On Efficient Models for Model Checking Message-Passing Distributed Protocols ⋆

Péter Bokor, Marco Serafini and Neeraj Suri

Technische Universität Darmstadt, Germany
{pbokor,marco,suri}@cs.tu-darmstadt.de

Abstract. The complexity of distributed algorithms, such as state machine replication, motivates the use of formal methods to assist correctness verification. The design of the formal model of an algorithm directly affects the efficiency of the analysis. Therefore, it is desirable that this model does not add "unnecessary" complexity to the analysis. In this paper, we consider a general message-passing (MP) model of distributed algorithms and compare different ways of modeling the message traffic. We prove that the different MP models are *equivalent* with respect to the common properties of distributed algorithms. Therefore, one can select the model which is best suited for the applied verification technique.
We consider MP models which differ regarding whether (1) the event of message delivery can be interleaved with other events and (2) a computation event must consume all messages that have been delivered after the last computation event of the same process. For generalized MP distributed protocols and especially focusing on fault-tolerance, we show that our proposed model (without interleaved delivery events and with relaxed semantics of computation events) is significantly more efficient for explicit state model checking. For example, the model size of the Paxos algorithm is $1/13^{th}$ that of existing equivalent MP models.

1 Introduction

The use of distributed, message-passing (MP) protocols is an increasingly advocated approach for performance and availability objectives across the spectrum of service and safety critical systems, e.g., Google File System [5] or Microsoft's state machine replication [14]. Due to the complexity of MP protocols, automated tools are desired for the debugging and verification of these protocols. *Model checking* (MC) [8] is suggested as a good formal analysis candidate given its ability to prove complex properties and especially to find bugs. However, MC is often restricted by state space explosion, i.e., when detailed models require a prohibitively large number of states to explore. Therefore, we expect from the formal model that it does not introduce unnecessary complexity and, at the same time, provides a faithful representation of the system.

We start from an established model of MP algorithms [1], written as **MP**, where two kinds of events are defined: *computation events* that receive/send messages and update the local state of the executing process, and *delivery events* that move messages from output to input buffers representing message channels. Typically, an MP algorithm is

⋆ Research supported in part by Microsoft Research, IBM Faculty Award

supposed to implement an *abstraction*, e.g., of reliable communication, an accessible and correct server or register, and the specification of this abstraction is independent of the underlying MP system. In other words, delivery events are “invisible” with respect to the desired properties of common MP algorithms. In this paper, we propose alternative semantics of delivery events and show that they result in equivalent models if delivery events are invisible. One of the proposed semantics turns out to be particularly efficient for MC as it yields smaller models than other semantics, as shown in our experiments.

The original model **MP** allows events to be interleaved arbitrarily. For example, consider the following run which consists of two computation and one delivery events: “process p_i sends message m_i to p_j”; “p_k sends m_k to p_j”; “m_i is delivered”. Note that the delivery event corresponding to m_i is interleaved with the sending of m_k. **MP** defines that computation events must empty all local input buffers (restricted computation semantic). For example, any computation event at process p_j extending the previous run must consume m_i. In an attempt to find an abstraction which faithfully and effectively models MP algorithms, we deal with the following two questions.

*First, is **MP** a general model, i.e., does the restricted computation semantic limits to a class of systems?* In fact, in our first proposed model, termed as **M-MP**, we relax the computation semantic of **MP** and allow computation events to empty a *subset* of all local input buffers. As a result, a computation event at process p_j extending the run above can also be an internal event and needs not necessarily process m_i. In this way, **M-MP** adds a new source of non-determinism compared to **MP**. However, we prove that **MP** and **M-MP** are equivalent. Therefore, **MP** is indeed a general model.

*Second, can we obtain a model which is equivalent with **MP** but yields smaller state spaces?* It is intuitive that the interleaving semantic of delivery events results in a large number of states. Therefore, our next proposed model, called **M**, eliminates explicit delivery events such that messages that are sent by some computation event $comp$ are placed in the target input buffers in an atomic step together with $comp$. Therefore, our sample run is not a valid run in **M**. A similar, and valid, run would look like this: “p_i sends m_i to p_j and m_i is delivered”; “p_k sends m_k to p_j and m_k is delivered”. Interestingly, this model together with the relaxed computation semantic result in a model equivalent with **MP** (and thus with **M-MP** too). Intuitively, this is because the non-determinism of when delivery events are scheduled is replaced by the non-determinism of deciding the set of those locally delivered messages that are processed by a computation event.

The basis of our equivalence is *stuttering* [12], a property usually used in an optimization of temporal logic MC called partial-order reduction (POR) [8]. Intuitively, two runs are stuttering equivalent if they can be partitioned into blocks where the ith block consists of subsequent system states exhibiting the same assertions in both runs. For example, if a, b, c are assertions labeling states of the system, then the runs $abc...$ and $abbc...$ are stuttering equivalent. Intuitively, we show that each block, e.g., the block of b's, corresponds to one visible computation event and the length of the block is determined by (invisible) delivery events executed in the MP model in use.

Related Work In POR, certain runs of the system are not explored such that POR guarantees that each of these runs is stuttering equivalent with at least one the explored ones.

It is in theory possible that given a model, say **M-MP**, a stuttering equivalent model, such as **MP** or **M**, is automatically generated by POR. However, independent of the technique used, POR always has some run-time overhead.

A recent work closely related to ours is [6] where, similarly to the reduction from **MP** to **M**, it is shown that a fine-grained model of distributed MP algorithms can be reduced to a stuttering equivalent coarse-grained model. The main difference between [6] and this paper lies in the system model. In [6], a special class of MP algorithms is characterized by (1) communication-closed rounds and (2) crash faults based on the Heard-Of model [7]. The reduction theorem shows that it suffices to model a run of the system as sequence of synchronized rounds where in each round every correct process sends and receives messages and updates its local state. Our system model is a general one and it does not assume that a run is divided into rounds such that a message sent in round i must be delivered before the end of round i otherwise the message is considered to be lost (communication-closedness), neither do we restrict to crash faults.

In our models, different events can be unrestrictedly interleaved, which corresponds to asynchronous systems and also it allows "unfair" runs where certain processes can make no steps. In order to model synchronous rounds or attain fairness [8], the restriction of our general model is necessary.

The presented general model also allows the modeling of process and communication faults [3]. For example, a Byzantine process is a regular process sending arbitrary messages or lossy channels can be modeled through auxiliary computation events deleting messages from channels. We remark that the proposed formal model cannot directly mimic channel overflows because channels are defined as (infinite) sets of messages. However, channel overflow can be modeled through lossy channels.

2 The System Model

2.1 Model of Computation in Message-Passing Systems

The system consists of n processes. Processes are interconnected via directed *channels* from a set *Chan* following an arbitrary topology. If there is a channel from process i to j, process i maintains an output buffer called $outbuf_j$ which contains the messages in transit, i.e., messages that are sent by process i to j but not yet delivered by the channel. Similarly, process j maintains an input buffer called $inbuf_i$ containing the messages sent by i to j and delivered by the channel. Formally, a buffer is a set of messages. By assumption, buffers are infinite and initially empty. In addition, every process i maintains a *local state*, initially taken from $Init_i$. An (initial) *configuration* of the system is a tuple $c = (p_1, ..., p_n)$ where p_i stores process i's (initial) local state and its input and output buffers. We write $proc(c) = (s_1, ..., s_n)$ to mean the tuple of local states of each process.

Transitions between configurations are modeled via *computation events*. The set of all computation events is denoted by $Comp$. Every computation event $comp$ is associated with a process i and a finite set M of messages. The effect of $comp$ on the current configuration is deterministic and is defined as follows: every messages M is removed from i's input buffers, i's local state s_i is updated, and at most one message is added

to every output buffer of i.[1] We say that $comp$ is *enabled* in a configuration c if M is a subset of the union of all input buffers of process i and the update of process i's local state is defined by $comp$. Non-determinism can be modeled through concurrently enabled computation events.

Based on the previous definitions, a *program* is a tuple $(n, Chan, Comp)$. As we will now see, we have multiple choices to model the *delivery* of messages.

2.2 Message-Passing Models

*The **MP** Model [1].* This is an existing model where special events, called *delivery events*, are used to move a message from an output to the corresponding input buffer. Formally, a delivery event of a message sent by process i to j is denoted as $del(i, j, m)$ which means that message m is removed from $outbuf_j$ of process i and placed into $inbuf_i$ of process j. Event $del(i, j, m)$ is *enabled* if m is in $outbuf_j$ of process i.

In addition, the **MP** model defines that every computation event $comp$ associated with some process i must empty *all* input buffers of i (restricted computation semantic). This means that if $comp$ is executed in a configuration c and M is the union of all input buffers of process i in c, then M is the set of messages that is associated with $comp$.

*The **M-MP** Model.* In an attempt to find out whether the restricted computation semantic means a real restriction, we define a new model called **M-MP** which is similar to **MP** but, given the union M of all input buffers of process i in configuration c, a computation event $comp$ associated with i and message set M' can be executed in c if $comp$ is enabled in c and $M' \subseteq M$. We will see that **M-MP** is equivalent with **MP**, thus, this relaxation is unnecessary.

*The **M** Model.* Intuitively, delivery events generate lots of intermediate states where the local states of the processes are unchanged. Therefore, in the next model, called **M**, we ban delivery events and place messages directly to input buffers: messages that are sent by some computation event $comp$ are moved to the corresponding input buffers in an atomic step together with $comp$. Formally, we define a computation event $comp$ associated with process i as in **M-MP** with the exception that every message m placed by $comp$ into output buffer $output_j$ of i is placed into input buffer $input_i$ of process j. In **M**, we use the same relaxed semantic of computation events as in **M-MP**. In fact, this is key to prove that the new model **M** is equivalent with **MP** and **M-MP**.

2.3 Semantics: State Transition System

Given a program P and a message-passing model **MP**, **M-MP**, or **M**, we define a *state transition system* (STS) MP_P, $M\text{-}MP_P$, or M_P in the usual way. An STS is a tuple (S, T, S_0, L) where S is the set of states, T is a set of events such that $\alpha : S \to S$ for each $\alpha \in T$, $S_0 \subseteq S$ is the set of initial states, and $L : S \to 2^{AP}$ is the labeling function which labels each state with a set of atomic propositions, a subset of AP.

[1] Note that computation events with $M = \emptyset$ can be used to model internal events.

In all STSs, S is the set of all configurations and S_0 is the set of initial configurations. In MP_P and M-MP_P, T is the set of all computation and delivery events. In M_P, T is the set of all computation events and it contains no delivery events. For every $\alpha \in T$ and $c, c' \in S$, $\alpha(c) = c'$ if α is enabled in c and if c' is the configuration which results in executing α in c as defined by **MP**, **M-MP**, **M**, respectively.

3 Equivalent Message-Passing Models with Invisible Delivery

3.1 The Notion of Equivalence and the Structure of Our Proof

Although the models **MP**, **M-MP** and **M** differ from each other in how they model the delivery of messages, we prove that given a program P they yield STSs that preserve exactly the same set of properties written in temporal logic. In order to show the equivalence, we assume that a property can only make assertions about process states by restricting that **(A1)** for every delivery event α and configuration $c \in S$ such that α is enabled in c, $L(c) = L(\alpha(c))$.

Property Language. We adopt temporal logic [8] as the property description language. Temporal operators (such as "future", "until", etc.) are interpreted over *runs*. Formally, a run σ is a sequence of configuration $c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} c_2 ...$ where c_0 is an initial configuration and $c_{i+1} = \alpha_i(c_i)$ with $\alpha_i \in T$. In this case, we call c_i a *reachable* configuration.

We ban the "next" operator from our property language. Intuitively, this is because the length of a run to reach a configuration is different depending on the MP model. For example, the delivery of multiple messages is done atomically in **M**, whereas it corresponds to a sequence of delivery events in **MP**. It is known that the use of the "next" operator can be avoided in the specification of most concurrent systems [12]. We assume that our temporal logic is linear-time (LTL). Therefore, conditions about different branches along a run cannot be specified (in contrast to CTL). This is not a limitation because the common properties of distributed protocols specify that *every* (fair) run must satisfy the property. The logic we have just described is the well-known next-free LTL (or LTL-X) [8].

Property Preservation. The key to our equivalence results is that LTL-X cannot distinguish between *stuttering equivalent* runs [12]: given an LTL-X formula f and two stuttering equivalent runs σ and σ', f holds along σ if and only if it holds along σ' [8]. Formally, two runs $\sigma = c_0 \xrightarrow{\alpha_0} c_1 ...$ and $\sigma' = c'_0 \xrightarrow{\beta_0} c'_1 ...$ are stuttering equivalent, $\sigma \approx_{st} \sigma'$ in short, if there are two infinite sequences of integers $0 = i_0 < i_1 < ...$ and $0 = j_0 < j_1 < ...$ such that for every $k \geq 0$, $L(c_{i_k}) = L(c_{i_k+1}) = ... = L(c_{i_{k+1}-1}) = L(c'_{j_k}) = L(c'_{j_k+1}) = ... = L(c'_{j_{k+1}-1})$.

Proof Structure. In order to prove stuttering equivalence between two STSs A and B we have to show for *every* run in A a stuttering equivalent run in B and vice versa. In case of three STSs, corresponding to a program P and three MP models, this means at most six proofs. In order to minimize the number of proofs we utilize that every run according to the **MP** model is also a valid run according to **M-MP**. Therefore, if we

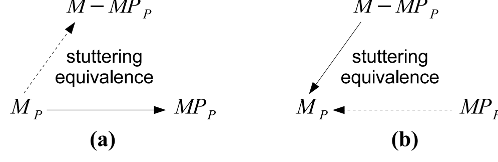

Fig. 1. Equivalence of different MP models given program P and assumption **A1**. Solid line directions are proven by (a) Theorem 1 and (b) Theorem 2 while dashed line arrows are implications thereof. The equivalence is completed by the transitivity of stuttering as stated in Theorem 3.

prove that for every run according to **M** there is a stuttering equivalent run according to **MP** (Figure 1(a)) and for every run according to **M-MP** there is a stuttering equivalent run according to **M** (Figure 1(b)), then we know that the model **M** is equivalent with **MP** and **M-MP**. In addition, the transitivity of stuttering equivalence implies that the models **MP** and **M-MP** are also equivalent. In summary, our equivalence results imply the following property preservation.

Corollary 1. *Given a program P and an LTL-X formula f such that **A1** holds, let $MP_P, M\text{-}MP_P$ and M_P be the STS corresponding to P under the message-passing model **MP**, **M-MP** and **M**, respectively. Then, f holds in MP_P iff it holds in $M\text{-}MP_P$ and iff it holds in M_P.*

In the subsequent Subsection, we precisely state and explain the main ideas of the Theorems that imply Corollary 1. The proofs of these Theorems can be found in our technical report [4].

3.2 Stuttering Equivalent Paths in MP, M-MP and M

As explained above, we have to find for each run σ_M in M_P a run σ_{MP} in MP_P such that σ_{MP} is stuttering equivalent with σ_M. Similarly, for every run $\sigma_{M\text{-}MP}$ in $M\text{-}MP_P$ we need to show a stuttering equivalent run σ_M in M_P.

First, consider $\sigma_M = c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} c_2 ....$ Since computation events in σ_M may leave messages in the input buffers of the associated process, we construct σ_{MP} such that it performs the same sequence of computation events as σ_M but delivers messages "on demand". This means that, given two subsequent computation events α_i and α_{i+1} ($i \geq 0$), α_i and α_{i+1} occur in σ_{MP} in the same order as in σ_M and α_{i+1} is directly preceded by a sequence of exactly those delivery events that are consumed by α_{i+1}. In this way, the restricted computation semantic can be respected. We have to prove that this construction of σ_{MP} is always possible and that, as delivery events are invisible, $\sigma_M \approx_{st} \sigma_{MP}$ holds. Formally, we have the following result.

Theorem 1. *Given a program P and a run $\sigma_M = c_0 \xrightarrow{\alpha_0} c_1 ...$ in M_P, a run $\sigma_{MP} = c'_0 \xrightarrow{\beta_0} c'_1 ...$ in MP_P can be constructed as follows. Initially, $c'_0 = c_0$. Furthermore, for every $i = 0, 1, ...$ and α_i, execute (in arbitrary order) a delivery event β_j for each message that is consumed by α_i in σ_M, and then execute α_i in σ_{MP}. In addition, $\sigma_M \approx_{st} \sigma_{MP}$.*

Second, consider $\sigma_{M\text{-}MP} = c_0 \xrightarrow{\alpha_0} c_1 \xrightarrow{\alpha_1} c_2 ...$ where α_i is a delivery or a computation event. Intuitively, σ_M is the same as $\sigma_{M\text{-}MP}$ without delivery events. This means that although in σ_M all messages are delivered atomically it is possible, according to **M**'s semantics, to empty only a subset of messages delivered at each process. Similarly to the previous case, $\sigma_{M\text{-}MP}$ and σ_M are stuttering equivalent because delivery events are invisible and the sequence of computation events is identical in the two runs.

Theorem 2. *Given a program P and a run $\sigma_{M\text{-}MP} = c_0 \xrightarrow{\alpha_0} c_1 ...$ in $M\text{-}MP_P$, a run $\sigma_M = c'_0 \xrightarrow{\beta_0} c'_1 ...$ in M_P can be constructed as follows. Initially, $c'_0 = c_0$. Furthermore, $\beta_1, \beta_2, ...$ is the sequence of all computation events of $\sigma_{M\text{-}MP}$ in the order as they appear in $\sigma_{M\text{-}MP}$. In addition, $\sigma_{M\text{-}MP} \approx_{st} \sigma_M$.*

Theorems 1 and 2 directly imply the stuttering equivalence of $M\text{-}MP_P$ and MP_P if we can show the transitivity of stuttering equivalence. The proof is based on the simple observation that there might be multiple partitioning, i.e., integer sequences $i_0, i_1, ...$ and $j_0, j_1, ...$, of the same pair of stuttering equivalent runs. Then, the transitivity property can be easily shown based on the partitioning where adjacent segments never have the same labels. Formally, we have the following result.

Theorem 3. *Given an STS and three runs $\sigma, \sigma', \sigma''$, $\sigma \approx_{st} \sigma'$ and $\sigma' \approx_{st} \sigma''$ imply that $\sigma \approx_{st} \sigma''$ also holds.*

4 Evaluation

Setup We compared the efficiency of MC with different MP models. We used the Murφ model checker [9] which supports symmetry reduction (SR) [8, 13], a powerful optimization known to be very efficient for fault-tolerant (FT) distributed protocols [3]. We model checked some basic properties of the Paxos algorithm [10], a highly concurrent crash-tolerant consensus algorithm. The MC results of another FT algorithm, the Byzantine Generals [11], can be found in our technical report [4].

Paxos solves consensus, where many processes can propose a local value and only one of these values is decided. Paxos uses three symmetric roles, m leaders, n acceptors, and some learners, and assumes that at most a minority of all acceptors is crash faulty. Leaders send a proposal, composed of the current local value and a proposal number, to all acceptors. An acceptor accepts a proposal only if it has not yet received any other proposal with a higher proposal number. A proposal is termed as chosen if a majority of acceptors accepts it. A chosen proposal can be learnt by the learners by collecting the accepted proposals from the acceptors. Consensus requires that no two proposals with different values are ever chosen (safety) and that a proposal is learnt (liveness).

Results The results of our MC experiments are shown in Table 1. These include the verification of the safety property of Paxos as well as false properties and fault-injected protocols where, for each case, a counterexample was found. Our experiments cover those (non-trivial) settings that were feasible to verify with Murφ.[2] For each case we ran six experiments with the three different MP models and without and with SR.

[2] All experiments ran on DETERlab machines [2] with Xeon processors and 4 GB memory.

Protocol	Param.	Property	Model	States	Transitions	Time	Result
Paxos	$m = 2$ $n = 3$	safety	**M-MP**	—			Out of mem.
			M-MP symm.	1,754,621	12,463,946	15 m	Verified
			MP	—			Out of mem.
			MP symm.	1,754,621	11,647,308	13 m	Verified
			M	1,577,161	11,411,586	3 m	Verified
			M symm.	135,271	980,290	24 s	Verified
		Erroneous safety (chosen = accepted)	**M-MP**	—			Out of mem.
			M-MP symm.	468,581	2,676,397	2 m	CE found
			MP	—			Out of mem.
			MP symm.	468,444	2,488,162	2 m	CE found
			M	476,575	2,435,659	31 s	CE found
			M symm.	49,290	256,761	8 s	CE found
Faulty Paxos (always accept proposals)		safety	**M-MP**	—			Out of mem.
			M-MP symm.	890,127	5,174,054	7 m	CE found
			MP	—			Out of mem.
			MP symm.	894,166	4,840,435	6 m	CE found
			M	1,026,203	5,598,379	1 m	CE found
			M symm.	99,781	553,159	15 s	CE found
Paxos	$m = 2$ $n = 4$	safety	**M-MP**	—			Out of mem.
			M-MP symm.	—			Out of mem.
			MP	—			Out of mem.
			MP symm.	—			Out of mem.
			M	—			Out of mem.
			M symm.	775,355	7,701,472	4 m	Verified

Table 1. MC results of Paxos with Murφ by using different MP models (**M-MP**, **MP** and **M**) and symmetry reduction.

We observe that using **M** yields significantly fewer explored states and transitions and less time than **M-MP** and **MP**. For example, in the verification of Paxos with $n = 3$, the number of states and transitions is approximately $1/13$ of those in the STS with **M-MP**. The same factor for the verification time is $1/40$. In this example, the size of the **M**-model without SR is already smaller than other models that exploit symmetries. In addition, the verification of Paxos with $n = 4$ is only feasible with the **M**-model.

References

1. H. Attiya, J. Welch. *Distributed Computing*. John Wiley and Sons, 2004.
2. T. Benzel et al. Design, Deployment, and Use of the Deter Testbed. Proc. *DETER Community Workshop on Cyber Security Experimentation and Test*, 2007.
3. P. Bokor, M. Serafini, N. Suri, H. Veith. Role-Based Symmetry Reduction of Fault-tolerant Distributed Protocols with Language Support. Proc. *ICFEM*, pp. 147–166, 2009.
4. http://www.deeds.informatik.tu-darmstadt.de/peter/MP.pdf
5. T. D. Chandra et al. Paxos Made Live: An Engineer. Persp. Proc. *PODC*, pp. 398–407, 2007.
6. 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.
7. B. Charron-Bost, A. Schiper The Heard-Of Model: Computing in Distributed Systems with Benign Failures. *Distr. Comp.*, To Appear, 2009.
8. E. Clarke, O. Grumberg, D. Peled. *Model Checking*. MIT Press, 2000.
9. D. L. Dill et al. Protocol Verif. as a Hardware Design Aid. Proc. *ICCD*, pp. 522–525, 1992.
10. L. Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
11. L. Lamport et al. The Byz. Generals Problem *ACM TOPLAS*, 4(3):382–401, 1982.
12. L. Lamport What good is Temporal Logic? Proc. *Inf. Processing.*, pp. 657–667, 1983.
13. A. Miller et al. Symmetry in Temporal Logic MC. *ACM Comp. Surv.*, 38(3):8, 2006.
14. J. Yang et al. MODIST: Transp. MC of Unmodif. Distr. Sys. Proc. *NSDI*, pp. 213–228, 2009.

Appendix

Proofs of Theorems 1-3

Theorem 1. *Given a program P and a run $\sigma_M = c_0 \xrightarrow{\alpha_0} c_1 ...$ in M_P, a run $\sigma_{MP} = c'_0 \xrightarrow{\beta_0} c'_1 ...$ in MP_P can be constructed as follows. Initially, $c'_0 = c_0$. Furthermore, for every $i = 0, 1, ...$ and α_i, execute (in arbitrary order) a delivery event β_j for each message that is consumed by α_i in σ_M, and then execute α_i in σ_{MP}. In addition, $\sigma_M \approx_{st} \sigma_{MP}$.*

Proof. The proof is an induction on i where α_i is the ith (computation) event in σ_M. Initially, $\beta_0 = \alpha_0$ because all buffers are empty in c_0 and α_0 consumes no message. Because of $proc(c_0) = proc(c'_0)$, α_0 is enabled in c'_0. Furthermore, since events are deterministic, $proc(c_1) = proc(c'_1)$. Also, all input buffers are empty in c'_1 because α_0 can only add messages to output buffers.

Given α_i, denote $\sigma_{MP} = c'_0 \xrightarrow{\alpha_0} c'_1 \xrightarrow{\beta_1} c'_2 ... \xrightarrow{\alpha_i} c'_j \xrightarrow{\beta_j} c'_{j+1} ... \xrightarrow{\beta_{j+k-1}} c'_{j+k} \xrightarrow{\alpha_{i+1}} c'_{j+k+1}$. By induction, we know that $proc(c_i) = proc(c'_j)$, where j depends on the number of delivery events before c'_j. Also, all input buffers are empty in c'_j. Observe that the sets of all messages in c_i and c'_j are identical because the same sequence of computation events has been executed before c_i and c'_j. All messages $m_1, ..., m_k$ that are consumed by α_{i+1} in σ_M must reside in output buffers in c'_j because input buffers are empty. Therefore, it is possible to execute a delivery event β_{j+l} corresponding to each m_l. Now, α_{i+1} is enabled in c'_{j+k} because $m_1, ..., m_k$ are in the input buffers and $proc(c_i) = proc(c'_{j+1}) = ... = proc(c'_{j+k})$. Furthermore, $proc(c_{i+1}) = proc(c'_{j+k+1})$ and all input buffers are empty in c'_{j+k+1}. We have proven the induction step.

To show that $\sigma_M \approx_{st} \sigma_{MP}$ we partition σ_{MP} along the computation events α_i in σ_M. Again, from **A1**, an appropriate assignment for stuttering equivalence is $i_0 = 0, i_1 = 1, ...$ and $j_0 = 0, j_1 = k_1, j_2 = k_2, ...$ where $\beta_{k_1} = \alpha_1, \beta_{k_2} = \alpha_2, ...$

Theorem 2. *Given a program P and a run $\sigma_{M\text{-}MP} = c_0 \xrightarrow{\alpha_0} c_1 ...$ in $M\text{-}MP_P$, a run $\sigma_M = c'_0 \xrightarrow{\beta_0} c'_1 ...$ in M_P can be constructed as follows. Initially, $c'_0 = c_0$. Furthermore, $\beta_1, \beta_2, ...$ is the sequence of all computation events of $\sigma_{M\text{-}MP}$ in the order as they appear in $\sigma_{M\text{-}MP}$. In addition, $\sigma_{M\text{-}MP} \approx_{st} \sigma_M$.*

Proof. The proof is an induction on i where β_i is the ith event in σ_M. Initially, similarly to Theorem 1, $\beta_0 = \alpha_0$ because all buffers are empty in c_0 and α_0 cannot be a delivery event. Because of $proc(c_0) = proc(c'_0)$, α_0 is enabled in c'_0. Furthermore, since computation events are deterministic, $proc(c_1) = proc(c'_1)$.

Assume two subsequent configurations $c_j \xrightarrow{\alpha_j} c_{j+1}$ in $\sigma_{M\text{-}MP}$ such that α_j is a computation event. Let α_j be the ith computation event executed along $\sigma_{M\text{-}MP}$. By induction, $proc(c_j) = proc(c'_i)$. Therefore, α_j can only be disabled in c'_i if some messages that are consumed by α_j are missing in the input buffers in c'_i. This is impossible as the sets of all messages in c_j and c'_i are identical because the same sequence of computation events has been executed before c_j and c'_i and in c'_i all these messages are in input buffers. Let $m_1, ..., m_k$ be the set of messages consumed by α_j and l the ID of the process associated with α_j. Even if $m_1, ..., m_k$ is a real subset of all messages in

the input buffers of process l in c'_i, **M** allows that only $m_1, ..., m_k$ are consumed by α_j in c'_i. As α_j is deterministic, $proc(c_{j+1}) = proc(c'_{i+1})$. We have proven the induction step.

Similarly to the proof in Theorem 1, we partition σ_{MP} along the computation events in σ_M. An appropriate assignment for stuttering equivalence is $i_0 = 0, i_1 = 1, ...$ and $j_0 = 0, j_1 = k_1, j_2 = k_2, ...$ where $\alpha_{k_1} = \beta_1, \alpha_{k_2} = \beta_2, ...$

Theorem 3. *Given an STS and three runs $\sigma, \sigma', \sigma''$, $\sigma \approx_{st} \sigma'$ and $\sigma' \approx_{st} \sigma''$ imply that $\sigma \approx_{st} \sigma''$ also holds.*

Proof. The runs $\sigma, \sigma', \sigma''$ are sequences of configurations in the form $\sigma = c_0c_1...$, $\sigma' = c'_0c'_1...$, and $\sigma'' = c''_0c''_1....$ From $\sigma \approx_{st} \sigma'$, there must exist $e_0 = 0 < e_1 < ...$ and $f_0 = 0 < f_1 < ...$ such that $L(c_{e_k}) = L(c_{e_k+1}) = ... = L(c_{e_{k+1}-1}) = L(c'_{f_k}) = L(c'_{f_k+1}) = ... = L(c'_{f_{k+1}-1})$. Also, from $\sigma' \approx_{st} \sigma''$, there must exist $g_0 = 0 < g_1 < ...$ and $h_0 = 0 < h_1 < ...$ such that $L(c'_{g_k}) = L(c'_{g_k+1}) = ... = L(c'_{g_{k+1}-1}) = L(c''_{h_k}) = L(c''_{h_k+1}) = ... = L(c''_{h_{k+1}-1})$.

The proof is based on the simple observation that there might be multiple partitioning of the same pair of stuttering equivalent runs. Then, we choose the one where adjacent segments never have the same labels. As we will see, this partitioning implies the transitivity property. To this end, we modify the assignments of e_k, f_k, g_k, h_k. For the brevity of the discussion, we define $k > 0$, $i \in \{e, f, g, h\}$, and $s \in \{c, c', c''\}$. For example, if $i = e$ and $s = c$ then we write s_{i_k} and mean c_{e_k}. For each k, i and s, we re-assign $i_{k+1} = i_{k+l+1}, i_{k+2} = i_{k+l+2}, ...$ if $L(s_{i_{k+1}-1}) = L(s_{i_{k+1}}) = L(s_{i_{k+2}}) = ... = L(s_{i_{k+l}})$ and $L(s_{i_{k+1}-1}) \neq L(s_{i_{k+l+1}})$.

First, we have to prove that the new assignment preserves stuttering equivalence. W.l.o.g. we consider e_k and f_k and the modified assignments e'_k and f'_k. The proof is an induction on k. For $k = 0$, we know that $L(c_{e_0}) = ... = L(c_{e_1-1}) = L(c_{e_1}) = ... = L(c_{e_2-1}) = ... = L(c_{e_l}) = ... = L(c_{e_{l+1}-1}) = L(c'_{f_0}) = ... = L(c'_{f_1-1}) = ... = L(c'_{f_{l'}}) = ... = L(c'_{f_{l'+1}-1})$ for some $l, l' > 0$ where $e'_1 = e_{l+1}$ and $f'_1 = f_{l'+1}$. We also know that $L(c_{e_0}) \neq L(c_{e_{l+1}})$ and $L(c'_{f_0}) \neq L(c'_{f_{l'+1}})$. Moreover, we know that $l = l'$. Otherwise, either $l' < l$ or $l' > l$ holds. However, $l' < l$ means $l' + 1 \leq l$ which implies $L(c_{e_{l'+1}}) = L(c'_{f_{l'+1}}) = L(c_{e_0}) = L(c'_{f_0})$. This contradicts with $L(c'_{f_0}) \neq L(c'_{f_{l'+1}})$. Similarly, $l' > l$ leads to a contradiction. Note that $l = l'$ implies that $L(c_{e'_1}) = L(c'_{f'_1})$. We have proven that $\sigma \approx_{st} \sigma'$ for $k = 0$ with e'_k and f'_k. Now, assuming that the same holds for every k, we prove that it also holds for $k + 1$ (induction). By the induction hypothesis, we have that $L(c_{e'_{k+1}}) = L(c'_{f'_{k+1}})$ and there is a common accumulated $l > 0$ such that $e'_{k+1} = e_l$ and $f'_{k+1} = f_l$. We know from $\sigma \approx_{st} \sigma'$ that $L(c_{e_l}) = L(c'_{f_l})$. Now, the induction step follows similar to the base case.

Second, we prove that the assignments e'_k and h'_k are appropriate to show that $\sigma \approx_{st} \sigma''$. We use another induction on k. From $\sigma \approx_{st} \sigma'$, we have $L(c_{e'_0}) = ... = L(c_{e'_1-1}) = L(c'_{f'_0}) = ... = L(c'_{f'_1-1})$. From $\sigma' \approx_{st} \sigma''$, we have $L(c'_{g'_0}) = ... = L(c'_{g'_1-1}) = L(c''_{h'_0}) = ... = L(c''_{h'_1-1})$. Since $e'_0 = f'_0 = g'_0 = h'_0 = 0$, we have that $L(c_{e'_0}) = L(c''_{h'_0})$. This implies $L(c_{e'_0}) = ... = L(c_{e'_1-1}) = L(c''_{h'_0}) = ... = L(c''_{h'_1-1})$, i.e., $\sigma \approx_{st} \sigma''$ for $k = 0$. We know that $L(c_{e'_1}) = L(c'_{f'_1})$ and $L(c'_{g'_1}) = L(c''_{h'_1})$. This implies that $L(c_{e'_1}) = L(c''_{h'_1})$; otherwise, it must be that $L(c'_{f'_1}) \neq L(c'_{g'_1})$. However,

Protocol	Param.	Property	Model	States	Transitions	Time	Result
OM(1)	$n = 3$	safety	**M-MP**	1,391	22,038	0.1 s	Verified
			M-MP symm.	279	4,430	0.1 s	Verified
			MP	1,391	21,354	0.1 s	Verified
			MP symm.	279	4,300	0.1 s	Verified
			M	161	2,406	0.1 s	Verified
			M symm.	44	668	0.1 s	Verified
	$n = 4$	safety	**M-MP**	125,913	3,529,132	6 m	Verified
			M-MP symm.	16,728	479,014	45 s	Verified
			MP	125,913	3,461,708	6 m	Verified
			MP symm.	16,357	458,885	40 s	Verified
			M	601	17,356	0.1 s	Verified
			M symm.	85	2,486	0.1 s	Verified
	$n = 5$	safety	**M-MP**		—		Out of mem.
			M-MP symm.		—		Timeout*
			MP		—		Out of mem.
			MP symm.		—		Timeout*
			M	2,205	155,460	5 s	Verified
			M symm.	148	10,028	1.2 s	Verified
Faulty OM(1) (two Byzantine faults)	$n = 3$	safety	**M-MP**	632	6,063	0.1 s	CE found
			M-MP symm.	127	1,276	0.1 s	CE found
			MP	632	5,561	0.1 s	CE found
			MP symm.	127	1,175	0.1 s	CE found
			M	207	1,546	0.1 s	CE found
			M symm.	49	459	0.1 s	CE found

Table 1. MC results of OM(1) with Murφ by using different message-passing models (**M-MP**, **MP** and **M**) and symmetry reduction. *A timeout of 10 hours was used.

this is impossible because, by construction of f'_k and g'_k, $f'_1 = g'_1$. In summary, we have shown that $\sigma \approx_{st} \sigma''$ for $k = 0$ with e'_k and h'_k and that $L(c_{e'_1}) = L(c''_{h'_1})$. Now, the induction step can be shown similarly to the base case.

Experiments: Oral Messages Algorithm

The Oral Messages algorithm (OM(1) for short) [11] is a synchronous, Byzantine fault-tolerant broadcast protocol using reliable channels. It defines two roles, a single general who proposes its local value (the message to be broadcast) and n symmetric lieutenants who will agree on same value if at most one general or lieutenant is Byzantine and $n \geq 3$ (safety). If the general is correct its local value must be the agreed value.

The results of model checking OM(1) are depicted in Table 1 (in Appendix). In addition to the efficiency of using **M** as MP model, we observe that the model **MP** is more efficient than **M-MP**. Debugging and verification are faster with **MP** than with **M-MP**, **MP** use strictly fewer transitions, and they yield approximately the same number of states. **MP** yields fewer transitions because configurations where no computation event can empty all input buffers of the corresponding process are dead-locks. However, configurations that are reachable in **M-MP** from such states are also reachable in **MP** along runs where the delivery of some messages is delayed. In fact, the **M-MP** and **MP** semantics imply the same set of reachable states as seen in our examples. Note that the number of visited states can vary if symmetry reduction is used (e.g., OM(1) with $n = 4$). This is because the detection of symmetric states in Murφ is done by imperfect heuristics which can (wrongly) classify symmetric states as asymmetric.