To Crash or Not To Crash: Efficient Modeling of Fail-Stop Faults

Habib Saissi[†], Péter Bokor[†], Marco Serafini[‡] and Neeraj Suri[†]

[†]Technische Universität Darmstadt, Germany
{pbokor,saissi,suri}@cs.tu-darmstadt.de
[‡]Yahoo! Research, Barcelona, Spain
serafini@yahoo-inc.com

Abstract. A commonly used approach in practical verification is to verify a simplified *model* of the system rather than the system itself, which would entail infeasible verification complexity. This paper introduces a model for efficient model checking of message-passing systems with *crash* faults. The key to the achieved efficiency is the intuition that the event of process crash can be omitted in the model as crashed processes can be mimicked by "slow" ones. We formally prove this intuition for a general class of systems and their specifications.
We evaluate model checking efficiency using two models, one where crash events are modeled as separate state transitions (explicit model) and another where these events are omitted (implicit model). Our experiments with widely-used and representative protocol examples show significant reductions of model checking memory and time when using the implicit model instead of the explicit one.

1 Introduction

Fault-tolerance is a general concept for building dependable systems. It guarantees that the system delivers correct service despite the presence of faults. Usually, the behavior and number of faults is restricted by a fault-model, which is a set of assumptions about the system and its environment. For example, the Paxos protocol [12] delivers consensus (the service) as long as faulty processes fail by crashing. Of course, this concept is valid only if the system that implements fault-tolerance is not faulty itself. For example, faulty implementations of Paxos fail to deliver consensus even if the fault-model is respected [14].

Model checking [10] can be used to automatically verify that fault-tolerance is implemented correctly. As the efficiency of model checking decreases with increasing state space sizes, its applicability is limited to small (sub-)systems or to simplified *models* of the real system. These represent different use-cases of model checking, which can contribute to the correctness of the system in different ways. For example, model checking a faithful model of the system can help fast prototyping and verifying conceptual designs.

In this paper, we propose a model for efficient model checking of *message-passing* systems with *crash* faults. These systems see more and more applications

given that (a) message-passing is an intuitive communication model [3, 1] and (b) the crash fault-model is a widely-applied abstraction in the current practice of reliable systems [7]. We expect every verification method to be *sound*, i.e., it does not miss bugs in the system. In order to ensure that model checking using our proposed model is sound, we compare it with a reference model of crash events, which we adapt from [3]: A crashed process stops receiving and processing messages in the future; if a process crashes during sending messages, only a subset of these messages might be sent. We call this reference model *explicit* because a crash event is modeled via an explicit state transition that drives the system from a state where the process is correct into a state where this process is crashed.

The explicit model yields large state spaces because crash events are interleaved, i.e., executed concurrently, with other events. In order to mitigate this state space explosion, we leverage the intuition that slow processes cannot be distinguished from crashed ones. We carefully investigate models of computation and communication to verify this intuition. For example, the intuition does not hold if a protocol inherently relies on crash events (e.g., using failure detectors [8]) or if the property under verification explicitly mentions crashes.

We call a model without crash events *implicit* because it can mimic the effect of crashes through the above intuition. We formally prove the soundness (and completeness) of the implicit model by showing that the truth of a general class of properties is indistinguishable in the explicit and implicit models.

The explicit model is exponentially larger than the implicit model. This exponential blow can be further worsened in practical model checking, where storing and comparing states (stateful optimization) is hard or even impossible. On the other hand, reductions (such as symmetry or partial order reductions [10]) can be used to prune the state space that actually needs to be explored. As a practical implication of our equivalence result, we model check representative message-passing protocols and measure the realized benefit of using the implicit model instead of the explicit one.

In summary, we make the following specific contributions:

- We define a formal model eligible for model checking systems where processes communicate via messages and might fail by crashing. First, we adopt a model from [3] (Section 2) and use it as a reference model to show the soundness of our proposed, simplified model. We call this reference model explicit because it explicitly models the event of crashing. We define a model, called implicit model, by simply removing crash events from the explicit model (Section 3).
- We formally prove that the implicit model preserves arbitrary LTL (Linear Temporal Logic) properties [10] of the explicit model if these properties do not depend on (i) the crash status of some process and (ii) the set of undelivered messages (Section 4). This class of properties is general and expressive enough to specify standard properties of fault-tolerant message-passing protocols, e.g., consensus, or variants of linearizability.

– We use MP-Basset [5], a SW model checker for message-passing systems, to model check the explicit and implicit models of representative crash-tolerant protocols (Section 5). Using the stateful and partial-order reduced optimizations of MP-Basset, our experiments show that model checking the explicit model results in state space explosion, even with relaxed forms of the crash-fault semantics. At the same time, the implicit model enables feasible model checking of the same protocol instances.

2 A Formal Model of Message-Passing Systems

We start by recalling a formal model of general message-passing systems (Section 2.1) and a suitable property language (Section 2.2). The precise semantics of the formalism is given via state graphs (Section 2.3).

2.1 Basic Message-Passing Model

Conceptually, we adopt the formal model of *message-passing system* (MP system) from [3]. Strictly speaking, the following model is taken from [4, 5], which was shown to be equivalent with the model in [3] but better suited for model checking.

An MP system consists of n *processes* that communicate via *messages*. Messages are sent between processes via *channels* according to a network topology. Every two processes i and j that are connected via a channel from i to j maintain a *buffer* $buf_{i,j}$, which is a set of messages for storing undelivered messages sent from process i to process j. The buffer $buf_{i,j}$ is called the *outgoing* (*incoming*) buffer of process i (j).

Every process i maintains a *local state* from a set Q_i. The *state* of the system is a tuple $s = (q_1, q_2, ..., q_n, b_1, ..., b_m)$, where $q_i \in Q_i$ for all $1 \leq i \leq n$ and $b_1, ..., b_m$ are the buffers of the system.

Transitions between the states of an MP system are modeled through *events*. The *execution* of an event denoted by $comp_i$ (short for computation) involves the following indivisible (atomic) change to the system: a (maybe empty) subset of the messages is removed from the union of all incoming buffers of i, the current local state q_i is changed to a (maybe the same) state from Q_i, and some (maybe zero) messages are added to the output buffers of i. Every event is associated with a *guard*, which is a predicate that depends only on a subset of the union of the incoming buffers and the local state of the process. The event can only be executed if the guard evaluates to true. In this case, we say the event is *enabled* in the current state. Otherwise, the event is *disabled*.

The set of all events is denoted as $COMP = \cup_{i=1}^{n} COMP_i$, where $COMP_i$ is the set of all events executed by i.

An *initial* state of the system is the state before the execution of any event. We assume that channels are empty in initial states.

2.2 Property Language: Temporal Logic

Properties that the system is expected to fulfill are interpreted over *runs*. A run of a message-passing system is a sequence of states $s_0, s_1, ...$ such that s_0 is an initial state and, for $i > 0$, the state s_i is the state resulting of the execution of an event $comp_i$ in s_{i-1} such that $comp_i$ is enabled in s_{i-1}. We call s_i a *reachable* state. By convention, initial states are also reachable.

The most simple properties specify single states. This requires the definition of a *labeling function*, which assigns *atomic propositions* from a set AP to each state. Formally, the labeling function is defined as $L : S \longrightarrow 2^{AP}$, where S denotes the set of all states. For example, atomic propositions combined with the usual Boolean connectives can be used to define *invariants*, a simple and expressive set of properties. A property is an invariant if it holds in every reachable state.

We adopt Linear Temporal Logic (LTL) [10] to specify *temporal* properties. In addition to atomic propositions and Boolean connectives, LTL defines *temporal operators*. For example, the operator **F** ("eventually" or "future") asserts that a property will hold in a state that is reachable (along a run) from the current state. As an example LTL formula, consider $\mathbf{F}p$. This formula expresses liveness, i.e., some atomic proposition p ("something good") must hold after the execution of an indefinite number of events.

2.3 Kripke Structure : Syntax & Semantics

We use the standard semantics of LTL [10]. As it is based on a *Kripke structure*, we associate MP systems with Kripke structures. A Kripke structure is a tuple (S, S_0, T, AP, L), where S is a set of states, $S_0 \subseteq S$ is a set of initial states, $T \subseteq S \times S$ is a set of transitions, AP is a set of atomic propositions, and L is a labeling function. Given an MP system M with (initial) state set S (S_0), and atomic propositions AP, and labeling function L, we associate with M the Kripke structure $M_{KS} = (S, S_0, T, AP, L)$, where $(s, s') \in T$ iff there is an event *comp* of the MP system such that *comp* is enabled in s and executing *comp* in s results in s'.

As a result, a run of the MP system M is a run (also called path [10]) of the Kripke structure M_{KS} and the standard semantics of LTL specifications can be applied. As this semantics assumes infinite runs, we define an additional event, called *dummy* event and denoted *dum*. The dummy event is enabled in every state and its execution does not alter the state of the system. Note that without the dummy event it is possible that no event is enabled in a state, resulting in finite runs.

3 MP Systems with Crash Faults

In this Section, we define MP systems where processes can *crash*. In the crash fault-model, a process can stop receiving, processing, and sending messages, and

it remains doing so forever. If the process crashes during the execution of an event, it executes the event as in the fault-free case except that it sends a subset of the messages that it was supposed to send [3].

Formally, given an MP system M, we define another MP system *crash* M by adding crash events. Note that we stay in the realm of MP systems (as defined in Section 2) without extending neither their syntax nor semantics.

The MP system *crash* M is identical with M except the following changes. In addition to a state from Q_i, the local state of process i (for every $1 \leq i \leq n$) contains a *crash flag*, which takes its values from $\{\bot, \top\}$. The value $\bot$ means that process i is crashed, otherwise the flag's value assumes $\top$. Formally, the local state of a process i is a tuple $q_i^c = (q_i, c_i)$, where $q_i \in Q_i$ and c_i is the crash flag of i. The set of events in *crash* M is $COMP^c = \cup_{i=1}^{n} COMP_i^c$, where, for every process i, $COMP_i^c = E_i \cup CE_i$ such that

- $E_i = \{comp' | comp \in COMP_i$ such that $comp'$ is identical with $comp$ and $comp'$ does not change $c_i\}$,
- $CE_i = \{comp^c | comp \in COMP_i$ such that $comp^c$ is identical with $comp$ and, when executed in a state, $comp^c$ sets $c_i = \bot$ and $MSG^c \subseteq MSG$ where MSG and MSG^c are the sets of messages sent by $comp$ and $comp^c\}$.

Intuitively, *crash* M inherits the events in E_i from the fault-free M, while CE_i contains the crash-faulty variants of these events. We call $comp^c$ in CE_i *crash-induced non-atomic send* if $MSG^c \subset MSG$ and $MSG^c \neq \emptyset$.

In addition, an event in *crash* M can only be executed by some process i if the crash flag c_i assumes $\top$. Formally, the guard of every event is extended with an additional condition (conjunct) defined as $c_i = \top$.[1]

4 The Equivalence of Explicit and Implicit Models

Given an MP system M, we call *crash* M an *explicit model* of crash faults. In contrast, M itself is an *implicit model* as no state transition directly models the crash of a process.

We first define a general equivalence between state graphs (Section 4.1), which we use to show as a special case that an explicit and the corresponding implicit models are equivalent (Section 4.2).

4.1 General Equivalence Basis

First, we define an equivalence relation between runs of Kripke structures. Intuitively, two runs are equivalent if they are of the same length and the i^{th} states in both runs are labeled the same.

Definition 1 *Given two Kripke structures* $M = (S, S_0, T, AP, L)$ *and* $M' = (S', S_0', T', AP, L')$*, a run* $\sigma = s_0, s_1, ...$ *in* M *is said to be* label-equivalent *with another run* $\sigma' = s_0', s_1', ...$ *in* M' *iff for every* $i = 0, 1, ...$*,* $L(s_i) = L'(s_i')$*. In this case, we write* $\sigma \approx_{AP} \sigma'$*.*

[1] Note that the guard of the dummy event (see Section 2.3) must not be changed.

The previous definition can be naturally generalized to the label-equivalence of two Kripke structures.

Definition 2 *Given two Kripke structures* $M = (S, S_0, T, AP, L)$ *and* $M' = (S', S'_0, T', AP, L')$*, they are said to be* label-equivalent *iff the following two conditions hold:*

- *For every run* σ *in* M*, there exists a run* σ' *in* M' *so that* $\sigma \approx_{AP} \sigma'$*.*
- *For every run* σ' *in* M'*, there exists a run* σ *in* M *so that* $\sigma \approx_{AP} \sigma'$*.*

The next corollary follows from the above definitions and the semantics of LTL [10]. It says that the truth of an arbitrary LTL property is indistinguishable in label-equivalent Kripke structures. The notation $M \models \phi$ means that the (LTL) formula ϕ holds for every run of the (Kripke structure) model M.

Corollary 1 *[10] Given two label-equivalent Kripke structures* M *and* M' *and a LTL formula* ϕ*, the following holds:*

$$M \models \phi \text{ iff } M' \models \phi \ .$$

Proof. The $\Rightarrow$ direction: Assume that $M' \not\models \phi$. Therefore, there must be a run σ' in M' such that $\sigma' \not\models \phi$. Since M and M' are label-equivalent, there is a run σ in M such that σ and σ' are label-equivalent. By definition, σ and σ' are of the same length and the corresponding states are labeled the same. This implies that $\sigma \not\models \phi$ [10], a contradiction.

The reverse direction can be similarly proven.

4.2 The Equivalence Theorem

In this section, we prove the label-equivalence between an MP system M and its crash-augmented version *crash* M. More precisely, we show label-equivalence between their Kripke structure counterparts.

To this end, we first define a special labeling function for MP systems, which is independent of the crashed status of processes and undelivered messages.

Definition 3 *Given an MP system* M*, a set of atomic propositions* AP*, the Kripke structure* (S, S_0, T, AP, L) *associated with* M*, and the Kripke structure* (S', S'_0, T', AP, L') *associated with* crash M*,* L *and* L' *are* crash/buffer -independent*, if for all* $s = (q_1, ..., q_n, b_1, ..., b_m) \in S$ *and* $s' = ((q_1, c_1), ..., (q_n, c_n), b'_1, ..., b'_m)) \in S'$*,* $L(s) = L'(s')$*.*

The following theorem states our main result, which together with Corollary 1 imply that an LTL formula holds for M iff it holds for *crash* M.

Theorem 1 *Given an MP system* M*, a set of atomic propositions* AP*, the Kripke structure* $M_{KS} = (S, S_0, T, AP, L)$ *associated with* M*, and the Kripke structure* $M^c_{KS} = (S', S'_0, T', AP, L')$ *associated with* crash M*, if* L *and* L' *are crash/buffer-independent, then* M_{KS} *and* M^c_{KS} *are label-equivalent.*

Proof. Let $\sigma = s_0, s_1, ...$ and $\sigma' = s'_0, s'_1, ...$ are runs of M_{KS} and M^c_{KS}, respectively. The proof is by induction on the length of the prefixes of σ and σ'. Given a prefix of σ (and σ'), we construct a prefix of a run in M^c_{KS} (in M_{KS}) such that label-equivalence holds for these prefixes. Then, label-equivalence between σ (and σ') and the constructed run follows by induction.

The $\Rightarrow$ direction. Intuitively, we construct a run σ' in *crash* M such that the events executed in M and *crash* M are the same. In other words, *crash* M simulates the non-faulty M.

Consider the prefix s_0, s_1 of σ as the base case. We know that $s_0 = (q_1, ..., q_m, b_1, ..., b_m) \in S_0$. In our construction, let $s'_0 = ((q'_1, c_1), ..., (q'_n, c_n), b'_1, ..., b'_m)$ be from S'_0 such that s_0 and s'_0 are *matching*, i.e., $q_1 = q'_1, ..., q_n = q'_n$ and $b_1 = b'_1, ..., b_m = b'_m$. Now, let *comp* be an event in M such that executing it in s_0 results in s_1. If *comp* is a dummy event, then we construct s'_1 such that $s'_1 = s'_0$. Otherwise, if *comp* is executed by process i, then let *comp*$'$ be a *matching event* with *comp*, i.e., *comp*$'$ is the event corresponding to *comp* as defined by E_i. Given that $s_1 = (qq_1, ..., qq_n, bb_1, ..., bb_m)$, let in our construction $s'_1 = ((qq_1, cc_1), ..., (qq_n, cc_n), bb_1, ..., bb_m)$ be the state resulting from the execution of *comp*$'$ in s'_0. Note that *comp*$'$ is enabled in s'_0 because $s'_0 \in S_0$ and so $c_i = \top$. Furthermore, since *comp* and *comp*$'$ are matching, there is an execution of *comp*$'$ satisfying that s_1 and s'_1 are matching over the local states of processes and the content of buffers. Since L and L' are crash/buffer-independent, we have that $L(s_0) = L'(s'_0)$ and $L(s_1) = L'(s'_1)$.

In the induction step, assume that there is a run in *crash* M with prefix $s'_0, s'_1, ..., s'_k$ that is label-equivalent with $s_0, s_1, ..., s_k$. Let s_k be the tuple $(q_1, ..., q_m, b_1, ..., b_m)$. By construction, we have that $s'_k = ((q_1, c_1), ..., (q_n, c_n), b_1, ..., b_m)$. The construction of s'_{k+1} is analogous to that of s'_1. Note that *comp*$'$ is enabled because $c_i = \top$ for all $1 \leq i \leq n$. This is because our construction selects *comp*$'$ from E_i, thus, the value of c_i remains unchanged.

The $\Leftarrow$ direction. Intuitively, we construct a run σ in M such that crashing and non-crashing events are replaced by there matching counterparts in M, i.e, non-faulty events that receive/send the same messages and perform the same local state transition.

Let s'_0, s'_1 be a prefix of σ' where $s'_0 = ((q_1, c_1), ..., (q_n, c_n), b_1, ..., b_m)$. Then, let $s_0 = (q_1, ..., q_n, b_1, ..., b_m)$ from S_0. We know that such s_0 exists by construction of *crash* M. Since L and L' are crash/buffer-independent, we have that $L'(s'_0) = L(s_0)$.

Now, let e be the event in *crash* M that results in s'_1 when executed s'_0. Similarly to the first part of the proof ($\Rightarrow$ direction), in case $e =$ *dum* and $e =$ *comp*$' \in E_i$, the corresponding event in M is *dum* and the matching *comp* that is used to construct s_1 when executed in s_0. If $e = comp^c \in CE_i$, then consider the matching event *comp* as defined by CE_i. Let s'_1 be the tuple $((qq_1, cc_1), ..., (qq_n, cc_n), bb_1, ..., bb_m)$. We construct $s_1 = (qq_1, ..., qq_n, bb'_1, ..., bb'_m)$ as the state resulting from the execution of *comp* in s_0. Note that the content of the buffers may be different, more precisely $bb_j \subseteq bb'_j$ for all $1 \leq j \leq m$, if $comp^c$

is a crash-induced non-atomic send. As L and L' are crash/buffer-independent, $L'(s'_1) = L(s_1)$ holds.

By the induction hypothesis, there is a run in M with prefix $s_0, ..., s_k$ that is label-equivalent with $s'_0, ..., s'_k$. By construction, given $s'_k = ((q_1, c_1), ..., (q_n, c_n), b_1, ..., b_m)$, we have that $s_k = (q_1, ..., q_n, b'_1, ..., b'_m)$ and $b_j \subseteq b'_j$ for all $1 \leq j \leq m$. The construction of s_{k+1} is similar to that of s_1. Note that the matching *comp* can always be executed in s_k because the buffers in s_k contains at least those messages in s'_k.

4.3 Implications of Different Buffer Models

Our model of MP systems from Section 2 assumes that every buffer is an *infinite* set of messages. As some applications might require modeling finite buffers, we now discuss how modeling finite buffers affects our equivalence result.

We consider two models of finite buffers. In the first model, a (non-dummy) event can only be enabled if *all* buffers that this event sends messages to have the capacity of delivering (storing) these messages. The proof of Theorem 1 can be easily modified using this model of finite buffers.

In the second model, a message m in a full buffer *buff* can be *overwritten* by a message m' that is sent via this buffer. This means that m will be lost and replaced by m' in *buff*. It turns out that the construction used in the proof of Theorem 1 does not work with this model of finite buffers. The problem is that these non-atomic send events can result in overwriting a *subset* of those messages that are overwritten in the non-faulty model. This might result in a process entering a local state that is unreachable for this process in the non-faulty model, thus, invalidating the equivalence result. Note that in our model with infinite buffers *all* messages that are available in *crash* M are also available in M, a property that does not hold using the second model of finite buffers.

5 Experiments: Model Checking Efficiency with Explicit and Implicit Models

Evaluation objective. Given an MP system with n processes, the explicit model is at least 2^n times larger than the implicit model. This is because for every every state in the implicit model there are 2^n corresponding states in the explicit model where every process can be crashed or alive. The exponential blow is further worsened by non-atomic sends. For simplicity, we consider a relaxed crash-model semantics where non-atomic sends are assumed not to happen.

Ideally, the size of a model is proportional with model checking memory and time. However, practical model checking can distort this trend. Firstly, a model checker might visit the successors of a state many times if this state is reachable through multiple runs. The reason for this is that storing and comparing states in stateful model checking [10] might be inefficient or even impossible given powerful specification languages [11]. Secondly, different reduction techniques [10]

enable sound verification by exploring only a fraction of the model. Depending on the system, one model can be better "reducable" than another.

Focusing on stateful and partial-order reduced [10] optimizations of model checking, our objective is to show that model checking the explicit model is exponentially more expensive (in terms of memory and time) than the implicit model. This would demonstrate the practicability of our equivalence result.

Example protocols. We consider two representative crash-tolerant protocols, i.e., they satisfy their specifications under the assumption that processes can only fail by crashing:

1. The Paxos protocol solves *consensus*, a fundamental primitive that can be used to implement state-machine replication [12]. Intuitively, consensus means that at most one value is "chosen", i.e., all processes agree on this value.
2. Our second example is *regular storage* protocol in the style of [2]. 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.

For debugging purposes, we inject faults into (a) correct processes and (b) the specification of the protocols and show that the model checker is able to find the bugs. In particular, we specify two faulty versions of Paxos, namely "Faulty Paxos" and "Faulty Paxos 2". 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"). More details and the source of these models can be found at [16].

Setup: tools, resources, and metrics. We use the MP-Basset model checker [5, 16] to conduct our experiments. MP-Basset is a model checker for message-passing systems implementing the following optimizations: stateful model checking via Java Pathfinder [15] and highly customizable static partial-order reduction [6]. In our experiments, partial-order reduction is customized for message-passing (read more details in Section 6). The experiments are run in the DETER testbed [17] on 2GHz Xeon machines with 4GB memory.

We measure model checking memory (the number of visited states) and time for each experiment. In the explicit model, we gradually add $1, 2, ...$ crash-prone processes and run a new experiment. In case of faulty protocols and specifications, the model checker stops at finding the first counterexample. Therefore, these searches are not exhaustive.

The model checker returns OK if the specification holds for the protocol. Otherwise, a counterexample (CE), i.e., a run violating the specification, is given. We add up to three (two) crashes for the OK (CE) cases. The reason of running more experiments without bugs is to measure how adding new crash-faulty processes affects the size of the explicit model.

Protocol (# processes)	Spec.	Result	Explicit model			Implicit model	
			# crashes	States	Time	States	Time
Paxos (6)	Safety	OK	1	1,541,622	9h50m	**548,961**	**3h18m**
			2	4,216,431	27h44m		
			3	11,843,034	83h		
Faulty Paxos (6)	Safety	CE	1	14,785	4m49s	**3,415**	**1m40s**
			2	33,598	10m53s		
Faulty Paxos 2 (7)	Safety	CE	1	1,442,262	12h20m	**173,414**	**1h28m**
			2	3,047,842	25h40m		
Register (5)	Regularity	OK	1	56,508	16m36s	**18,451**	**4m32s**
			2	128,697	40m50		
			3	301,562	1h40m		
Register (5)	Wrong regularity	CE	1	9,781	2m45	3,497	55s
			2	**1,213**	**29s**		
Register (6)	Wrong regularity	CE	1	18,272	7m	**6,987**	**2m32s**
			2	42,506	15m		

Table 1: Stateful and partial-order reduced state space exploration results with implicit and explicit models using the MP-Basset model checker.

Experimental results. Our results are shown in Table 1. We model check only meaningful instances of both protocols, i.e., at most one fault is tolerated. For each experiment, we emphasize the best result (least model checking memory and time) using bold text. We observe the following trends:

- The implicit model is more efficient than the explicit one in *all except one* experiments. In this one experiment the model checker finds the bug slightly faster using the explicit model. As the CE experiments are non-exhaustive, finding counterexamples quickly depends on how the model checker schedules events. In MP-Basset, the additional (crash) events in the explicit model affect this scheduling, as shown by our experiments. Heuristics can be used to "guide" the model checker towards the bug [13].
- Model checking memory and time of the explicit model is *exponential* in the number of crash-faulty processes compared to the implicit model. This trend is also depicted in Figure 1, where we show the number of states in the explicit model as a function of n, where n is the number of crashes. Note that the number of states grows even faster than 2^n. Again, this ideal formula is biased by the imperfect stateful optimization and partial-order reduction.

6 Related Work

Our reduction from the explicit to the implicit model allows sound and also complete verification of the specified class of properties (LTL with crash/buffer-independent labeling function). Although other reduction techniques such as symmetry or partial-order reductions [10] apply for a more general class of systems, they require manual intervention of the user. These techniques are orthogonal to the explicit/implicit model of crashes and can be applied for further reductions of both models.

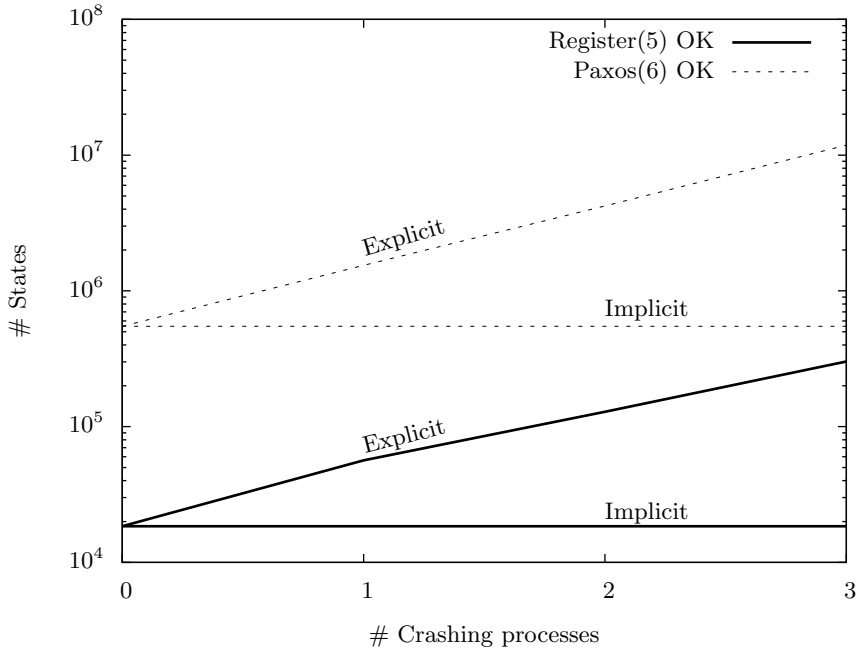

Fig. 1: The size of the explicit model as a function of the number of crashes.

In our experiments, we use partial-order reduction of both the explicit and implicit models. We apply this reduction for message-passing systems as proposed in [6]. For the explicit model, we extend the partial-order reduction with crash events and use the flexible and intuitive framework of [6] to prove the soundness of the reduction. Intuitively, we define events that are "non-interfering" with crash events, which is key to partial-order reduction. For example, a crash event e^c is non-interfering with every other event e in the sense that if e is disabled in a state, then it will remain so after the execution of e^c.

Another related reduction approach is [9], which reduces from a fine-grained model to a stuttering-equivalent coarse-grained model to allow efficient model checking. Although the underlying model is message-passing with crash faults, it assumes (synchronous) round-based communication and crashed-faults are expressed through so called Heard-Of sets. Our equivalence result does not directly apply under these assumptions but, instead, under the general model of [3].

7 Conclusion

We have defined a formal model that allows efficient model checking of message-passing systems with crash faults. The proposed model is a reduction from a detailed (and obviously sound) model and it accounts for sound verification

for a certain class of properties. Natural extensions of our approach include reductions for other fault-models (such as non-silent malicious faults) or proving the equivalence with respect to more general temporal logics (such as branching-time logics).

We see the strength of our contribution on the practical side. Our equivalence result formally verifies the natural intuition that crash events need not be modeled explicitly. Therefore, system designers can use this as a formal argument (rather than as "reasonable simplification") in the development and certification process. These are small but important steps towards scalable verification of real systems.

References

1. G. Agha, I.A. Mason, S. Smith, C. Talcott. A Foundation for Actor Computation. *Journal of Functional Programming*, 7(1): 1–72, 1997.
2. H. Attiya, A. Bar-Noy, D. Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42(1):124–142, 1995.
3. H. Attiya, J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing, 2004.
4. P. Bokor, M. Serafini, N. Suri, On Efficient Models for Model Checking Message-Passing Distributed Protocols, IFIP Int. Conf. on Formal Techniques for Distributed Systems (FMOODS & FORTE), pages 216-223, 2010.
5. P. Bokor, J. Kinder, M. Serafini, N. Suri. Efficient Model Checking of Fault-Tolerant Distributed Protocols. *DSN-DCCS*, 2011, To appear.
6. P. Bokor, J. Kinder, M. Serafini, N. Suri. Supporting Domain-Specific State Space Reductions through Local Partial-Order Reduction. Technische Universität Darmstadt, Technical Report, 2011.
7. K. Birman. *Reliable Distributed Systems: Technologies, Web Services, and Applications*, Springer, 2005.
8. T.D. Chandra, S. Toueg Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM*, 43(2):225–267, 1996.
9. 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.
10. E. Clarke, O. Grumberg, D. Peled. *Model Checking*, MIT Press, 2000.
11. P. Godefroid. Model checking for programming languages using VeriSoft. *POPL*, pp. 174–186, 1997.
12. L. Lamport. The Part-time Parliament. *ACM Trans. Comp. Sys.*, 16(2):133–169, 1998.
13. M. Talupur, H. Han. Biased Model Checking Using Flows. *TACAS*, pp. 239–253, 2011.
14. J. Yang et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. *NSDI*, pp. 213–228, 2009.
15. http://babelfish.arc.nasa.gov/trac/jpf
16. http://www.deeds.informatik.tu-darmstadt.de/peter/mp-basset/
17. http://www.isi.deterlab.net/