in MP (Section V-A). Next we detail the verification results using Basset and MP-Basset (Section V-B).

A. Target Systems and Protocol-Level Abstractions

We use three widely used and representative fault-tolerant protocols to demonstrate the benefits of our approach. Each protocol assumes a threshold of the minimum number of correct processes. However, they define different fault models and also specify different properties. We now introduce these protocols and the properties that we analyzed with MP-Basset. Note that the goal of this evaluation is to evaluate the benefit of quorum transitions and transition refinement, and not a complete verification of these protocols.

- The Paxos protocol solves *consensus*, a fundamental primitive that can be used to implement state-machine replication [20]. Intuitively, consensus means that at most one value is "chosen", i.e., all processes agree on this value. Paxos solves consensus if a minority of processes can fail by crashing.
- Our second example is a consistent *multicast* protocol called Echo Multicast [26]. The *agreement* property of consistent multicast specifies that no two processes receive different messages. Echo Multicast implements agreement in a Byzantine environment [22] where up to one third of the processes can fail arbitrarily and the remaining processes are called honest.
- Our third example is *regular storage* protocol in the style of [3]. The objective of distributed storage is to reliably store data despite failures of the base (storing) objects. A regular storage guarantees that a read operation returns a value not older than the one written by the latest preceding write operation. The protocol assumes a crash-tolerant setting where a minority of all base objects might crash.

We remark that none of our target protocols assumes synchrony, i.e., an upper-bound of the worst-case message delivery time. Synchrony is required only for progress, e.g., a

value is eventually chosen in Paxos. Furthermore, messages can be delivered out-of-order.

Protocol settings. The protocols are parametric in the number of processes. In addition, processes can be of different type. In a given protocol setting we specify the number of processes of each type. Next we summarize the different process types in each protocol:

- *Paxos* defines proposer, acceptor, and learner processes. A proposer can initiate a consensus instance by proposing a value to be chosen. Acceptors store values proposed by proposers. Learners receive messages from acceptors to learn about proposals and output a chosen value. A Paxos setting (P,A,L) gives the number of proposers, acceptors, and learners, respectively. For example, Paxos (2,3,1) (as in Tables I-II) specifies two proposers, three acceptors, and a single learner.
- *Echo Multicast* defines initiator and receiver processes. In a setting (HR,HI,BR,BI), we define the number of honest receivers, initiators, Byzantine receivers and initiators, respectively.
- Every *storage* protocol defines writers, base objects, and readers. Since the selected protocol is a single-writer one, a setting (B,R) defines the number of base objects and readers, respectively.

Process faults. The above protocols tolerate two classes of faults, crash (Paxos and regular storage) and Byzantine (multicast). We do not explicitly model crash faults. This is because MP-Basset schedules processes in all possible ways and the effect of crash is implicitly modeled by scheduling other (non-crashed) processes first. In other words, crashed and correct processes taking no steps are equivalent. To model Byzantine faults, we specify processes that do not obey the protocol. We consider different attack strategies to challenge the multicast protocol. A complete model of Byzantine faults is beyond the scope of this paper.

We distinguish Byzantine processes whether they are initiators or receivers:

- A *Byzantine initiator* attempts to violate the agreement property by sending different messages to each of two groups of honest receivers.
- A *Byzantine receiver* sends invalid confirmations to an honest initiator and cooperates with a Byzantine initiator by confirming (signing) both of its messages.

Fault injection. For debugging purposes we also inject faults into (a) correct processes and (b) the specification of the protocols. In particular, we specify "Faulty Paxos", where learners do not compare the values received from the acceptors. In case of Echo Multicast and regular storage we utilize deliberately incorrect specifications. For example, in Echo Multicast we exceed the threshold of the number of maximum Byzantine processes ("wrong agreement"). For storage we require that a read operation that completes after a write has to return the value written by the write even if the two operations are concurrent ("wrong regularity").

B. Evaluation Strategy and Results.

We perform three experiments for each protocol setting:

- (Table I) We show that using quorum semantics reduces the size of the *overall* state space. We run our experiments with POR-optimization. We use two stubborn set-based POR implementations, a DPOR algorithm [13] implemented in Basset and an SPOR algorithm [9] (see Section IV).[5] As Basset does not support quorum transitions, we apply DPOR only for models with single-message transitions. Furthermore, as the safety property of regular storage (a form of linearizability) is not preserved by the DPOR implementations of Basset, we use unreduced search for verification in this case.
- (Table II) We show that transition refinement can *additionally* save model checking resources. As our split strategies refine transitions of the same process, the refined transitions are inter-dependent. Thus, transition refinement is ineffective with dynamic POR (see discussion in Section III-B) and the results are not depicted. In Table II we measure the performance of SPOR [9] for models splitting only reply transitions (reply-split), only non-reply quorum transitions (quorum-split), and all of these transitions (combined-split).
- (Across Tables I-II) We demonstrate that our approach can be used for efficient *debugging*. We show that finding the *first* bug[6] in faulty protocols or in protocols with wrong specification requires little resources.

Seed transitions. As explained in Section III-A, the performance of POR depends on the first transition in the stubborn set. We use a heuristic where transitions are preferred that either start a new instance of the protocol (e.g., READ transition in Paxos) or, if there is no such transition, do not terminate an ongoing instance (e.g., READ_REPL or WRITE transitions but not an ACCEPT transition). This heuristic shows good performance in our POR experiments. Intuitively, the execution of such a transition "delays" the decision of which instance is pursued at a given process. Surprisingly, this heuristic suggests the opposite of the *transaction* strategy proposed in [5]. We speculate that the difference lies in that our target protocols allow more concurrency than the cache coherence protocol analyzed in [5]. There, the processing of further client requests is blocked until the centralized cache controller (assumed to be fault-free) completes the ongoing instance of the protocol started by another client.

Note that our heuristic depends on the semantics of the protocol, which might be hard to automate. In fact, our seed

[5]Other DPOR algorithms in Basset such as [27] have property preservation guarantees other than stubborn sets. We chose [13] for a fair comparison with the stubborn-set based SPOR algorithm of MP-Basset.

[6]The bug first found by the model checker, after which the search is terminated and a counterexample is returned.

Table I
QUORUM SEMANTICS RESULTS.

			Baseline experiments				Our quorum results	
Protocol	**Property**	**Result**	**No quorum (DPOR[13])[1]**		**No quorum (SPOR[9])**		**Quorum[2] (SPOR[9])**	
			States	**Time**	**States**	**Time**	**States**	**Time**
Paxos (2,3,1)	Consensus	Verified	>16,087,468	>48h[4]	6,247,530	23h	***2,822,764***	***9h37m***
Faulty Paxos (2,3,1)	Consensus	CE[5]	***162***	***8s***	524	12s	279	10s
Echo Multicast (3,0,1,1)	Agreement	Verified	2911	41s	9222	2m22s	***652***	***12s***
Echo Multicast (2,1,0,1)	Agreement	Verified	***2010***	***27s***	9986	1m55s	2787	31s
Echo Multicast (2,1,2,1)	Wrong agreement	CE[5]	66	6s	66	9s	***48***	***6s***
Regular storage (3,1)	Regularity	Verified	2,358,345[3]	8h57m[3]	185,711	33m49s	***20,039***	***3m4s***
Regular storage (3,2)	Wrong regularity	CE[5]	286,410	1h1m	72,937	12m37s	***41,331***	***6m46s***

[1]Run by Basset (stateless search). [2]DPOR not supported. [3]Unreduced (stateful) search. [4]Time-out after 48h. [5]Counterexample found.

Table II
TRANSITION REFINEMENT IN ACTION.

					Our transition refinement results					
Protocol	**Property**	**Result**	**Quorum (SPOR[9])[1]**		**Reply-split[2,3]**		**Quorum-split[2,3]**		**Combined-split[2,3]**	
			States	**Time**	**States**	**Time**	**States**	**Time**	**States**	**Time**
Paxos (2,3,1)	Consensus	Verified	2,822,764	9h37m	1,087,486	3h47m	1,826,560	11h28m	***548,061***	***3h30m***
Faulty Paxos (2,3,1)	Consensus	CE[5]	279	10s	105	8s	279	10s	***105***	***8s***
Echo Multicast (3,0,1,1)	Agreement	Verified	652	12s	652	12s	232	12s	***232***	***12s***
Echo Multicast (2,1,0,1)	Agreement	Verified	2787	31s	1165	18s	2787	31s	***1165***	***18s***
Echo Multicast (3,1,1,1)	Agreement	Verified	12,023,663	>48h[4]	>10,472,557	>48h[4]	7,600,843	>48h[4]	***7,087,193***	***42h21m***
Echo Multicast (2,1,2,1)	Wrong agreement	CE[5]	***48***	***6s***	48	7s	48	9s	48	9s
Regular storage (3,1)	Regularity	Verified	20,039	3m4s	18,451	3m13s	18,451	4m31s	18,451	4m32s
Regular storage (3,2)	Wrong regularity	CE[5]	41,331	6m46s	***6,969***	***1m32s***	29,877	9m51s	6,987	2m34s

[1]Unsplit from Table I. [2]All protocols are modeled with quorum transitions. [3]Using the static POR algorithm from [9]. [4]Time-out after 48h. [5]Counterexample found.

priorities were set by hand. Other heuristics that require no user intervention are proposed in [5], [24]. For example, some heuristics are based on different characteristics of the pending messages [24]. A comprehensive comparison of how different heuristics perform for quorum transition-based protocols is beyond the scope of this paper.

Evaluation results. We use Basset and MP-Basset to run the experiments. All experiments ran on DETERlab machines [42] with Xeon processors and 4 GB of memory. The results are shown in Tables I-II. Our POR experiments utilize the "opposite transaction heuristic" explained above. The transaction heuristic resulted in very little reduction (not shown). The depicted protocol settings were selected such that they represent a meaningful instance of the protocol (e.g., enough processes to tolerate faults) and are feasible for model checking. Since the current version of MP-Basset does not support the automation of transition refinement, the split models were created by hand. For each protocol and setting, we highlight the best search strategy (if any) with bold italic numbers. For example, the quorum model of Paxos in Table I is the smallest and its model checking takes the shortest time. We observe the following trends:

- Using quorum transitions can reduce both *verification memory* and *time* (up to 89% and 91% for regular storage with SPOR) compared to the single-message case. First, the models with quorum transitions are smaller. Second, although the throughput of model checking quorum models is smaller (due to more complex semantics), the overall verification time is less because of state space reduction.
- Transition refinement can achieve *additional reduction* in terms of both memory and time (up to 81% and 64% for Paxos) compared to the unsplit case. Although the throughput falls with quorum-split (because split quorum transitions trigger a time-consuming optimization of the SPOR algorithm), it can achieve significant space reduction, which adds up to an overall time reduction (see combined-split of Paxos and multicast (3,1,1,1)).
- The proposed optimizations can also find bugs *fast* using *little memory*. If the bug is "deep" in the search space, we observe similar trends as for verification (see regular storage results with wrong property).

Behind the numbers. The above results offer interesting insights about the different search strategies. Firstly, the benefit of stateful over stateless search becomes significant with large state spaces. Otherwise, the stateless search can be faster because (1) it has no overhead of state comparison and (2) it revisits just a few states. In this case, the benefit of DPOR over SPOR can be exposed, e.g., for Faulty Paxos (Table I). In addition, the benefit of quorum models decreases with the small quorum size of Echo Multicast (2,1,0,1) (Table I).

If an optimization is ineffective in a particular fraction of the state space, then the search strategy with the least overhead achieves the best result, as can be seen for Echo Multicast (2,1,2,1) in Table II. In other cases, the split

strategies achieve no reduction in the entire state space. For example, reply-split is ineffective if there is a single initiator to which the receivers can reply (Echo Multicast (3,0,1,1)), or quorum-split makes no difference if the quorum contains all receivers (Echo Multicast (2,1,0,1)). Another example is regular storage (3,1), where reply-split and quorum-split are both ineffective. Note that the size of the reduced state space is slightly different (smaller) compared to the unsplit case. This is because split models define other (refined) transitions and the order in which these are executed can be different (depending on the scheduler of the model checker).

Note that, in general, there is no guarantee that a bug is found if POR does not preserve the property under verification. However, if the bug is contained also in the reduced state space, then the POR search tends to find the bug quicker as it explores "different" paths [35]. For example, we refer to the DPOR result of wrong regularity in Table I for which the unreduced search times out (not depicted).

VI. Related Work

Our formal model of message-passing systems adapted from [4] can be seen as an actor program [1]. Similarly, the proposed MP language shares commonalities with actor languages such as ActorFoundry [19]. For example, a concept similar to quorum transitions appears as joint transitions of actors [14]. However, these are proposed to make a specification language expressive and not to mitigate state space explosion. The actor model implements rich semantics, e.g., synchronization between actors or dynamic creation of actors. New transition refinement strategies can be devised for such richer semantics, if needed.

The protocol design and also verification can be simplified by making assumptions such as synchrony and fail-silent faults [30], [11] (note that none of our example protocols makes synchrony assumptions). The benefit of quorum transitions is possibly less significant in such systems as the possible interleavings of single-message transitions is constrained by synchronized events.

Existing work on POR concentrates on the definition and implementation of sound POR-conditions *given* the transitions of the system [15], [31], [12]. A concept similar to transition refinement is operation refinement in [15]. However, operation refinement is specific for a proof-of-concept modeling language, it is discussed informally, and its effect on the performance of POR implementations is not studied. In addition, no general applicable operation refinement is proposed nor its effect is evaluated in practical verification.

Promela, the input language of the SPIN model checker [17], supports message-passing and can be used, as any other general-purpose specification language, to implement the semantics of quorum transition (via atomic blocks). We consider our contribution, regarding quorum transitions, primarily on the modeling-side as substantiated by our MP-Basset experiments. However, we could have as well used other model checkers such as SPIN for these experiments. SPIN's POR algorithm is limited to *exclusive* reads and writes of FIFO channels. MP-Basset (and Basset) implement more general POR algorithms and can be extended with such FIFO-specific independencies.

Automated verification of specific message-passing fault-tolerant protocols is often done via model checking, e.g., [29], [28]. Other approaches target general classes of protocols. For example, MODIST is a POR-optimized directed testing tool for the analysis of unmodified distributed message-passing protocols [35]. As MODIST explores the state space of the real, unabstracted system, it is generally non-exhaustive due to state space explosion. A related approach is the network abstraction layer from [2], which is useful if the processes being model checked communicate with other external processes. Crystalball is a tool to debug and prevent failures through a combination of real execution and model checking [33]. Other work utilizes symmetry reduction for scalable model checking of message-passing protocols [7]. These and similar techniques are orthogonal to ours and can be used in combination.

VII. Conclusions and Future Work

We have devised, implemented, and evaluated a framework for efficient model checking of message-passing distributed protocols using quorum transitions. The framework consists of (a) using quorum transitions in protocol-level abstractions and (b) a new technique called transition refinement to improve the performance of certain partial-order reduction algorithms.

An open issue to pursue in our future work is whether the presented reduction techniques show similar trends in symbolic model checking. Such a comparison is especially motivated by the fact that certain POR methods, e.g. [18], are best suited for symbolic model checking.

Acknowledgement: We are deeply appreciative of Gul Agha, Steven Lauterburg and Rajesh Karmani from UIUC for making the sources of Basset available as well as for their continuing support of Basset.

References

[1] G. Agha, I. A. Mason, S. Smith, and C. Talcott, A foundation for Actor Computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[2] C. Artho, W. Leungwattanakit, M. Hagiya, Y. Tanabe. Efficient Model Checking of Networked Applications. *Objects, Models, Components and Patterns*, pp. 22–40, 2008.

[3] H. Attiya, A. Bar-Noy, D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.

[4] H. Attiya, J. Welch. *Distributed Computing*. Wiley, 2004.

[5] R. Bhattacharya, S. German, G. Gopalakrishnan. Exploiting Symmetry and Transactions for Partial Order Reduction of Rule Based Specifications. *SPIN*, pp. 252-270, 2006.

[6] K. P. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications.* Springer, 2005.

[7] P. Bokor, M. Serafini, N. Suri, H. Veith. Role-Based Symmetry Reduction of Fault-tolerant Distributed Protocols with Language Support. *ICFEM*, pp. 147–166, 2009.

[8] P. Bokor, M. Serafini, N. Suri. On Efficient Models for Model Checking Message-Passing Distributed Protocols. *FORTE*, pp. 216–223, 2010.

[9] P. Bokor, J. Kinder, M. Serafini, N. Suri. Local Partial-Order Reduction. Tech. Report, TR-TUD-DEEDS-11-01-2010, 2010.

[10] M. Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. *OSDI*, pp. 335-350, 2006.

[11] M. Chaouch-Saad, V. Charron-Bost, S. Merz. A Reduction Theorem for the Verification of Round-Based Distributed Algorithms. Proc. *Reachability Problems*, pp. 93–106, 2009.

[12] E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 2000.

[13] C. Flanagan, P. Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. *POPL*, pp. 110–121, 2005.

[14] S. Frolund, G. Agha. Abstracting Interactions Based on Message Sets. *Object-based Models and Languages for Concurrent Systems.*, pp. 107–124, 1995.

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

[16] G. Gueta, C. Flanagan, E. Yahav, M. Sagiv. Cartesian Partial-Order Reduction. *SPIN*, pp. 95–112, 2007.

[17] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley, 2004.

[18] V. Kahlon, C. Wang, A. Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. *CAV*, pp. 398–413, 2009.

[19] R. K. Karmani, A. Shali, G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. *Int. Conf. Principles and Practice of Programming in Java.*, pp. 11–20, 2009.

[20] L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, 1998.

[21] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[22] L. Lamport, R. Shostak, M. Pease. The Byzantine Generals Problem. *ACM Trans. Prog. Lang. and Sys.*, 4(3): 382–401, 1982.

[23] S. Lauterburg, M. Dotta, D. Marinov, G. Agha. A Framework for State-Space Exploration of Java-Based Actor Programs. *Automated Software Engineering*, pp. 468–479, 2009.

[24] S. Lauterburg, R.K. Karmani, D. Marinov, G. Agha. Evaluating Ordering Heuristics for Dynamic Partial-Order Reduction Techniques. *FASE*, pp. 308–322, 2010.

[25] M. Odersky, L. Spoon, B. Venners. *Programming in Scala*. Artima, 2008.

[26] M. K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. *CCS*, pp. 68–80, 1994.

[27] K. Sen, G. Agha. Automated Systematic Testing of Open Distributed Programs. *FASE*, pp. 339–356, 2006.

[28] M. Serafini et al. Application-Level Diagnostic and Membership Protocols for Generic Time-Triggered Systems. *IEEE Trans. on Dep. and Sec. Comp.*, 2011 (To appear).

[29] W. Steiner, J. Rushby, M. Sorea, H. Pfeifer. Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation. *DSN*, pp. 189–198, 2004.

[30] T. Tsuchiya, A. Schiper. Using Bounded Model Checking to Verify Consensus Algorithms. *DISC*, pp. 466–480, 2008.

[31] A. Valmari. The State Explosion Problem. *Petri Nets I: Basic Models*, pp. 429-528, 1998.

[32] P. Verissimo, L. Rodrigues. *Distributed Systems for System Architects.* Kluwer, 2001.

[33] M. Yabandeh, N. Knezevic, D. Kostic, V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *NSDI*, pp. 229–244,2009.

[34] Y. Yang, X. Chen, G. Gopalakrishnan, R. Kirby. Efficient Stateful Dynamic Partial Order Reduction. *SPIN*, pp. 288–305, 2008.

[35] J. Yang et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. *NSDI*, pp. 213–228, 2009.

[36] http://www.deeds.informatik.tu-darmstadt.de/peter/MP-Basset.pdf

[37] http://hadoop.apache.org/zookeeper/issue_tracking.html

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

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

[40] http://aws.amazon.com/s3/

[41] http://code.google.com/p/upright/

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

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

APPENDIX

I. MP-BASSET – USER GUIDE

Basic Model Structure. MP-Basset *models* are written in Java-syntax and consist of the following (copying Basset's actor programs):

- Process (or actor) *classes* are "types" of processes. For example, Paxos defines three classes of processes: proposers, acceptors, and learners. An actual process is an instantiation of its class. Every process class must extend the Actor class and define a constructor for instantiation. In additions, a process class can define *variables* and *transitions*. Variables are used to encode the local state of the process. Transitions specify the change of the local state.
- A *driver* is a configuration file describing the number and classes of processes of the system under test. Every process is created as an instance of its class and is launched by the driver. If a transition t requires no message to be executed, then "fake" messages called t are sent by the driver to the process executing t. For example, proposers in Paxos can be triggered from within the driver.
- The *specification* expresses the desired properties of the system. In the current version of MP-Basset, the specification is a set of Java assertions that can be defined within transitions. In other words, the specification restricts to invariants (or global predicates). If the assertion evaluates to false, then the search is terminated and a counterexample is given (if the `+fw.ce=1` flag is set).[7]

Models are stored under /jpf-actor/src/examples. An example model of the Paxos protocol can be found in the package paxos.actor (DriverMP.java, ProposerMP.java, AcceptorMP.java, and LearnerMP.java).

Defining Transitions. The syntax of transitions is explained in Table III. In MP-Basset, every transition t is annotated with `@message` and is named after the *type* of the messages that can be consumed by t. Formally, the type of the message corresponds to a subset of all messages and a transition can only process messages from the corresponding subset. A message consists of its type (or name) and a tuple of values. These values are passed as parameters to the transition consuming the message. For example, "READ(proposer1, 2)" in Paxos is a message of type READ carrying the parameters proposer1 (the name of the sender of the message) and 2 (proposal number). A transition might change the local state of the process and send messages to other processes. Sending a message follows the syntax `send(recipient,msgType,p1,...,pN)` with the recipient, the type, and the parameters of the message.

A special class of transitions is *quorum transitions*. Quorum transitions can consume more than one messages. A quorum transition msgType has an additional parameter of the Java type `Object[]` that is an array of messages each of them of type msgType. The order of the elements in this array is arbitrary, in accordance to the the MP semantics. Given a quorum transition of the form `msgType(p1,...,pN,Object[] messages)`, the parameters `p1`,...,`pN` define the format of the message so that a message can be cast from its `Object` type in the array. *By convention, `p1`,...,`pN` must not be read or written by a quorum transition!*

Every transition msgType can be associated with a *guard* (similarly to disabling "local synchronization guards" in ActorFoundry). Guards are annotated with `@Guard`. The guard is a Java function with boolean return value. A guard must not change the local state of the process nor send any message. If no guard is defined, any set containing messages of type msgType and sent to this process is accessible for transition msgType (as of MP semantics).

SPOR Support. Transitions can be annotated with `@LPORAnnotation` in order to ease SPOR and to enable our transition refinement strategies (Table IV). The methods of this annotation are summarized in Table IV. The last three methods relate to quorum and reply-split, respectively. In particular, the user can tune the initial transition heuristic of POR. A possible heuristic is the "opposite transaction heuristic" where the greater `priority()` the most likely that t does *not* finish a concurrent operation (e.g., a Paxos instance or a multicast). *In the current version of MP-Basset, the correctness of @LPORAnnotation must be verified by the user!*

We now review the most important property preservation features of the SPOR algorithm in MP-Basset:

- All deadlock states are preserved, i.e., the reduced state graph contains a deadlock state s iff s is in the unreduced state graph.
- If the unreduced state graph contains no cycles, no infinite paths, and all visible transitions with respect to a "state-predicate" P [12] are annotated (with `isVisible=true`), then *global reachability* is preserved, i.e., there is a state $s \in S$ in the reduced state graph such that $\neg P(s)$ iff there is $s' \in S$ in the unreduced state graph such that $\neg P(s')$.

Transition Refinement: Reply-split. Reply-split can be implemented by using `isReplyTransition()` (see Table IV). The current implementation only supports reply-split of single-message transitions. If a transition t consuming messages of type `msgType` is flagged by this annotation, then we assume that

- the transition is called `msgType_senderID` where

[7] Note that although the message-passing computation model "isolate" one process from another, Java in MP-Basset allows access to the state of a remote process. Be aware: this is a hack and side effects must be avoided!

Name	Syntax	Description
Single-message transition	`@message msgType(Type1 p1, ..., TypeN pN)`	`msgType`: type of incoming message `pi`: i^{th} parameter of the message
Quorum transition	`@message msgType(Type1 p1, ..., TypeN pN, Object[] msgSet)`	`msgSet`: incoming messages, each of type `msgType` `pi`: i^{th} parameter of `msgType`
Guard	`@Guard _msgType(Type1 p1, ..., TypeN pN, Object[] msgSet)`	Guard of `msgType`, returns boolean `msgSet`: only for quorum transitions

Table III
THE SYNTAX OF MP-BASSET TRANSITION `MSGTYPE`.

Name	Default return value	Description
messageIn()	`""`	The only type of message t can receive.
messageOut()	`""`	The only type of message t can send.
isReceiver()	`true`	t might process incoming messages.
recipients()	`Actor.class`	The class of processes that might send a message to this transition.
isSender()	`true`	t might send messages.
senders()	`Actor.class`	The class of processes whom this transition might send a message.
isStateSensitive()	`false`	t's guard reads the local state.
isWrite()	`false`	t writes the local state.
priority()	`0`	POR initial transition heuristic.
isQuorumTransition()	`false`	t is quorum transition.
quorumPeers()	{}	(quorum-split) t receives messages only from the listed processes.
isReplyTransition()	`false`	(reply-split) t is a reply transition.
quorumSize()	`0`	Size of the quorum if t is an exact quorum transition.
isVisible()	`false`	True if t is a visible transition.

Table IV
SUMMARY OF ANNOTATIONS OF TRANSITION t.

senderID is the ID of the (only) process that "communicates" with this transition,

- the process senderID sending messages of type `msgType` to the process executing t renames these messages to `msgType_senderID` before sending them.

Note that the `messageIn()` annotation of a reply transition does not have to be changed, i.e., it assumes `msgType`.

Transition Refinement: Quorum-split. A quorum-split transition is a special quorum transition where the set of processes (quorum peers) from which the transition consumes a message is fix. A quorum transition `msgType` can be "split" into a set of quorum transitions by specifying the quorum peers as an array of process IDs in `quorumPeers()`. Since Java does not allow identical method names with the same signature quorum-split transition names must have a prefix `msgType__` ("__" means double underscore).

Examples of both transition refinements techniques can be found in paxos.actor in classes QSplit*.java.

Running MP-Basset. MP-Basset is built upon Basset and can be run similarly using the additional flag `+fw.mp=1`.

MP-Basset implements a static POR algorithm called LPOR. LPOR can be run with NET-optimization (necessary enabling transitions). We call this algorithm LPOR-NET. Currently, the NET relation (required by LPOR-NET) is based on quorum transitions. Therefore, if there are no quorum transitions in the protocol, then LPOR and LPOR-NET achieve the same reduction. Since LPOR-NET has additional overhead compared to LPOR, we recommend to use LPOR in these cases. The flag `+fw.spor=` shall be set 1 for LPOR and 4 for LPOR-NET (the values 2 and 3 are reserved for pre-computed versions of LPOR).

Compatibility with Basset. The current implementation of MP-Basset is not fully compatible with Basset. For example, the dynamic POR (DPOR) algorithms implemented by Basset do not work with quorum transitions. Also, DPOR and LPOR cannot be combined. Protocols without quorum transitions and using the "guards" of ActorFoundry can be model checked with Basset. The different DPOR algorithms can be used by setting the value of `+fw.dpor` (see Basset documentation).