Unreliability in Practical Subclasses of Communicating Systems

Amrita Suresh
University of Oxford, UK

Nobuko Yoshida
University of Oxford, UK

Abstract

Systems of communicating automata are prominent models for peer-to-peer message-passing over unbounded channels, but in the general scenario, most verification properties are undecidable. To address this issue, two decidable subclasses, *Realisable with Synchronous Communication* (RSC) and *k-Multiparty Compatibility* (k-MC), were proposed in the literature, with corresponding verification tools developed and applied in practice. Unfortunately, both RSC and k-MC are not resilient under failures: (1) their decidability relies on the assumption of perfect channels and (2) most standard protocols do not satisfy RSC or k-MC under failures. To address these limitations, this paper studies the resilience of RSC and k-MC under two distinct failure models: *interference* and *crash-stop failures*. For interference, we relax the conditions of RSC and k-MC and prove that the inclusions of these relaxed properties remain decidable under interference, preserving their known complexity bounds. We then propose a novel crash-handling communicating system that captures wider behaviours than existing multiparty session types (MPST) with crash-stop failures. We study a translation of MPST with crash-stop failures into this system integrating RSC and k-MC properties, and establish their decidability results. Finally, by verifying representative protocols from the literature using RSC and k-MC tools extended to interferences, we evaluate the relaxed systems and demonstrate their resilience.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Distributed computing models; Theory of computation → Automata extensions

Keywords and phrases Communicating automata, lossy channel, corruption, out of order, session types, crash-stop failure

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2025.14

Funding This work is supported by EPSRC EP/T006544/2, EP/T014709/2, EP/Y005244/1, EP/V000462/1, EP/X015955/1, EP/Z0005801/1, EU Horizon (TARDIS) 101093006 and ARIA

Acknowledgements We thank Martin Vassor for his comments on an early version of this paper.

1 Introduction

Asynchronous processes that communicate using First In First Out (FIFO) channels [12], henceforth referred to as FIFO systems, are widely used to model distributed protocols, but their verification is known to be computationally challenging. The model is Turing-powerful for even just two processes communicating via two unidirectional FIFO channels [12].

To address this challenge, several efforts have focused on identifying practical yet decidable subclasses – those expressive enough to model a wide range of distributed protocols, while ensuring that verification problems such as reachability and model checking remain decidable. Most FIFO systems assume *perfect* channels, which is too restrictive to model the real-world distributed phenomena where system failures often happen. This paper investigates whether two practical decidable subclasses of communicating systems, *Realisable with Synchronous Communication* (RSC) [20] and *k-Multiparty Compatibility* (k-MC) [37], are *resilient* when integrated with two different kinds of *failures*. These failure models were originally introduced

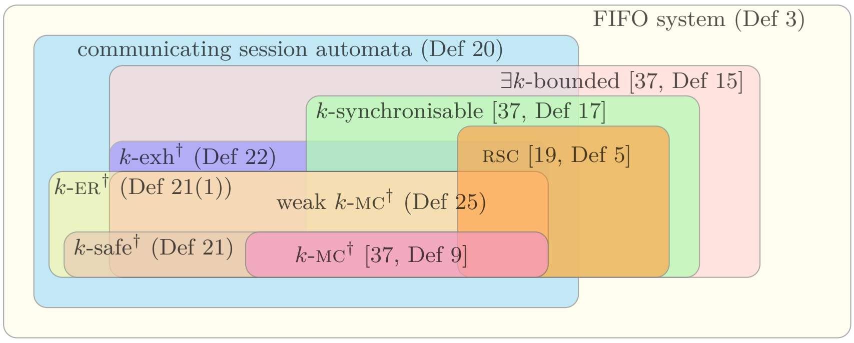


Figure 1 Classes of communication systems (since the †-marked definitions are introduced in the context of CSA (Def 20), we restrict them accordingly).

in the contexts of *contracts* [39] and *session types* [3,6]. We say a system is *resilient* under a given failure model if (i) the inclusion remains decidable, and (ii) the verification properties of interest remain decidable under that failure model.

Failures in communications. A widely studied failure model in FIFO systems is *lossy channels*. Finkel [23] showed that the termination problem is decidable for the class of *completely specified protocols*, a model which strictly includes FIFO systems with lossy channels. Abdulla and Jonsson [1] developed algorithms for verifying termination, safety, and eventuality properties for protocols on lossy channels, by showing that they belong to the class of well-structured transition systems.

Another type of failure, studied in a more practical setting, occurs when one or more processes *crash*. In the most general case, Fekete et al. [22] proved that if an underlying process crashes, no fault-tolerant reliable communication protocol can be implemented. To address this, they consider faultless models which attempt to capture the behaviour of crashes by broadcasting crash messages. Such approaches have been explored in the context of *runtime verification* techniques [33] and *session types* [3,5,6]. In this work, we closely study a failure model proposed by [3,5].

Restricting the channel behaviour. To define decidable subclasses, many works study how to restrict read and write access to channels. For two-process (binary) FIFO systems, the notion of *half-duplex* communication was introduced in [15], where at most one direction of communication is active at any time. For such systems, reachability is decidable in polynomial time. However, generalising this idea to the multiparty setting often yields subclasses that are either too restrictive or lose decidability.

Di Giusto et al. [19,20] extended this idea to multiparty systems while preserving decidability, resulting in the notion of systems *realisable with synchronous communication* (RSC). They showed that this definition overlaps with that in [15] for mailbox communication. However, in the case of peer-to-peer communication where the two definitions differ, peer-to-peer RSC behaviour was proved to be decidable. RSC systems are related to *synchronisable systems* [8,10,21], in which FIFO behaviours must admit a synchronisable execution. The tool *ReSCu* applies this idea to verify real-world distributed protocols [18].

Another approach to restricting channel behaviours is to bound the length of the channel. Lohrey [38] introduced *existentially bounded systems* (see also [25,26]) where all executions that reach a final state with empty channels can be re-ordered into a bounded execution.

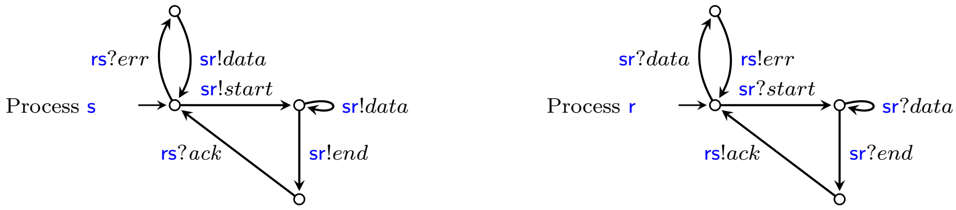


Figure 2 The above system $\mathcal{S}$ is half-duplex in the absence of errors. However, in case of (any or multiple) errors, it is no longer half-duplex.

Although many verification problems are decidable for this class of systems, checking if a system is existentially k-bounded is undecidable, even if k is given as part of the input.

A decidable bounded approach, *k-multiparty compatibility* (k-MC), was introduced in [37]. This property is defined by two conditions, *exhaustivity* and *safety*. Exhaustivity implies existential boundedness and characterises systems where each automaton behaves the same way under bounds of a certain length. Checking k-MC is decidable, and the tool k-MC-checker is implemented and applied to verify Rust [13, 34] and OCaml [32] programs.

Combining the two approaches. As far as we are aware, the intersection between expressive, decidable subclasses and communication failures is less explored. Lozes and Villard [39] studied reliability in binary half-duplex systems and showed that many communicating contracts can be verified with this model. Inspired by this, we investigate whether practical *multiparty* subclasses, RSC and k-MC, remain robust in the presence of communication errors.

Although failure models such as lossy channels are well studied, the complexity of verification in their presence is often very high — for instance, reachability in lossy systems is non-primitive recursive [23]. Our goal is not only to show that RSC and k-MC systems are resilient, but also that their inclusion remains decidable under failure models, with complexity maintained from the failure-free case.

This paper extends RSC [19, 20] and k-MC [37] by integrating two distinct failure models. RSC and k-MC systems are incomparable to each other (RSC is not a subset of k-MC and vice-versa), but both are closely related to existentially bounded systems. Figure 1 illustrates their relationship with other models.

For failures, first, we consider *interferences* from the environments by modelling *lossiness*, *corruption* (a message is altered to a different message) and *out-of-ordering* (two messages in a queue are swapped) of channels studied in the context of FIFO systems. Secondly, we consider potential *crashes* of processes introduced in the setting of session types [3, 6].

Let us consider the following simple half-duplex protocol as an example.

▶ **Example 1.** The system in Figure 2 is half-duplex under the assumption of perfect channels [15]. It consists of two processes, a sender (s) and a (dual) receiver (r), communicating via unbounded FIFO channels. A transition sr!m denotes that the sender puts (asynchronously) a message m on channel sr, and similarly, sr?m denotes that message m is consumed by the receiver from channel sr. Since the channel rs only contains messages after the receiver receives the *end* message and has emptied sr, the system satisfies the half-duplex condition. Moreover, the sender never sends *data* without having first sent *start* so the error loop is never triggered.

Now suppose the channels are prone to *corruption*. A message *data* could be altered to *end* after being sent. This allows the receiver to react prematurely by sending *ack*, while the sender continues sending *data*. As a result, both channels may become non-empty, violating the half-duplex property. Similarly in the presence of other forms of interference, as shown in Example 6, this system no longer satisfies the half-duplex condition.

Contributions and outline. The main objective of this paper is to investigate whether multiparty adaptations of half-duplex systems (RSC and k-MC) retain both their expressiveness for modelling real-world protocols and the decidability of their inclusions, with preserved complexity, under two distinct kinds of communication failures: interferences and crash-stops. § 2 introduces preliminary notions, notably FIFO systems and interference models; § 3 studies RSC under interference, and shows that relaxing certain conditions on matching send and receive actions retains both expressiveness and decidability (Theorem 19). § 4 examines k-MC with interferences, and proposes a relaxed version, k-WMC (weak k-MC), by weakening the *safety* condition. We prove that checking the k-WMC property remains decidable under interferences (Theorem 26). § 5 introduces the FIFO systems with crash-stop failures (called *crash-handling systems*), and shows that checking RSC and k-WMC under crash-stop failures is decidable (Theorems 32 and 33); § 6 defines a translation from (local) multiparty session types (MPST) to crash-handling systems and proves that this translation preserves trace semantics. This implies the decidability of RSC and k-MC within the asynchronous MPST system extended to crash-stop failures (Theorem 40); § 7 evaluates protocols from the literature extending the existing tools with support for interferences; and § 8 concludes with further related and future work. Proofs are provided in the appendix. The tools and benchmarks are publicly available from `https://github.com/NobukoYoshida/Interference-Tool`.

2 Preliminaries

For a finite set Σ, we denote by Σ^* the set of finite words over Σ, and the empty word with ε. We use $|w|$ to denote the length of the word w, and $w_1 \cdot w_2$ indicates the concatenation of two words $w_1, w_2 \in \Sigma^*$. Given a (non-deterministic) finite-state automaton $\mathcal{A}$, we denote by $\mathcal{L}(\mathcal{A})$ the language accepted by $\mathcal{A}$. Consider a finite set of processes $\mathbb{P}$ (ranged over by $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$ or occasionally by $\mathsf{r}_1, \mathsf{r}_2, \ldots$) and a set of messages Σ. In this paper, we consider the *peer-to-peer* communication model; i.e., there is a pair of unidirectional FIFO channels between each pair of processes, one for each direction of communication. In our model, processes act either by point-to-point communication or by internal actions (actions local to a single process). Moreover, in this setting, we consider messages to be atomic, akin to letters of an alphabet.

Let $Ch = \{\mathsf{pq} \mid \mathsf{p} \neq \mathsf{q} \text{ and } \mathsf{p}, \mathsf{q} \in \mathbb{P}\}$ be a set of *channels* that stand for point-to-point links. Since we are considering the peer-to-peer model of communication, there is a unique process that can send a message to (or dually, receive a message from) a particular channel. An action $a = (\mathsf{pq}, !, m) \in \mathsf{Act}$ indicates that a process p sends a message m on the channel pq. Similarly, $a = (\mathsf{qp}, ?, m) \in \mathsf{Act}$ indicates that p receives a message m on the channel qp. We henceforth denote an action $a = (\mathsf{pq}, \dagger, m) \in \mathsf{Act}$, where $\dagger \in \{!, ?\}$, in a simplified form as $\mathsf{pq}\dagger m$. An *internal action* c_{p} means that process p performs the action c. We define a finite set of actions as $\mathsf{Act} \subseteq (Ch \times \{!, ?\} \times \Sigma) \cup \mathsf{Act}_\tau$ where Act_τ is the set of all internal actions.

▶ **Definition 2** (FIFO automaton). *A FIFO automaton $\mathcal{A}_{\mathsf{p}}$, associated with p, is defined as $\mathcal{A}_{\mathsf{p}} = (Q_{\mathsf{p}}, \delta_{\mathsf{p}}, q_{0\mathsf{p}})$ where: Q_{p} is the finite set of control-states, $\delta_{\mathsf{p}} \subseteq Q_{\mathsf{p}} \times \mathsf{Act} \times Q_{\mathsf{p}}$ is the transition relation, and $q_{0\mathsf{p}} \in Q_{\mathsf{p}}$ is the initial control-state.*

Note that in this model, there are no final or accepting states.

The set of outgoing channels of process p is represented by $Ch_{\mathsf{o},\mathsf{p}} = \{\mathsf{pq} \mid \mathsf{q} \in \mathbb{P} \setminus \mathsf{p}\}$. Similarly, $Ch_{\mathsf{i},\mathsf{p}} = \{\mathsf{qp} \mid \mathsf{q} \in \mathbb{P} \setminus \mathsf{p}\}$ is the set of incoming channels of process p.

Given an action a, an *active process*, denoted by $\mathsf{proc}(a)$, is defined as: $\mathsf{proc}(\mathsf{pq}!m) = \mathsf{p}$ and $\mathsf{proc}(\mathsf{pq}?m) = \mathsf{q}$. Similarly, $\mathsf{ch}(\mathsf{pq}!m) = \mathsf{ch}(\mathsf{pq}?m) = \mathsf{pq}$.

We say a control state $q \in Q_\mathsf{p}$ is a *sending state* (resp. *receiving state*) if all its outgoing transitions are labelled by send (resp. receive) actions. If a control state is neither a sending nor receiving state, i.e., it either has both send and receive actions or neither, then we call it a *mixed state*. We say a sending (resp. receiving) state is *directed* if all the send (resp. receive) actions from that control state are to the same process. Like for finite-state automata, we say that a FIFO automaton $\mathcal{A}_\mathsf{p} = (Q_\mathsf{p}, \delta_\mathsf{p}, q_{0\mathsf{p}})$ is *deterministic* if for all transitions $(q, a, q'), (q, a', q'') \in \delta_\mathsf{p}$, $a = a' \implies q' = q''$. We write $q_1 \xrightarrow{a_1 \cdots a_n} q_{n+1}$ for $(q_1, a_1, q_2) \cdots (q_n, a_n, q_{n+1}) \in \delta_\mathsf{p}$. Unless specified otherwise, we consider non-deterministic automata, allowing mixed states, and all states do not have to be directed.

▶ **Definition 3** (FIFO system). *A FIFO system $\mathcal{S} = (\mathcal{A}_\mathsf{p})_{\mathsf{p}\in\mathbb{P}}$ is a set of communicating FIFO automata. A configuration of $\mathcal{S}$ is a pair $\gamma = (\vec{q}; \vec{w})$ where $\vec{q} = (q_\mathsf{p})_{\mathsf{p}\in\mathbb{P}}$ is called the* global state *with $q_\mathsf{p} \in Q_\mathsf{p}$ being one of the local control-states of $\mathcal{A}_\mathsf{p}$, and where $\vec{w} = (w_\mathsf{pq})_{\mathsf{pq}\in Ch}$ with $w_\mathsf{pq} \in \Sigma^*$.*

The *initial configuration* of $\mathcal{S}$ is $\gamma_0 = (\vec{q_0}; \vec{\varepsilon})$ where $\vec{q_0} = (q_{0\mathsf{p}})_{\mathsf{p}\in\mathbb{P}}$ and we write $\vec{\varepsilon}$ for the $|Ch|$-tuple $(\varepsilon, \dots, \varepsilon)$. We let $\mathcal{A}_\mathsf{p} = (Q_\mathsf{p}, \delta_\mathsf{p}, q_{0\mathsf{p}})$ be a FIFO automaton. Let $\mathcal{S} = (\mathcal{A}_\mathsf{p})_{\mathsf{p}\in\mathbb{P}}$ be the system whose initial configuration is γ_0. The FIFO automaton $\mathsf{product}(\mathcal{S})$ associated with $\mathcal{S}$ is the standard asynchronous product automaton: $\mathsf{product}(\mathcal{S}) = (Q, \delta, \vec{q_0})$ where $Q = \prod_{\mathsf{p}\in\mathbb{P}} Q_\mathsf{p}$, $\vec{q_0} = (q_{0\mathsf{p}})_{\mathsf{p}\in\mathbb{P}}$, and δ is the set of triples $(\vec{q_1}, a, \vec{q_2})$ for which there exists $\mathsf{p} \in \mathbb{P}$ such that $({q_1}_\mathsf{p}, a, {q_2}_\mathsf{p}) \in \delta_\mathsf{p}$ and ${q_1}_\mathsf{r} = {q_2}_\mathsf{r}$ for all $\mathsf{r} \in \mathbb{P} \setminus \{\mathsf{p}\}$. An *execution* $e = a_1 \cdot a_2 \cdots a_n \in \mathsf{Act}^*$ is an arbitrary finite sequence of actions. We write $\mathsf{executions}(\mathcal{S})$ for $\{e \in \mathsf{Act}^* \mid \gamma_0 \xrightarrow{e} \gamma$ for some configuration $\gamma\}$. Given $e = a_1 \cdot a_2 \cdots a_n$, we write $\mathsf{Act}(e) = \{a_1, a_2, \dots, a_n\}$. Moreover, we say two systems are trace-equivalent if they produce the same set of executions, i.e. $\mathcal{S} \approx \mathcal{S}'$ is as follows: $\forall \phi, \phi \in \mathsf{executions}(\mathcal{S}) \Leftrightarrow \phi \in \mathsf{executions}(\mathcal{S}')$.

Interferences. In this paper, we do not restrict the study to perfect channels, and instead consider that they may subject to *interferences* from the environment. Interferences are modelled as potential evolution of channel contents without a change in the global state of the system. As in [39], we model interferences by a preorder over words $\succeq \subseteq \Sigma^* \times \Sigma^*$, which will parametrise the semantics of dialogue systems. We denote by $w \succeq w'$ if w and w' are the contents of the buffer respectively before and after the interferences.

▶ **Definition 4** (Interference model). *(from [39]) An interference model is a binary relation $\succeq \subseteq \Sigma^* \times \Sigma^*$ satisfying the following axioms:*

Reflexivity	*Transitivity*	*Additivity*	*Integrity*	*Non-expansion*
$\dfrac{a \in \Sigma}{a \succeq a}$	$\dfrac{w \succeq w' \quad w' \succeq w''}{w \succeq w''}$	$\dfrac{w_1 \succeq w_1' \quad w_2 \succeq w_2'}{w_1 \cdot w_2 \succeq w_1' \cdot w_2'}$	$\dfrac{\varepsilon \succeq w}{w = \varepsilon}$	$\dfrac{w \succeq w'}{\lvert w\rvert \geq \lvert w'\rvert}$

Axiom *Additivity* defines that failures can happen at any part of the words; axiom *Integrity* says ε is the least word; and axiom *Non-expansion* says that $\succeq$ preserves the size of words. Based on interferences, we define three failures as follows:

- *Lossiness*: Possible leaks of messages during transmission are modelled by adding the axiom $a \succeq \varepsilon$.
- *Corruption*: Possible transformation of a message a into a message b is modelled by adding the axiom $a \succeq b$.
- *Out-of-order*: Out-of-order communications are modelled by adding axioms $a \cdot b \succeq b \cdot a$ for all $a, b \in \Sigma$.

We now define successor configurations for FIFO systems with interferences.

▶ **Definition 5** (Successor configuration under interference). *Let $\mathcal{S}$ be a FIFO system. A configuration $\gamma' = (\vec{q}\,'; \vec{w}\,')$ is a* successor *of another configuration $\gamma = (\vec{q}; \vec{w})$, by firing the transition $(q_\mathsf{p}, a, q'_\mathsf{p}) \in \delta_\mathsf{p}$, written $\gamma \to \gamma'$ or $\gamma \xrightarrow{a} \gamma'$, if either:* (1) *$a = \mathsf{pq}!m$ and* (a) *$q'_\mathsf{r} = q_\mathsf{r}$ for all $\mathsf{r} \neq \mathsf{p}$; and* (b) *$w'_\mathsf{pq} \preceq w_\mathsf{pq} \cdot m$ and $w'_\mathsf{rs} \preceq w_\mathsf{rs}$ for all $\mathsf{rs} \neq \mathsf{pq}$; or* (2) *$a = \mathsf{qp}?m$ and* (a) *$q'_\mathsf{r} = q_\mathsf{r}$ for all $\mathsf{r} \neq \mathsf{p}$; and* (b) *$m \cdot w'_\mathsf{qp} \preceq w_\mathsf{qp}$ and $w'_\mathsf{rs} \preceq w_\mathsf{rs}$ for all $\mathsf{rs} \neq \mathsf{qp}$.*

The condition (1-b) puts the content to a channel pq, while (2-b) gets the content from a channel pq. The reflexive and transitive closure of $\to$ is $\xrightarrow{*}$. We write $\gamma_1 \xrightarrow{a_1 \cdot a_2 \cdots a_m} \gamma_{m+1}$ for $\gamma_1 \xrightarrow{a_1} \gamma_2 \cdots \gamma_m \xrightarrow{a_m} \gamma_{m+1}$. Moreover, we write $(\gamma_1, a_1 \cdot a_2 \cdots a_m, \gamma_{m+1}) \subseteq \delta$ to denote $\{(\gamma_1, a_1, \gamma_2), \ldots, (\gamma_m, a_m, \gamma_{m+1})\} \subseteq \delta$. A configuration γ is *reachable* if $\gamma_0 \xrightarrow{*} \gamma$ and we define $RS(\mathcal{S}) = \{\gamma \mid \gamma_0 \xrightarrow{*} \gamma\}$.

A configuration $\gamma = (\vec{q}; \vec{w})$ is said to be k-*bounded* if for all $\mathsf{pq} \in Ch$, $|w_\mathsf{pq}| \leq k$. We say that an execution $e = e_1 e_2 \ldots e_n$ is k-*bounded* from γ_1 if $\gamma_1 \xrightarrow{e_1} \gamma_2 \ldots \gamma_n \xrightarrow{e_n} \gamma_{n+1}$ and for all $1 \leq i \leq n+1$, γ_i is k-bounded; we denote this as $\gamma_1 \xrightarrow{e}_k \gamma_{n+1}$.

We define the k-reachability set of $\mathcal{S}$ to be the largest subset $RS_k(\mathcal{S})$ of $RS(\mathcal{S})$ within which each configuration γ can be reached by a k-bounded execution from γ_0. Note that, given a FIFO system $\mathcal{S}$, for every integer k, the set $RS_k(\mathcal{S})$ is finite and computable.

▶ **Example 6.** Let us revisit the system in Figure 2 and explore each of the interferences with the following executions (we denote by red the messages subject to interference):

- *Corruption*: Let us consider execution $e_c = \mathsf{sr}!start \,.\, \mathsf{sr}?start \,.\, \mathsf{sr}!data \,.\, \mathsf{sr}?end \,.\, \mathsf{rs}!ack \,.\mathsf{sr}!data$. Here, the message *data* has been corrupted to *end*. Hence, process r incorrectly receives the message *end*, and assumes that process s has stopped sending *data*, while process s continues to send it.
- *Lossiness*: Consider the execution $e_\ell = \mathsf{sr}!start \,.\, \mathsf{sr}?start \,.\, \mathsf{sr}!data \,.\, \mathsf{sr}?data \,.\, \mathsf{sr}!end$. Here, the message *end* has been lost, which means process r will be stuck waiting for process s to either send *data* or *end*.
- *Out-of-order*: Let $e_o = \mathsf{sr}!start \,.\, \mathsf{sr}?start \,.\, \mathsf{sr}!data \,.\, \mathsf{sr}!end \,.\, \mathsf{sr}?end \,.\, \mathsf{rs}!ack \,.\, \mathsf{rs}?ack \,.\, \mathsf{sr}?data \,.\, \mathsf{rs}!err \,.\, \mathsf{rs}?err$. In this case, the order of *data* and *end* has been swapped in the queue, which leads to a configuration where the error message is sent.

As shown in [1, 23], for communicating automata with lossiness, the reachability set is recognisable, and the reachability problem is decidable. In the case of out-of-order scheduling, it is easy to see that the problem reduces to reachability in Petri nets. It is less clear, but it can also be reduced to Petri net reachability problem in case of corruption. We recall these proofs in Appendix A. However, the complexity of reachability for these systems is very high – it is non-primitive recursive for lossy systems [46], and Ackermann-hard for corruption and out-of-order [14]. Hence, it is still worth exploring subclasses in the presence of errors.

3 RSC systems with interferences

We first extend the definitions of synchrony in systems from [20] to consider possible interferences. The main extension relates to the definition of *matching pairs*. Intuitively, matching pairs refer to a send action and the corresponding receive action in a given execution. In the presence of interferences, it is not necessary that the same message that is sent is received (due to corruption), or that the k-th send action corresponds to the k-th receive action (due to lossiness or out-of-order). Hence we extend the definition of matching pairs.

▶ **Definition 7** (Matching pair with interference). *Given an execution $e = a_1 \dots a_n$, if there exists a channel* pq*, messages $m, m' \in \Sigma$ and $j, j', k, k' \in \{1, \dots, n\}$ where $j < j'$, and the following four conditions:*

(1) $a_j =$ pq$!m$; (2) $a_{j'} =$ pq$?m'$; (3) a_j is the k-th send action to pq *in e; and (4) $a_{j'}$ is the k'-th receive action on* pq *in e, then we say that $\{j, j'\} \subseteq \{1, \dots, n\}$ is a* matching pair with interference*, or i-matching pair.*

Note that if $m = m'$ and $k = k'$, we are back to the original definition of matching pairs in [19, Section 2], which we shall refer to henceforth as *perfect matching pairs*. When we refer to a matching pair, we mean either a perfect or i-matching pair. Moreover, our formalism allows for a single message to have more than one kind of interference, e.g. the same message can be corrupted and received out-of-order.

▶ **Example 8.** Consider the following execution $e = a_1 \dots a_5 = \mathsf{pq}!a \cdot \mathsf{qp}!b \cdot \mathsf{qp}?b \cdot \mathsf{pq}!c \cdot \mathsf{pq}?c$. For the channel qp, we have a perfect matching pair $\{2, 3\}$ which corresponds to the actions qp$!b$ and qp$?b$, the 1st send and receive action along qp. For the channel pq, we see that the 1st receive action is not pq$?a$, and hence, there is no perfect matching pair corresponding to pq$!a$. However, in case of interferences, we can have the following cases:

- If the message a is lost, i.e., pq$!a$ would be a lost action, pq$!c \cdot$ pq$?c$ would be a matched send-receive pair, and therefore, $\{4, 5\}$ would be the corresponding i-matching pair.
- If the message a was corrupted to c, then, pq$?c$ would be the receive action corresponding to pq$!a$, and we would have $\{1, 5\}$ as an i-matching pair.
- If the trace with an appended action as follows: $e' = \mathsf{pq}!a \cdot \mathsf{qp}!b \cdot \mathsf{qp}?b \cdot \mathsf{pq}!c \cdot \mathsf{pq}?c \cdot \mathsf{pq}?a$, then it could be that messages a and c were scheduled out-of-order in the channel pq. Then we have i-matching pairs $\{1, 6\}$ and $\{4, 5\}$.

We now modify the definition of *interactions* from [19] as follows.

▶ **Definition 9** (Interaction). *An* interaction *of e is either a (perfect or i-) matching pair, or a singleton $\{j\}$ such that a_j is a send action and j does not belong to any matching pair (such an interaction is called unmatched send).*

Given $e = a_1 \cdots a_n$, a set of interactions ν is called a *valid communication* of e if for every index $j \in \{1, \dots, n\}$, there exists exactly one interaction $\chi \in \nu$ such that $j \in \chi$. I.e., we need to ensure that every action in e belongs to exactly one interaction in the valid communication. We denote by $\mathsf{Comm}(e)$ the set of all valid communications associated to e.

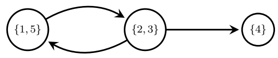
▶ **Example 10.** Revisiting Example 8, given the execution $e = a_1 \dots a_5 = \mathsf{pq}!a \cdot \mathsf{qp}!b \cdot \mathsf{qp}?b \cdot \mathsf{pq}!c \cdot \mathsf{pq}?c$, there are two valid communications, $\nu_1 = \{\{1, 5\}, \{2, 3\}, \{4\}\}$ and $\nu_2 = \{\{1\}, \{2, 3\}, \{4, 5\}\}$, and $\mathsf{Comm}(e) = \{\nu_1, \nu_2\}$.

For the rest of this section, when we refer to an execution, we are referring to a tuple (e, ν) such that $\nu \in \mathsf{Comm}(e)$. We say that two actions a_1, a_2 *commute* if $\mathsf{proc}(a_1) \neq \mathsf{proc}(a_2)$ and they do not form a matching pair.

Given an execution (e, ν) such that $e = a_1 \dots a_n$ and $\nu \in \mathsf{Comm}(e)$, we say that $j \prec_{e,\nu} j'$ if (1) $j < j'$ and (2) a_j, $a_{j'}$ do not commute in ν. We now graphically characterise *causally equivalent* executions, using the notion of a conflict graph. This allows us to establish equivalences between different executions in which actions can be interchanged.

▶ **Definition 11** (Conflict graph). *Given an execution (e, ν), the conflict graph* $\mathsf{cgraph}(e, \nu)$ *is the directed graph $(\nu, \rightarrow_{e,\nu})$ where for all interactions $\chi_1, \chi_2 \in \nu$, $\chi_1 \rightarrow_{e,\nu} \chi_2$ if there is $j_1 \in \chi_1$ and $j_2 \in \chi_2$ such that $j_1 \prec_{e,\nu} j_2$.*

The conflict graph corresponding to Example 10, $\mathsf{cgraph}(e, \nu_1)$, is:



Two executions (e, ν) and (e', ν') are causally equivalent, denoted by $(e, \nu) \sim (e', \nu')$, if their conflict graphs are isomorphic.

We are now ready to define i-RSC systems, which is the extension of RSC to include interferences. Intuitively, i-RSC executions can be reordered to mimic rendezvous (or synchronous) communication. In the case with interference, we enforce that every valid communication is equivalent to a RSC execution.

▶ **Definition 12** (i-RSC system)**.** *An execution (e, ν) is i-RSC if all matching pairs in ν are of the form $\{j, j+1\}$. A system $\mathcal{S}$ is i-RSC if for all tuples (e, ν) such that $e \in \mathsf{executions}(\mathcal{S})$ and $\nu \in \mathit{Comm}(e)$, we have $\mathsf{cgraph}(e, \nu) = \mathsf{cgraph}(e', \nu')$ where (e', ν') is an i-RSC execution.*

▶ **Example 13.** From Ex. 10, (e, ν_2) is an i-RSC execution, but (e, ν_1) is neither an i-RSC execution nor equivalent to one, as message a has to be sent before message b is received by process p while message b has to be sent before the corresponding message (which is now c due to corruption) is received by process q.

This is the strictest version, however, this can be adapted to include only one communication by assuming a single instance of ν instead of all. We formalise our observation about non-i-RSC behaviours from Example 13, and show that i-RSC still maintains the good properties of the conflict graph as in [19].

▶ **Lemma 14.** *An execution (e, ν) is causally equivalent to an i-RSC execution iff the associated conflict graph $\mathsf{cgraph}(e, \nu)$ is acyclic.*

A borderline violation for interferences defined below is a key concept for the decidability of RSC systems. Intuitively, it provides a “minimal counter-example” for non-RSC behaviour.

▶ **Definition 15** (Borderline violation)**.** *An execution (e, ν) is a borderline violation if* (1) *(e, ν) is not causally equivalent to an i-RSC execution,* (2) *$e = e' \cdot \mathsf{c}?m$ for some execution e' such that* (a) *for all $\nu' \in \mathit{Comm}(e')$, (e', ν') is equivalent to an i-RSC execution and* (b) *there exists $\nu_1 \in \mathit{Comm}(e')$ such that (e', ν_1) is an i-RSC execution.*

▶ **Lemma 16.** *$\mathcal{S}$ is i-RSC if and only if for all $e \in \mathsf{executions}(\mathcal{S})$ and $\nu \in \mathit{Comm}(e)$, (e, ν) is not a borderline violation.*

Following the same approach as in [19], we show that inclusion into the i-RSC class is decidable. For simplicity, we construct the following sets: $\mathsf{Act}_{nr} = \{\mathsf{c}!?m \mid \mathsf{c}!m \in \mathsf{Act}, \mathsf{c}?m' \in \mathsf{Act}\} \cup \{\mathsf{c}!m \mid \mathsf{c}!m \in \mathsf{Act}\}$ and $\mathsf{Act}_? = \{\mathsf{c}?m \mid \mathsf{c}?m \in \mathsf{Act}\}$. Note that $\mathsf{c}!?m$ could include a send-receive pair where the message sent is different from the one received. This ensures inclusion of matching pairs due to corruption. An i-RSC execution can be represented by a word in Act_{nr}^* and a borderline violation by a word in $\mathsf{Act}_{nr}^*.\mathsf{Act}_?$. We first show that the set of borderline violations is regular.

▶ **Lemma 17.** *Let $\mathcal{S}$ with $\mathsf{product}(\mathcal{S}) = (Q, \Sigma, \mathit{Ch}, \mathsf{Act}, \delta, q_o)$. There is a non-deterministic finite state automaton $\mathcal{A}_{bv}$ computable in time $\mathcal{O}(|\mathit{Ch}|^3|\Sigma|^2)$ such that $\mathcal{L}(\mathcal{A}_{bv}) = \{e \in \mathsf{Act}_{nr}^*.\mathsf{Act}_? \mid \exists \nu \in \mathit{Comm}(e)$ such that (e, ν) is a borderline violation$\}$.*

Next we show that the subset of executions $\mathsf{executions}(\mathcal{S})$ that begin with an i-RSC prefix and terminate with a reception is regular. The construction of the automaton $\mathcal{A}_{rsc}$ recognising such a language mimics the i-RSC executions of the original system $\mathcal{S}$, storing only the information on non-empty buffers, guessing which is the send message that will be matched by the final reception.

For the following result, we let the size $|\mathcal{A}|$ of an automaton $\mathcal{A} = (Q, \delta, q_0)$ be $|Q| + |\delta|$. Moreover, the size $|\mathcal{S}|$ of a system $\mathcal{S} = (\mathcal{A}_{\mathsf{p}})_{\mathsf{p}\in\mathbb{P}} = \sum_{\mathsf{p}\in\mathbb{P}} |\mathcal{A}_{\mathsf{p}}|$.

▶ **Lemma 18.** *Let $\mathcal{S}$ be a FIFO system. There exists a non-deterministic finite state automaton $\mathcal{A}_{rsc}$ over $\mathsf{Act}_{nr} \cup \mathsf{Act}_?$ such that $\mathcal{L}(\mathcal{A}_{rsc}) = \{e \cdot \mathsf{pq}?m \in \mathsf{Act}_{nr}^*.\mathsf{Act}_? \mid e \cdot \mathsf{pq}?m \in \mathsf{executions}(\mathcal{S})$ and $\exists \nu \in \mathit{Comm}(e)$ such that (e, ν) is an i-RSC execution$\}$, which can be constructed in time $\mathcal{O}(n^{|\mathbb{P}|+2}|Ch|^2 \times 2^{|Ch|})$, where n is the size of $\mathcal{S}$.*

Using the above lemmas, we derive the following main theorem in this section, which states that the inclusion of an i-RSC system is decidable; and the complexity is comparable to that of checking inclusion to RSC [19, Theorem 12].

▶ **Theorem 19.** *Given a system $\mathcal{S}$ of size n, deciding whether it is an i-RSC system can be done in time $\mathcal{O}(n^{|\mathbb{P}|+2}|Ch|^5 \times 2^{|Ch|} \times |\Sigma|^2)$.*

4 k-Multiparty Compatibility with interferences

We extend our analyses to consider *k-multiparty compatibility* (k-MC) which was introduced in [37] for a subset of FIFO systems, called *communicating session automata* (CSA). CSA strictly include systems corresponding to asynchronous multiparty session types [17].

▶ **Definition 20** (Communicating session automata)**.** *A deterministic FIFO automaton which has no mixed states is defined as a* session automaton*. FIFO systems comprising session automata are referred to as communicating session automata (CSA).*

In this section, we only consider communicating session automata. We begin by recalling the definition of k-MC which is composed of two properties, k-safety and k-exhaustivity.

▶ **Definition 21** (k-Safety, Definition 4 in [37])**.** *A communicating system $\mathcal{S}$ is k-*safe *if the following holds for all $(\vec{q}; \vec{w}) \in RS_k(\mathcal{S})$:*

(k-ER) $\forall \mathsf{pq} \in Ch$, *if* $\vec{w}_{\mathsf{pq}} = m.u$ *then* $(\vec{q}; \vec{w}) \xrightarrow{*}_k \xrightarrow{\mathsf{pq}?m}_k$.

(k-PG) *if* q_{p} *is receiving, then* $(\vec{q}; \vec{w}) \to_k \xrightarrow{\mathsf{pq}?m}$ *for some* $m \in \Sigma$.

The k-safety condition is composed of two properties, the first being eventual reception (k-ER) which ensures that every message sent to a channel is eventually received. The other property is progress (k-PG) where the system is not "stuck" at any receiving state.

A system is k-exhaustive if for all k-reachable configurations, whenever a send action is enabled, it can be fired within a k-bounded execution.

▶ **Definition 22** (k-Exhaustivity, Definition 8 in [37])**.** *A communicating system $\mathcal{S}$ is k-exhaustive if for all $(\vec{q}; \vec{w}) \in RS_k(\mathcal{S})$ and $\mathsf{pq} \in Ch$, if q_{p} is a sending state, then for all $(q_{\mathsf{p}}, \mathsf{pq}!m, q'_{\mathsf{p}}) \in \delta_{\mathsf{p}}$, there exists $(\vec{q}; \vec{w}) \xrightarrow{*}_k \xrightarrow{\mathsf{pq}!m}_k$.*

Example 23 shows that the reachability set of k-MC is not necessarily regular, unlike the binary half-duplex systems [15].

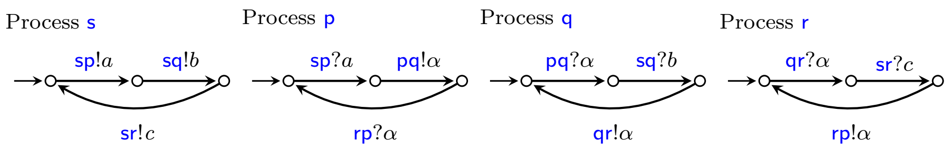


Figure 3 The above system $\mathcal{S}$ is k-MC but has a non-regular reachability set.

▶ **Example 23.** The system $\mathcal{S}$ depicted in Figure 3 is an example of a k-MC system for which the reachability set is not regular. It consists of four participants, sending messages amongst themselves. The first participant s can send equal number of a, b, c letters in a loop to participants p, q, and r respectively. The participants p, q, and r behave similarly, so let us take the example of p. It consumes one letter from the channel sp, then as a way of synchronisation sends a message α to q and waits to receive a message α from r. This ensures that participants p, q, and r consume equal number of letters from their respective channels with s. Hence, the reachability set for initial configuration (s_0, p_0, q_0, r_0) is $a^n\#b^n\#c^n$ which is context-sensitive, hence non-regular.

We prove that k-MC in the absence of errors, for a large class of systems the k-safety property subsumes k-exhaustivity.

▶ **Theorem 24.** *If a directed CSA $\mathcal{S}$ is k-safe, then $\mathcal{S}$ is k-exhaustive.*

k-MC with interferences. Theorem 24 shows that the k-safety is a too strong condition in the presence of interferences. For instance, in case of lossiness, progress cannot be guaranteed. This is because there is always the potential of losing messages and being in a receiving state forever. We are now ready to define k-weak multiparty compatibility.

▶ **Definition 25** (k-Weak Multiparty Compatibility)**.** *A communicating system $\mathcal{S}$ is* weakly k-MC*, or k-WMC, if it satisfies k-ER and is k-exhaustive.*

This notion covers a larger class of systems than k-MC systems, and it is more natural in the presence of errors. Moreover, we still retain the decidability of k-WMC in the presence of errors. We briefly discuss weaker refinements to these properties in § 8. We conclude with the following theorem which states that the k-WMC property is decidable.

▶ **Theorem 26.** *Given a system $\mathcal{S}$ with lossiness (resp. corruption, resp. out-of-order) errors, checking the k-WMC property is decidable and PSPACE-complete.*

5 Crash-stop failures

Session types [29, 30, 47] are a type discipline to ensure communication safety for message passing systems. Most session types assume a scenario where participants operate reliably, i.e. communication happens without failures. To model systems closer to the real world, Barwell et al. [3, 6] introduced session types with *crash-stop failures.* In this section, we consider the same notion for communicating systems which we define as crash-handling.

5.1 Crash-handling FIFO systems

We extend this framework to FIFO systems. As in [3, 6], we declare a (potentially empty) set of *reliable processes*, which we denote as $\mathcal{R} \subseteq \mathbb{P}$. If a process is assumed reliable, the other processes can interact with it without needing to handle its crashes. Hence if $\mathcal{R} = \mathbb{P}$,

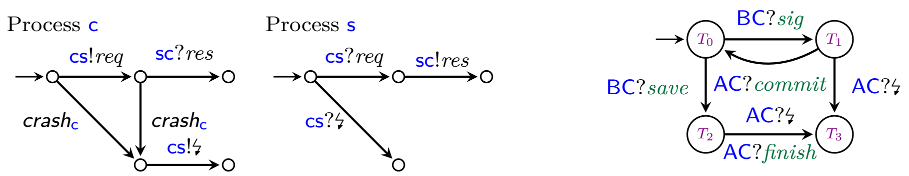


Figure 4 (a) The system $\mathcal{S}$ (right) is crash-handling. (b) FIFO automata (right) of the type in Example 36

there is no additional crash-handling behaviour for the system. In this way, we can model a mixture of reliable and unreliable processes. For simplicity in the construction, we enforce an additional constraint that in the crash-handling branches, there is no receive action from the crashed process.

We use a shorthand for the *broadcast* of a message $m \in \Sigma$ by process $\mathsf{p} \in \mathbb{P}$ along all outgoing channels: $(q, \mathsf{broadcast}_\mathsf{p}(m), q')$ if $q \xrightarrow{\mathsf{pr}_1!m.\mathsf{pr}_2!m\ldots\mathsf{pr}_n!m} q'$ such that $n = |Ch_{\mathsf{o},\mathsf{p}}|$ and $\mathsf{r}_\mathsf{i} \neq \mathsf{r}_\mathsf{j}$ for all $\mathsf{i} \neq \mathsf{j}$. We denote by $\mathsf{crash\text{-}broadcast}_\mathsf{p}(m)$ the concatenation $\mathit{crash}_\mathsf{p} \cdot \mathsf{broadcast}_\mathsf{p}(m)$ where $\mathit{crash}_\mathsf{p} \in \mathsf{Act}_\tau$ is an internal action reserved for when process p crashes.

Let $\mathcal{S} = (\mathcal{A}_\mathsf{p})_{\mathsf{p}\in\mathbb{P}}$ be a FIFO system over $\Sigma \uplus \{\lightning\}$. Let the set of reliable processes be $\mathcal{R} \subseteq \mathbb{P}$. For each $\mathsf{p} \in \mathbb{P}$:

- We divide the state set as follows : $Q_\mathsf{p} = Q_{\mathsf{p},1} \uplus Q_{\mathsf{p},2} \uplus Q_{\mathsf{p},3}$.
- Let $\mathsf{Act} \subseteq (Ch \times \{!, ?\} \times (\Sigma \uplus \{\lightning\})) \uplus \mathsf{Act}_\tau$ be the set of actions.
- We split $\delta_\mathsf{p} = \delta_{\mathsf{p},1} \uplus \delta_{\mathsf{p},2}$ such that:
  - $\delta_{\mathsf{p},1} \subseteq Q_{\mathsf{p},1} \times (Ch \times \{!, ?\} \times \Sigma) \times (Q_{\mathsf{p},1} \cup Q_{\mathsf{p},2})$, and
  - $\delta_{\mathsf{p},2} \subseteq Q_\mathsf{p} \times [(Ch \times \{!, ?\} \times \{\lightning\}) \cup \mathsf{Act}_\tau] \times Q_\mathsf{p}$.

We say that a process p has crash-handling behaviour in $\mathcal{S}$ if $\delta_{\mathsf{p},2}$ is the smallest set of transitions such that:

1. *Crash handling* (CH): For all $(q, \mathsf{rp}?a, q') \in \delta_{\mathsf{p},1}$ such that $\gamma_0 \xrightarrow{e} \gamma = (\vec{q}; \vec{w})$ and $\vec{q}_\mathsf{p} = q$ and $\mathsf{r} \in \mathbb{P} \setminus \mathcal{R}$, there exists $q'' \in (Q_{\mathsf{p},1} \cup Q_{\mathsf{p},2})$ such that $(q, \mathsf{rp}?\lightning, q'') \in \delta_{\mathsf{p},2}$.
2. *Crash broadcast* (CB): If $\mathsf{p} \notin \mathcal{R}$, then for all $q \in Q_{\mathsf{p},1}$, there exists a crash-broadcast $(q, \mathsf{crash\text{-}broadcast}_\mathsf{p}(\lightning), q_{\mathsf{stop}}) \subseteq \delta_{\mathsf{p},2}$, for some $q_{\mathsf{stop}} \in Q_{\mathsf{p},2}$ and all intermediate states belonging to $Q_{\mathsf{p},3}$.
3. *Crash redundancy* (CR): Finally, we have the condition that any dangling crash messages are cleaned up. For all $q \in Q_{\mathsf{p},2}$, $(q, \mathsf{rp}?\lightning, q) \in \delta_{\mathsf{p},2}$.

Condition (CH) enforces that every state in the system which receives from an unreliable process has a crash-handling branch, so that the receiving process is not deadlocked waiting for a message from a process that has crashed. Condition (CB) ensures that every unreliable process can non-deterministically take the internal action $\mathit{crash}_\mathsf{p}$ when it crashes and broadcast this information to all the other participants. Condition (CR) ensures that from all states in $Q_{\mathsf{p},2}$, any dangling crash messages are cleaned up from an (otherwise empty) channel.

▶ **Definition 27** (Crash-handling systems). *We say that a system $\mathcal{S}$ is crash-handling if every process $\mathsf{p} \in \mathbb{P}$ has crash-handling behaviour in $\mathcal{S}$.*

Consider the following example, which models a simple send-receive protocol between a sender and a receiver.

▶ **Example 28.** Figure 4(a) shows a crash-handling system. It consists of two processes, a server (s) and a client (c). We assume that the server is reliable, i.e. does not crash, while the client is unreliable, i.e could crash. Hence the client can crash in any control state, while the server is always ready to handle a crash when it is waiting for a message from the client.

In this construct, it is still possible to send messages to a crashed process. This is because from the perspective of the sending process, the crash of the receiving process is not necessarily known. Therefore, in this model, while processes can continue to send messages to crashed processes, the crashed processes would not be able to receive any messages.

Note that these properties are local to each individual automaton, hence the verification of these properties is decidable.

▶ **Lemma 29.** *It is decidable to check whether a system is crash-handling.*

We see that this behaviour can be appended to any FIFO system, but it does not affect the underlying verification properties of the automata. We demonstrate with the example of boundedness (i.e. checking if every execution is k-bounded for some k), but a similar argument can be used for reachability or deadlock.

▶ **Lemma 30.** *The boundedness problem is undecidable for crash-handling systems.*

5.2 Crash-handling subsystems

Next we investigate inclusion of crash-handling systems in the aforementioned classes.

Crash-handling RSC systems. Checking that the RSC property is decidable for crash-handling systems amounts to verifying if the proofs hold for communicating automata with internal actions. Let us first look at an example.

▶ **Example 31.** The system in Figure 4 is a crash-handling system that is also RSC. We see that the behaviour of the system in the absence of crashes is RSC, and in the presence of crashes, there is no additional non-RSC behaviour. Moreover, even if the *req* is sent, followed by the crash broadcast—since the crash message is never received, the behaviour of the system is still RSC. However, this need not be the case for other examples.

Next we show that the proofs from [19] can be adapted to automata with internal actions.

▶ **Theorem 32.** *Given a crash-handling system $\mathcal{S}$, it is decidable to check inclusion to the RSC class.*

Crash-handling k-WMC systems. We now show that checking k-WMC is decidable for crash-handling systems generated from a collection of local types. The reason for considering k-WMC instead of k-MC in [37, Definition 9] is that for crash-handling systems generated from local types, the end states are receiving states (as opposed to *final states*). This result is adapted from [37] with the inclusion of internal actions.

▶ **Theorem 33.** *Given a crash-handling system $\mathcal{S}$ generated from a collection of communicating session automata, it is decidable to check k-WMC, and can be done in PSPACE.*

6 Session types with crash-stop failures

This section shows that the crash-handling system strictly subsumes the crash-stop systems in [3, 5], preserving the semantics. We recall the crash-stop semantics for local types defined in [3] where the major additions are (1) a special local type stop to denote crashed processes; and (2) a crash-handling branch (*catch*) in one of branches of an external choice.

The syntax of the local types $(S, T, \ldots)$ are given as:

$$\begin{array}{llll} S, T & ::= & \mathsf{p}?\{m_i.T_i\}_{i\in I} \;\mid\; \mathsf{p}!\{m_i.T_i\}_{i\in I} & \text{(external choice, internal choice)} \\ & \mid & \mu\mathbf{t}.T \;\mid\; \mathbf{t} \;\mid\; \mathsf{end} \;\mid\; \mathsf{stop} & \text{(recursion, type variable, end, crash)} \end{array}$$

An external choice (branching) (resp., an internal choice (selection)), denoted by $\mathsf{p}?\{m_i.T_i\}_{i\in I}$ (resp., $\mathsf{p}!\{m_i.T_i\}_{i\in I}$) indicates that the *current* role is to *receive* from (or *send* to) the process p. We require pairwise-distinct, non-empty labels and the crash-handling label (*catch*) not appear in *internal* choices; and that singleton crash-handling labels not permitted in external choices. The type end indicates a *successful* termination (omitted where unambiguous), and recursive types are assumed *guarded*, i.e., $\mu\mathbf{t}.\mathbf{t}$ is not allowed, and recursive variables are unique. A *runtime* type stop denotes crashes.

We point out here that while this is a bottom-up view of the crash-handling behaviour introduced in [3], we have taken a purely type-based approach here. For a calculus based approach, we refer the reader to [4].

We define the LTS over local types and extend the notions to communicating systems. We use the same labels as the ones for communicating systems.

▶ **Definition 34** (LTS over local types)**.** *The relation $T \xrightarrow{a} T'$ for the local type of role p is defined as:*

$$\begin{array}{llll} [\mathsf{LR1}] & \mathsf{q}\dagger\{m_i.T_i\}_{i\in I} & \xrightarrow{\mathsf{pq}\dagger m_k} & T_k, \quad \textit{where } \dagger \in \{!, ?\} \textit{ and } m_k \neq \textit{catch}. \\ [\mathsf{LR2}] & T[\mu\mathbf{t}.T/\mathbf{t}] \xrightarrow{a} T' & \Longrightarrow & \mu\mathbf{t}.T \xrightarrow{a} T' \\ [\mathsf{LR3}] & \mathsf{q}\dagger\{m_i.T_i\}_{i\in I} & \xrightarrow{\textit{crash-broadcast}_{\mathsf{p}}(\lightning)} & \mathsf{stop}, \textit{ where } \dagger \in \{!, ?\}. \\ [\mathsf{LR4}] & \mathsf{q}?\{m_i.T_i\}_{i\in I} & \xrightarrow{\mathsf{qp}?\lightning} & T_k, \quad \textit{if } m_k = \textit{catch}. \\ [\mathsf{LR5}] & T & \xrightarrow{\mathsf{qp}?\lightning} & T, \quad \forall \mathsf{q} \in \mathbb{P} \setminus \{\mathsf{p}\} \textit{ for } T \in \{\mathsf{stop}, \mathsf{end}\}. \end{array}$$

Rules [LR1] and [LR2] are standard output/input and recursion rules, respectively; rule [LR3] accommodates for the crash of a process; rule [LR4] is the main rule for crash-handling where the reception of crash information leads the process to a crash-handling branch; and rule [LR5] allows any dangling crash information messages to be read in the sink states.

The LTS over a set of local types is defined as in Definition 2, where a configuration $\gamma = (\vec{T}; \vec{w})$ of a system is a pair with $\vec{T} = \{T_{\mathsf{p}}\}_{\mathsf{p}\in\mathbb{P}}$ and $\vec{w} = (w_{\mathsf{pq}})_{\mathsf{pq}\in Ch}$ with $w_{\mathsf{pq}} \in \Sigma^*$.

Next we algorithmically translate from local types to FIFO automata preserving the trace semantics. Below we write $\mu\vec{\mathbf{t}}.T$ for $\mu\mathbf{t}_1.\mu\mathbf{t}_2\ldots\mu\mathbf{t}_n.T$ with $n \geq 0$.

In order to construct the FIFO automata, we first need to define the set of states. Intuitively, this is the set of types which result from any continuation of the initial local type. Below we define *a type occurring in another type* (based on the definition in [48]).

▶ **Definition 35** (Type occurring in type, [48])**.** *We say a type T' occurs in T (denoted by $T' \in T$) if and only if at least one of the following conditions holds: (1) if T is $\mathsf{p}?\{m_i.T_i\}_{i\in I}$, there exists $i \in I$ such that $T' \in T_i$; (2) if T is $\mathsf{p}!\{m_i.T_i\}_{i\in I}$, there exists $i \in I$ such that $T' \in T_i$; (3) if T is $\mu\mathbf{t}.T_\mu$ then $T' \in T_\mu$; or (4) $T' = T$, where $=$ denotes the syntactic equality.*

▶ **Example 36.** Let $\mathbb{P} = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$ and $\mathcal{R} = \{\mathsf{B}, \mathsf{C}\}$. Consider a local type of C: $T = \mu\mathbf{t}.\mathsf{B}?\{\textit{sig}.\mathsf{A}?\{\textit{commit}.\mathbf{t}, \textit{catch}.\mathsf{end}\}, \ \textit{save}.\mathsf{A}?\{\textit{finish}.\mathsf{end}, \textit{catch}.\mathsf{end}\}\}$.

Then, the set of all $T' \in T$ is $\{T, \mathsf{B}?\{\textit{sig}.\mathsf{A}?\{\textit{commit}.\mathbf{t}, \textit{catch}.\mathsf{end}\}\}, \mathsf{A}?\{\textit{commit}.\mathbf{t}\}, \mathsf{B}?\{\textit{save}.\mathsf{A}?\{\textit{finish}.\mathsf{end}, \textit{catch}.\mathsf{end}\}\}, \mathsf{A}?\{\textit{catch}.\mathsf{end}\}, \mathsf{A}?\{\textit{finish}.\mathsf{end}\}, \mathsf{end}, \mathbf{t}\}$.

And now, we are ready to define the FIFO automata.

▶ **Definition 37** (From local types to FIFO automata). *Let T_0 be the local type of participant* p*. The automaton corresponding to T_0 is $\mathcal{A}(T_0) = (Q, \delta, q_0)$ where:*

1. $Q = \{T' \mid T' \in T_0, T' \neq \mathbf{t}, T' \neq \mu\mathbf{t}.T\} \cup \{q_{crash}\} \cup \{q_{send,\mathsf{r}} \mid \mathsf{r} \in \mathbb{P} \setminus \{\mathsf{p}\}\}$
2. $q_0 = \mathrm{strip}(T_0)$*;*
3. *δ is the smallest set of transitions such that $\forall T \in Q$:*
   a. *If $T = \mathsf{q}\dagger\{m_i.T_i\}_{i\in I}$ and $k \in I$, $m_k \neq catch$, and $\dagger \in \{!, ?\}$*
   $$\begin{cases} (T, \mathsf{pq} \dagger m_k, \mathrm{strip}(T_k)) \in \delta & \text{if } T_k \neq \mathbf{t} \\ (T, \mathsf{pq} \dagger m_k, \mathrm{strip}(T')) \in \delta & \text{if } T_k = \mathbf{t} \text{ with } \mu\mathbf{t}.T' \in T_0. \end{cases}$$
   b. *If $T = \mathsf{q}?\{m_i.T_i\}_{i\in I}$ with $k \in I$, $m_k = catch$*
   $$\begin{cases} (T, \mathsf{qp}?\lightning, \mathrm{strip}(T_k)) \in \delta & \text{if } T_k \neq \mathbf{t} \\ (T, \mathsf{qp}?\lightning, \mathrm{strip}(T')) \in \delta & \text{if } T_k = \mathbf{t} \text{ with } \mu\mathbf{t}.T' \in T_0. \end{cases}$$
   c. *If $T \notin \{\mathsf{stop}, \mathsf{end}\}$, then $(T, \textit{crash-broadcast}_{\mathsf{p}}(\lightning), \mathsf{stop}) \subseteq \delta$ where*
      i. $(T, \textit{crash}, q_{crash}) \in \delta$
      ii. $(q_{crash}, \mathsf{pr}_1!\lightning, q_{send,\mathsf{r}_1}) \in \delta$
      iii. $(q_{send,\mathsf{r}_i}, \mathsf{pr}_{i+1}!\lightning, q_{send,\mathsf{r}_{i+1}}) \in \delta\ \forall i \in \{1, \ldots, n-2\}$*, where* $n = |Ch_{o,\mathsf{p}}|$
      iv. $(q_{send,\mathsf{r}_{n-1}}, \textit{crash}, \mathsf{stop}) \in \delta$
   d. *If $T \in \{\mathsf{stop}, \mathsf{end}\}$, then $(T, \mathsf{qp}?\lightning, T) \in \delta$ for all $\mathsf{q} \in \mathbb{P} \setminus \{\mathsf{p}\}$.*

where $\mathrm{strip}(T) \stackrel{\text{def}}{=} \mathrm{strip}(T')$ *if* $T = \mu\mathbf{t}.T'$*; otherwise* $\mathrm{strip}(T) \stackrel{\text{def}}{=} T$.

▶ **Example 38.** The FIFO automata constructed from the type in Example 36 is shown in Figure 4. We see that for all receiving actions from process A, which is not in the reliable set of processes, there is a crash-handling branch, where:

$$T_0 = \mathsf{B}?\{sig.\mathsf{A}?\{commit.\mathbf{t}, catch.\mathsf{end}\},\ save.\mathsf{A}?\{finish.\mathsf{end}, catch.\mathsf{end}\}\}$$
$$T_1 = \mathsf{A}?\{commit.\mathbf{t}, catch.\mathsf{end}\} \quad T_2 = \mathsf{A}?\{finish.\mathsf{end}, catch.\mathsf{end}\} \quad T_3 = \mathsf{end}$$

We now prove that the automata generated from a local type can be composed into communicating session automata, and this translation preserves the semantics.

▶ **Lemma 39.** *Assume T_{p} is a local type. Then $\mathcal{A}(T_{\mathsf{p}})$ is deterministic, directed and has no mixed states. Moreover, $T_{\mathsf{p}} \approx \mathcal{A}(T_{\mathsf{p}})$, i.e. $\forall \phi, \phi \in \mathsf{executions}(T_{\mathsf{p}}) \Leftrightarrow \phi \in \mathsf{executions}(\mathcal{A}(T_{\mathsf{p}}))$.*

By Lemma 39, we derive that the resulting systems belong to the class of crash-handling systems, and the problem of checking RSC and k-WMC is decidable for this class.

▶ **Theorem 40.** *The FIFO system generated from the translation of crash-stop session types is a crash-handling system. Moreover, it is decidable to check inclusion to the* RSC *and k-*WMC *classes.*

7 Experimental evaluation

We verify protocols in the literature under interferences in order to compare how inclusion to the RSC and k-WMC classes change; and which of RSC and k-MC is more resilient under failures. We used the tools, RSC-checker *ReSCu* [18] (implemented in OCaml), and k-MC-checker `kmc` [37] (implemented in Haskell). The *ReSCu* tool implements a version of the out-of-order scheduling with an option, so we use this available option to take our benchmark.

The `kmc` tool implements the options to check (1) the k-exhaustivity (k-EXH, Def. 22); (2) the k-eventual reception (k-ER, Def 21); and (3) the progress (k-PG, Def 21). The k-weak multiparty compatibility condition (k-WMC, Def 25) no longer checks for k-PG but checks k-EXH and k-ER. Hence checking (1,2,3) gives us the justification whether k-WMC is more resilient than k-MC. For the out-of-order, we implemented the out-of-order scheduling in Haskell mirroring the implementation as in *ReSCu*. To model lossiness, we add reception self-loops as defined in completely specified protocols [23], and for corruption, we allow the sending of arbitrary messages; both of these are implemented in Python.

Table 1 shows the evaluation results. From the benchmarks in [18], we selected all the relevant benchmarks (CSA and peer-to-peer) in order to evaluate them by `kmc`. Interestingly, in the case of out-of-order errors, all of the protocols which satisfy k-MC without errors still satisfy k-MC. However RSC does not satisfy some k-MC protocols. This would imply that in most real-world examples, the flexibility in behaviour introduced by relaxing the FIFO condition does not affect the inclusion to k-MC. On the other hand, under lossiness and corruption, most examples no longer belong to k-MC. More specifically, k-PG fails for most cases. The k-ER also fails for many cases, especially in the presence of corruption. This justifies a relevance of our definition of k-WMC under those two failures.

Table 1 Experimental evaluation of benchmarks in the *ReSCu* and `kmc` tool, under the presence of no errors, out-of-order, lossiness and corruption errors. Note that the systems were checked for $k \leq 10$. TO denotes timeout after 5 minutes. The *-marked examples originate from the ReSCu tool [18], having been translated from other papers, as detailed in the original publication.

Protocol	No errors		Out of order		Lossiness				Corruption			
	k-MC	RSC	k-MC	RSC	k-exh	k-ER	k-PG	RSC	k-exh	k-ER	k-PG	RSC
Alternating Bit [43]	yes	yes	yes	yes	yes	yes	no	yes	yes	yes	no	yes
Alternating Bit [7]	yes	no	yes	yes	yes	yes	no	yes	yes	yes	no	yes
Bargain [36]	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no	yes
Client-Server-Logger [37]	yes	no	yes	no	yes	yes	no	yes	yes	yes	no	yes
Cloud System v4 [27]	yes	yes	yes	no	no	no	no	yes	no	no	no	no
Commit protocol [11]	yes	yes	yes	yes	yes	no	no	yes	yes	no	no	yes
Dev System [42]	yes	yes	yes	yes	yes	no	no	yes	yes	no	no	yes
Elevator [11]	yes	no	yes	no	yes	yes	no	no	no	TO	no	no
Elevator-dashed [11]	yes	no	yes	no	no	no	no	no	no	TO	no	no
Elevator-directed [11]	yes	no	yes	no	no	no	no	no	no	TO	no	no
Filter Collaboration [50]	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no	yes
Four Player Game [36]	yes	yes	yes	no	no	yes	no	yes	no	yes	no	yes
Health System [37]	yes	yes	yes	yes	yes	no	no	yes	yes	no	no	yes
Logistic [41]	yes	yes	yes	yes	yes	yes	no	yes	no	no	no	yes
Sanitary Agency (mod) [44]	yes	yes	yes	yes	yes	no	no	yes	yes	TO	no	yes
TPM Contract [28]	yes	yes	yes	no	yes	yes	no	yes	no	no	no	no
2-Paxos 2P3A (App F)	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	no	yes
Promela I* [18]	yes	no	yes	no	yes	yes	no	yes	yes	yes	yes	yes
Web Services* [18]	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no	yes
Trade System* [18]	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no	yes
Online Stock Broker* [18]	no	no	no	no	no	no	no	yes	no	no	no	yes
FTP* [18]	yes	yes	yes	yes	yes	yes	no	yes	no	no	no	yes
Client-server* [18]	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no	yes
Mars Explosion* [18]	yes	yes	yes	yes	yes	no	no	yes	no	no	no	yes
Online Computer Sale* [18]	no	yes	no	yes	yes	yes	no	yes	no	no	no	yes
e-Museum* [18]	yes	yes	yes	no	yes	no	no	yes	yes	no	no	yes
Vending Machine* [18]	yes	yes	yes	yes	yes	yes	no	yes	yes	no	no	yes
Bug Report* [18]	yes	yes	yes	no	yes	yes	no	yes	no	no	no	yes
Sanitary Agency* [18]	no	yes	no	yes	yes	yes	no	yes	yes	no	no	yes
SSH* [18]	no	yes	no	yes	yes	yes	no	yes	yes	yes	no	yes
Booking System* [18]	no	yes	no	yes	yes	yes	no	yes	yes	no	no	yes
Hand-crafted Example* [18]	no	yes	no	yes	yes	no	no	yes	yes	no	no	yes

We implement a k-bounded version of the Paxos protocol [35], a consensus algorithm that ensures agreement in a distributed system despite failures like lossiness and reordering, using a process of proposing and accepting values (c.f. Appendix F for the details). This version limits retry attempts to k. Our implementation (for 2 retries, 2 proposers and 3 acceptors) shows it is k-MC and RSC both without errors, and k-MC under lossiness. Since Paxos does not assume corruption, it is unsurprising that it is no longer k-MC under corruption.

8 Conclusion and further related work

In this paper, we derived decidability and complexity results for two subclasses, RSC and k-MC, under two types of communication failures: interferences and crash-stop failures. In the absence of errors, RSC systems and k-MC systems are incomparable, even if we restrict the analyses to 1-MC systems. For example, [37, Example 4] is 1-MC but not RSC. Conversely, [19, Example 4] is RSC but does not satisfy the progress condition, and hence is not k-MC for any $k \in \mathbb{N}$. Despite these distinctions, both classes aim to generalise the concept of half-duplex communication to multiparty systems. This serves as our primary motivation for examining failures in a uniform way across both RSC and k-MC systems.

In the interference model, we introduced i-RSC systems, which relax the matching-pair conditions in RSC; and k-WMC which omits the progress condition to accommodate a model with no final states. We proved that the inclusion problem for these relaxed properties remain decidable within the same complexity class as their error-free counterparts. The evaluation results in § 7 confirm that relaxed systems are more resilient than the original ones.

As the second failure model, we investigated crash-stop failures. We defined crash-handling communicating systems which strictly include the class of local types with crash-stop failures. We also proved that both RSC and k-MC properties are decidable for this class. Note that multiparty session types with crash-stop failures studied in [6] are limited to *synchronous* communications. Meanwhile, the asynchronous setting in [3] restricts expressiveness to a set of local types projected from global types (which is known to be less expressive than those not using global types [45]). Therefore, both of these systems are strictly subsumed by our crash-handling system as proven in Theorem 40.

Integrating the k-MC-checker and the *ReSCu* tool (with support for crash-stop failures) into the Scala toolchain of [3] is a promising direction for future work, potentially enabling the verification of a broader class of programs than those considered in [3, 45].

Due to the need to model failures in real-world distributed systems, various failure-handling systems have been studied in the session types literature, e.g., affine session types [24, 31, 40], link-failure [2] and event-driven failures [49]. Interpreting their failures into our framework would offer a uniform analysis of behavioural typed failure processes.

References

1 P. Abdulla and B. Jonsson. Verifying programs with unreliable channels. In *[1993] Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 160–170, June 1993. doi:10.1109/LICS.1993.287591.

2 Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session Types for Link Failures. In Ahmed Bouajjani and Alexandra Silva, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, pages 1–16, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-60225-7_1.

3 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Designing Asynchronous Multiparty Protocols With Crash Stop Failures. In *European Conference on Object Oriented Programming*, volume 263, pages 1:1–1:30, 2023. doi:10.4230/LIPIcs.ECOOP.2023.1.

4 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Designing Asynchronous Multiparty Protocols with Crash-Stop Failures, May 2023. arXiv:2305.06238 [cs]. URL: `http://arxiv.org/abs/2305.06238`.

5 Adam D. Barwell, Ping Hou, Nobuko Yoshida, and Fangyi Zhou. Crash-Stop Failures in Asynchronous Multiparty Session Types. *Logical Methods in Computer Science*, 2025. URL: `https://arxiv.org/abs/2311.11851`.

6 Adam D. Barwell, Alceste Scalas, Nobuko Yoshida, and Fangyi Zhou. Generalised Multiparty Session Types with Crash-Stop Failures. In Bartek Klin, S\lawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory (CONCUR 2022)*, volume 243 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 35:1–35:25, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. URL: `https://drops.dagstuhl.de/opus/volltexte/2022/17098`, `doi:10.4230/LIPIcs.CONCUR.2022.35`.

7 Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 1–12, Berlin, Heidelberg, 1996. Springer. `doi:10.1007/3-540-61474-5_53`.

8 Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Etienne Lozes, and Amrita Suresh. A Unifying Framework for Deciding Synchronizability. In Serge Haddad and Daniele Varacca, editors, *32nd International Conference on Concurrency Theory (CONCUR 2021)*, volume 203 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:18, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/14391`, `doi:10.4230/LIPIcs.CONCUR.2021.14`.

9 Benedikt Bollig, Dietrich Kuske, and Ingmar Meinecke. Propositional Dynamic Logic for Message-Passing Systems. *Logical Methods in Computer Science*, Volume 6, Issue 3, September 2010. URL: `https://lmcs.episciences.org/1057`, `doi:10.2168/LMCS-6(3:16)2010`.

10 Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the Completeness of Verifying Message Passing Programs Under Bounded Asynchrony. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, Lecture Notes in Computer Science, pages 372–391, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-319-96142-2_23`.

11 Ahmed Bouajjani, Constantin Enea, Kailiang Ji, and Shaz Qadeer. On the completeness of verifying message passing programs under bounded asynchrony. In *CAV 2018*, pages 372–391, 2018. `doi:10.1007/978-3-319-96142-2\_23`.

12 Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, April 1983. `doi:10.1145/322374.322380`.

13 Zak Cutner, Nobuko Yoshida, and Martin Vassor. Deadlock-free asynchronous message reordering in Rust with multiparty session types. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, pages 246–261, New York, NY, USA, March 2022. Association for Computing Machinery. `doi:10.1145/3503221.3508404`.

14 Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The Reachability Problem for Petri Nets Is Not Elementary. *J. ACM*, 68(1):7:1–7:28, 2021. `doi:10.1145/3422822`.

15 Gérard Cécé and Alain Finkel. Verification of programs with half-duplex communication. *Information and Computation*, 202(2):166–190, November 2005. URL: `https://www.sciencedirect.com/science/article/pii/S0890540105001082`, `doi:10.1016/j.ic.2005.05.006`.

16 Gérard Cécé, Alain Finkel, and S. Purushothaman Iyer. Unreliable Channels Are Easier to Verify Than Perfect Channels. *Information and Computation*, 124(1):20–31, January

1996. URL: https://www.sciencedirect.com/science/article/pii/S0890540196900036, doi:10.1006/inco.1996.0003.

17 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty Compatibility in Communicating Automata: Characterisation and Synthesis of Global Session Types. In Fedor V. Fomin, Rūsiņš Freivalds, Marta Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming*, Lecture Notes in Computer Science, pages 174–186, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-39212-2_18.

18 Loïc Desgeorges and Loïc Germerie Guizouarn. RSC to the ReSCu: Automated Verification of Systems of Communicating Automata. In Sung-Shik Jongmans and Antónia Lopes, editors, *Coordination Models and Languages*, Lecture Notes in Computer Science, pages 135–143, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-35361-1_7.

19 Cinzia Di Giusto, Loïc Germerie Guizouarn, and Etienne Lozes. Multiparty half-duplex systems and synchronous communications. *Journal of Logical and Algebraic Methods in Programming*, 131:100843, February 2023. URL: https://www.sciencedirect.com/science/article/pii/S2352220822000967, doi:10.1016/j.jlamp.2022.100843.

20 Cinzia Di Giusto, Loïc Germerie Guizouarn, and Etienne Lozes. Towards Generalised Half-Duplex Systems. *Electron. Proc. Theor. Comput. Sci.*, 347:22–37, October 2021. arXiv:2110.00145 [cs]. URL: http://arxiv.org/abs/2110.00145, doi:10.4204/EPTCS.347.2.

21 Cinzia Di Giusto, Laetitia Laversa, and Etienne Lozes. On the k-synchronizability of Systems. In Jean Goubault-Larrecq and Barbara König, editors, *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Proceedings*, Lecture Notes in Computer Science, pages 157–176, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-45231-5_9.

22 Alan Fekete, Nancy Lynch, Yishay Mansour, and John Spinelli. The impossibility of implementing reliable communication in the face of crashes. *J. ACM*, 40(5):1087–1107, November 1993. URL: https://dl.acm.org/doi/10.1145/174147.169676, doi:10.1145/174147.169676.

23 Alain Finkel. Decidability of the termination problem for completely specified protocols. *Distrib Comput*, 7(3):129–135, March 1994. doi:10.1007/BF02277857.

24 Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional Asynchronous Session Types: Session Types without Tiers. *Proc. ACM Program. Lang.*, 3(POPL):28:1–28:29, 2019. doi:10.1145/3290341.

25 Blaise Genest, Dietrich Kuske, and Anca Muscholl. On Communicating Automata with Bounded Channels. *Fundamenta Informaticae*, 80(1-3):147–167, January 2007. Publisher: IOS Press. URL: https://content.iospress.com/articles/fundamenta-informaticae/fi80-1-3-09.

26 Blaise Genest, Anca Muscholl, and Dietrich Kuske. A Kleene Theorem for a Class of Communicating Automata with Effective Algorithms. In Cristian Calude, Elena Calude, and Michael J. Dinneen, editors, *Developments in Language Theory, 8th International Conference, DLT 2004, Auckland, New Zealand, December 13-17, 2004, Proceedings*, volume 3340 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2004. URL: https://doi.org/10.1007/978-3-540-30550-7_4, doi:10.1007/978-3-540-30550-7_4.

27 Matthias Güdemann, Gwen Salaün, and Meriem Ouederni. Counterexample guided synthesis of monitors for realizability enforcement. In *ATVA 2012*, pages 238–253, 2012. doi:10.1007/978-3-642-33386-6_20.

28 Sylvain Hallé and Tevfik Bultan. Realizability analysis for message-based interactions using shared-state projections. In *SIGSOFT 2010*, pages 27–36, 2010. URL: http://doi.acm.org/10.1145/1882291.1882298, doi:10.1145/1882291.1882298.

29 Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming. In *Proceedings of ESOP 1998*, volume 1381, pages 22–138. Springer, 1998.

30 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of*

programming languages, POPL '08, pages 273–284, New York, NY, USA, January 2008. Association for Computing Machinery. doi:10.1145/1328438.1328472.

31 Ping Hou, Nicolas Lagaillardie, and Nobuko Yoshida. Fearless Asynchronous Communications with Timed Multiparty Session Protocols. In *ECOOP 2024*. Schloss Dagstuhl - Leibniz-Zentrum f"ur Informatik, 2024.

32 Keigo Imai, Julien Lange, and Rumyana Neykova. Kmclib: Automated inference and verification of session types from OCaml programs. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 379–386. Springer, 2022. doi:10.1007/978-3-030-99524-9_20.

33 Shokoufeh Kazemlou and Borzoo Bonakdarpour. Crash-Resilient Decentralized Synchronous Runtime Verification. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 207–212, October 2018. ISSN: 2575-8462. URL: https://ieeexplore.ieee.org/abstract/document/8613969, doi:10.1109/SRDS.2018.00032.

34 Nicolas Lagaillardie, Rumyana Neykova, and Nobuko Yoshida. Stay Safe under Panic: Affine Rust Programming with Multiparty Session Types. In *36th European Conference on Object-Oriented Programming*, volume 222 of *LIPIcs*, pages 4:1–4:29. Schloss Dagstuhl–Leibniz-Zentrum fur Informatik, 2022. doi:10.4230/LIPIcs.ECOOP.2022.4.

35 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

36 Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In *POPL 2015*, pages 221–232, 2015. URL: http://doi.acm.org/10.1145/2676726.2676964, doi:10.1145/2676726.2676964.

37 Julien Lange and Nobuko Yoshida. Verifying Asynchronous Interactions via Communicating Session Automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 97–117, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-25540-4_6.

38 Markus Lohrey and Anca Muscholl. Bounded MSC Communication. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 2002. URL: https://doi.org/10.1007/3-540-45931-6_21, doi:10.1007/3-540-45931-6_21.

39 Etienne Lozes and Jules Villard. Reliable Contracts for Unreliable Half-Duplex Communications. In Marco Carbone and Jean-Marc Petit, editors, *Web Services and Formal Methods*, Lecture Notes in Computer Science, pages 2–16, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-29834-9_2.

40 Dimitris Mostrous and Vasco Thudichum Vasconcelos. Affine sessions. In *International Conference on Coordination Languages and Models*, pages 115–130, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-662-43376-8_8.

41 OMG. Business Process Model and Notation, 2018. https://www.omg.org/spec/BPMN/2.0/.

42 Roly Perera, Julien Lange, and Simon J. Gay. Multiparty compatibility for concurrent objects. In *PLACES 2016*, pages 73–82, 2016. doi:10.4204/EPTCS.211.8.

43 Introduction to protocol engineering. Available at http://cs.uccs.edu/~cs522/pe/pe.htm, 2006.

44 Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. *IJBPIM*, 1(2):116–128, 2006. doi:10.1504/IJBPIM.2006.010025.

45 Alceste Scalas and Nobuko Yoshida. Less is more: multiparty session types revisited. *Proc. ACM Program. Lang.*, 3(POPL):30:1–30:29, January 2019. URL: https://dl.acm.org/doi/10.1145/3290343, doi:10.1145/3290343.

46 Philippe Schnoebelen. Verifying lossy channel systems has nonprimitive recursive complexity. *Inf. Process. Lett.*, 83(5):251–261, 2002. `doi:10.1016/S0020-0190(01)00337-4`.

47 Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817, pages 398–413, 1994.

48 Martin Vassor and Nobuko Yoshida. Refinements for multiparty message-passing protocols: Specification-agnostic theory and implementation. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria*, volume 313 of *LIPIcs*, pages 41:1–41:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. URL: `https://doi.org/10.4230/LIPIcs.ECOOP.2024.41`, `doi:10.4230/LIPICS.ECOOP.2024.41`.

49 Malte Viering, Raymond Hu, Patrick Eugster, and Lukasz Ziarek. A Multiparty Session Typing Discipline for Fault-Tolerant Event-Driven Distributed Programming. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. `doi:10.1145/3485501`.

50 Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997. URL: `http://doi.acm.org/10.1145/244795.244801`, `doi:10.1145/244795.244801`.

A Proofs from §2

We sketch the proofs of reachability under the presence of errors. The proof is well-known for FIFO systems with lossiness and out-of-order errors, but we for completeness sake, we include it here.

FIFO systems with lossiness. As shown in [16], for lossy systems, the reachability set is recognisable, and the reachability problem is decidable.

▶ **Lemma A.1** ([16]). *For FIFO systems with lossiness, the reachability set is recognisable.*

Proof. The proof follows from the fact that upward-closed sets are recognisable. Moreover, the complement of the reachability set of lossy FIFO systems is upwards-closed (under the subword ordering). Therefore, the reachability set is recognisable (since recognisable sets are closed under complementation). ◀

FIFO systems with out-of-order errors. For FIFO systems with out-of-order errors, reachability is decidable.

▶ **Lemma A.2.** *For FIFO systems with out-of-order errors, reachability is decidable.*

Proof. FIFO systems with out-of-order errors can be seen as FIFO systems with bags, or multisets. Loosely speaking, this can translate to a vector addition system with states (VASS), and [14] shows that reachability is Ackermann-complete for VASS. ◀

FIFO systems with corruption. In case of corruption, the reachability problem is decidable.

▶ **Lemma A.3.** *For FIFO systems with corruption errors, the reachability problem is decidable.*

Proof. Let $\mathcal{S}$ be a FIFO system with corruption. Let us consider a configuration $\gamma = (\vec{q}, \vec{w})$, with $\vec{w} = (w_{\mathsf{pq}})_{\mathsf{pq} \in Ch}$. Without loss of generality, let $w_{\mathsf{rs}} \in \Sigma^*$ be the channel contents of channel rs such that $|w_{\mathsf{rs}}| = n$. Since the channel is corrupt, we know that $\{(\vec{q}, \vec{w}') \mid w'_{\mathsf{pq}} = w_{\mathsf{pq}}$ for all $\mathsf{pq} \neq \mathsf{rs}$ and $w'_{\mathsf{rs}} \in \Sigma^*$ and $|w'_{\mathsf{rs}}| = n\} \subseteq RS(\mathcal{S})$, since the existing channel contents can be corrupted to any other word of the same length. Hence, the reachability set is the union of all such sets of configurations. In order to find out which lengths of words are reachable for each configuration, it is sufficient to modify the automata such that there is only one letter replacing all the transitions. This ensures that we correctly count the length of all the words that are reachable. This translates to checking the reachability of a VASS (since each channel can now be seen as a counter without zero tests), and then, once we know if a word of length n is reachable, we can be sure that any word of length n is reachable from the initial state. Hence, the problem reduces to the reachability problem in VASS. ◀

B Proofs from §3

▶ **Lemma 14.** *An execution (e, ν) is causally equivalent to an i-RSC execution iff the associated conflict graph* $\mathsf{cgraph}(e, \nu)$ *is acyclic.*

Proof. The left to right implication follows from two observations: first, two causally equivalent executions have isomorphic conflict graphs. Secondly, the conflict graph of an RSC execution is acyclic, because for an RSC execution and vertices χ_1, χ_2 in the conflict graph, $\chi_1 \rightarrow_{e,\nu} \chi_2$ if there is $j_1 \in \chi_1$ and $j_2 \in \chi_2$ such that $j_1 \prec_{e,\nu} j_2$. Moreover, if there is more than action in either χ_1 or χ_2, for i-RSC executions by definition, $min(\chi_1) \prec_{e,\nu} min(\chi_2)$. Therefore, if there is a cycle in the conflict graph, then this would imply $min(\chi_2) \prec_{e,\nu} min(\chi_1)$, which would be a contradiction.

For the converse direction, let us assume that a conflict graph associated to $e = a_1a_2 \ldots a_n$ is acyclic. Let us consider the associated communication set ν. Let $\chi_1 \ll \cdots \ll \chi_n$ be a topological order on ν. Let $e' = \chi_1 \cdots \chi_n$ be the corresponding RSC execution, and ν' the communication set associated to e' that is RSC.

Let σ be the permutation such that $e' = a_{\sigma(1)} \cdots a_{\sigma(n)}$. Following the proof idea in [19], we show that e is causally equivalent to e'. Let j, j' be two indices of e, and let us show that $j \prec j'$ iff $\sigma(j) \prec \sigma(j')$.

We have that $\{j, j'\}$ is a matching pair in e, iff, by construction, $\{\sigma(j), \sigma(j')\}$ is a matching pair in e'. If $\{j, j'\}$ is not a matching pair of e, then let χ and χ' be the interactions containing j and j' respectively. Since $j \prec j'$, there is an arrow between χ and χ' in the conflict graph, and moreover a_j and $a_{j'}$ cannot commute. Note that there is an arrow in the conflict graph of e' as well. Since the conflict graph is acyclic, we have $\sigma(j) \prec \sigma(j')$.

◀

▶ **Lemma 16.** *$\mathcal{S}$ is i-RSC if and only if for all $e \in$* $\mathsf{executions}(\mathcal{S})$ *and $\nu \in$ Comm(e), (e, ν) is not a borderline violation.*

Proof. By definition, if there exists an execution (e, ν) in system $\mathcal{S}$ such that it is a borderline violation, then $\mathcal{S}$ is not i-RSC. Conversely, if $\mathcal{S}$ is not i-RSC, let (e, ν) be (one of) the shortest execution that is not causally equivalent to an i-RSC execution. Then, $e = e' \cdot a$ such that for all $\nu' \in \mathsf{Comm}(e')$, we have (e', ν') is equivalent to an i-RSC execution. Let ν'' be the communication set of e' such that it is a subset of the communication set ν. Let (e'', ν'') be the i-RSC execution that is causally equivalent to (e', ν''). Then, there exists an execution $\hat{e}$ such that $(\hat{e}, \nu)$ is an execution of $\mathcal{S}$. Moreover, if a is a send action, then $(\hat{e}, \nu)$ is i-RSC which is a contradiction. Therefore, (e, ν) is a borderline violation. ◀

▶ **Lemma 17.** *Let $\mathcal{S}$ with* $\mathsf{product}(\mathcal{S}) = (Q, \Sigma, Ch, \mathsf{Act}, \delta, q_o)$. *There is a non-deterministic finite state automaton $\mathcal{A}_{bv}$ computable in time $\mathcal{O}(|Ch|^3|\Sigma|^2)$ such that $\mathcal{L}(\mathcal{A}_{bv}) = \{e \in \mathsf{Act}^*_{nr}.\mathsf{Act}_? \mid \exists \nu \in Comm(e)$ such that (e, ν) is a borderline violation$\}$.*

Proof. Let $\mathcal{A}_{bv} = (Q_{bv}, \delta_{bv}, q_{0,bv}, \{q_f\})$, with $Q_{bv} = \{q_{0,bv}, q_f\} \cup (Ch \times \mathsf{Act} \times \{0, 1\})$, and for all $a, a' \in \mathsf{Act}_{nr}$, for all $c \in Ch$, $m, m' \in \Sigma$:

1. $(q_{0,bv}, a, q_{0,bv})$: this is a loop on the initial state that reads all interactions until the chosen send message
2. $(q_{0,bv}, \mathsf{pq}!m, (\mathsf{pq}, \mathsf{pq}!m, 0))$: this is the transition where we non-deterministically select the send message that is matched to the final reception to be borderline.
3. In case we do not consider out-of-order errors, we add the following step: $((\mathsf{c}, a, 0), a', (\mathsf{c}, a, 0))$, if $\mathsf{ch}(a') \neq \mathsf{c}$: again loop for every communication but we do not accept any further communication on the channel c in order to stay borderline. Note that this step is skipped if we consider the general case with out-of-order errors as we can have matched pairs between a matched send and receive action.
4. $((\mathsf{c}, a, 0), a', (\mathsf{c}, a', 1))$, if $\mathsf{proc}(a) \cap \mathsf{proc}(a') \neq \emptyset$: here, the second interaction that will take part in the conflict graph cycle is guessed. We ensure there is a process in common with a for there to be an edge between them.
5. $((\mathsf{c}, a, 1), a', (\mathsf{c}, a, 1))$, if $\mathsf{ch}(a') \neq \mathsf{c}$: once again a loop for every interaction.
6. $((\mathsf{c}, a, 1), a', (\mathsf{c}, a', 1))$, if $\mathsf{proc}(a) \cap \mathsf{proc}(a') \neq \emptyset$: the next vertex (or vertices) (if any) of the conflict graph is guessed.
7. $((\mathsf{c}, a, 1), \mathsf{pq}?m', q_f)$, if $\mathsf{proc}(a) \cap \mathsf{proc}(\mathsf{pq}?m') \neq \emptyset$: finally, an execution is accepted if it closes the cycle.

Moreover, each transition of $\mathcal{A}_{bv}$ can be constructed in constant time, so $\mathcal{A}_{bv}$ can be constructed in time $\mathcal{O}(|Ch|^3|\Sigma|^2)$. ◀

▶ **Lemma 18.** *Let $\mathcal{S}$ be a FIFO system. There exists a non-deterministic finite state automaton $\mathcal{A}_{rsc}$ over $\mathsf{Act}_{nr} \cup \mathsf{Act}_?$ such that $\mathcal{L}(\mathcal{A}_{rsc}) = \{e \cdot \mathsf{pq}?m \in \mathsf{Act}_{nr}^*.\mathsf{Act}_? \mid e \cdot \mathsf{pq}?m \in \mathsf{executions}(\mathcal{S})$ and $\exists \nu \in \mathit{Comm}(e)$ such that (e, ν) is an i-RSC execution$\}$, which can be constructed in time $\mathcal{O}(n^{|\mathbb{P}|+2}|Ch|^2 \times 2^{|Ch|})$, where n is the size of $\mathcal{S}$.*

Proof. Let $\mathcal{A}_{rsc} = (Q_{\mathrm{rsc}}, \delta_{\mathrm{rsc}}, q_{0,\mathrm{rsc}}, \{q_f\})$ be the non-deterministic automata, with $Q_{\mathrm{rsc}} = Q \times (\{\varepsilon\} \cup Ch) \times 2^{Ch} \cup \{q_f\}$. We define the transitions as follows:

1. First, while performing the action $a \in \mathsf{Act}_{nr}$, $(q, \chi, S) \xrightarrow{a} (q', \chi', S')$ if
   - $(q, v) \implies (q', v')$ in the underlying transition system, for some buffer values v, v' and for all $\mathsf{c} \in Ch$, $v_c \neq \emptyset$ iff $\mathsf{c} \in S$ and $v'_{\mathsf{c}} \neq \emptyset$ iff $\mathsf{c} \in S'$, and
   - this condition is added in the absence of out-of-order errors: if a $= \mathsf{pq}!?m'$, then $\mathsf{pq} \notin S$, and
   - either $\chi = \chi'$, or $a = \mathsf{c}!m$ and $\chi' = \mathsf{c}$
2. Second, while performing the action $a = \mathsf{pq}?m$, we have $(q, \chi, S) \xrightarrow{a} q_f$ if $\chi = \mathsf{pq}$ and $(q, a, q') \in \delta_{\mathcal{S}}$ for some q'.

Each transition of $\mathcal{A}_{rsc}$ can be constructed in constant time. An upper bound on the number of transitions can be computed as follows: if $(q, \chi, S) \xrightarrow{a} (q', \chi', S')$ is a transition, then q and q' only differ on at most two machines (the one that executed the send, and the one that executed the receive), so there are at most n^2 different possibilities for q' once q and a are fixed. There are at most two possibilities for χ' once χ and a are fixed, and S' is fully determined by S and a. Finally, there are $n^{|\mathbb{P}|}(1 + |Ch|) \times 2^{|Ch|} \times 2 \times |Ch|$ possibilities for a choice of the pair $((q, \chi, S), a)$. ◀

▶ **Theorem 19.** *Given a system $\mathcal{S}$ of size n, deciding whether it is an i-RSC system can be done in time $\mathcal{O}(n^{|\mathbb{P}|+2}|Ch|^5 \times 2^{|Ch|} \times |\Sigma|^2)$.*

Proof. The set of borderline violations of a system $\mathcal{S}$ can be expressed as $\mathcal{L}(\mathcal{A}_{rsc}) \cdot \mathsf{Act}_{nr} \cap \mathcal{L}(\mathcal{A}_{bv})$. Therefore, checking for inclusion in i-RSC reduces to checking the emptiness of this intersection, which can be done in time $\mathcal{O}(n)$. ◀

Proof. Let $\mathcal{A} = (Q_\mathcal{A}, \delta_\mathcal{A}, q_{0,\mathcal{A}}, \{q_f\})$ be the non-deterministic automata, with $Q_\mathcal{A} = Q \times (\{\varepsilon\} \cup Ch) \times 2^{Ch} \cup \{q_f\}$ over alphabet $L_\mathcal{S} \cup \{\#\} \cup \Sigma$ such that $L(\mathcal{A}) = P(\mathcal{S})$. We define an automaton $\mathcal{A}_P$ over the alphabet Act of communications such that for all i-RSC executions e, $e \in L(\mathcal{A}_P)$ iff there is $\gamma \in P(\mathcal{S})$ such that $\gamma_0 \xrightarrow{e} \gamma$. Let us define $\mathcal{A}_P = (Q_P, \delta_P, Q_{0,P}, F_P)$ as follows:

- $Q_P = Q_\mathcal{S} \times Q_\mathcal{S} \times Q_\mathcal{A}^{|Ch|} \times Q_\mathcal{A}^{|Ch|}$, such that the components indicate the current control-state, the assumed final state, the current position of the last unreceived letter on each channel, and a copy of the initial positions on each channel, respectively.
- We say that state $(q_\mathcal{S}, q_F, q_\mathcal{A}, q_{Ch}) \in Q_{0,P}$ if: $q_\mathcal{S} = \gamma_0$, $q_\mathcal{A} = q_{Ch}$, and after reading the contents of q_F, the state is reachable from q_I for all buffers. We similarly define $F_\mathcal{A}$.
- We say $(q_\mathcal{S}, q_F, q_\mathcal{A}, q_{Ch}) \xrightarrow{c} (q'_\mathcal{S}, q'_F, q'_\mathcal{A}, q'_{Ch}) \in \delta_P$ if (1) $q_F = q'_F$; (2) $q_{Ch} = q'_{Ch}$; (3) $q_\mathcal{S} \xrightarrow{c}_\mathcal{S} q'_\mathcal{S}$; and either (4.1) if $c = \mathsf{pq}!?a$ then $q_\mathcal{A} = q'_\mathcal{A}$, else (4.2) if $c = \mathsf{pq}!a$ then $q_{\mathcal{A},\mathsf{pq}} \xrightarrow{a}_\mathcal{A} q'_{\mathcal{A},\mathsf{pq}}$. Hence, we only progress the automaton when there is an addition to the channel contents.

Now that the automaton is defined, we can say that the intersection $RS(\mathcal{S}) \cap P(\mathcal{S}) = \emptyset$ iff $L(\mathcal{A}_p) \cap L_(\mathcal{A}_{rsc}) = \emptyset$. ◀

C Proofs from §4

▶ **Theorem 24.** *If a directed CSA $\mathcal{S}$ is k-safe, then $\mathcal{S}$ is k-exhaustive.*

Proof. We prove this by contradiction. Let us assume that $\mathcal{S}$ is not k-exhaustive. In other words, there exists $s \in RS_k(\mathcal{S})$ and $\mathsf{pq} \in Ch$, such that q_p is a sending state and there is no execution of the kind $s \xrightarrow{*}_k \xrightarrow{\mathsf{pq}!m}_k$. In other words, the channel pq has k messages already, i.e. $|w_\mathsf{pq}| = k$. However, since $s \in RS_k(\mathcal{S})$, and $\mathcal{S}$ is k-safe, and more specifically satisfies eventual reception, there exists a configuration $t \in RS_k(\mathcal{S})$ such that $s \xrightarrow{*}_k \xrightarrow{\mathsf{pq}?m'}_k t$, such that $w_\mathsf{pq} = m' \cdot u$. Moreover, since the execution $s \xrightarrow{e}_k t$ is k-bounded, we can be sure that there have been no new sends along the channel pq. Furthermore, since $\mathcal{S}$ is directed and has no mixed states, in configuration t, p is still at state q_p. Therefore, we now have $t \xrightarrow{\mathsf{pq}!m}_k$, which contradicts our initial assumption. ◀

▶ **Theorem 26.** *Given a system $\mathcal{S}$ with lossiness (resp. corruption, resp. out-of-order) errors, checking the k-WMC property is decidable and PSPACE-complete.*

Proof. To check whether $\mathcal{S}$ is not k-exhaustive, i.e., for each sending state q_p and send action from q_p, we check whether there is a reachable configuration from which this send action cannot be fired. Hence, we need to search $RS_k(\mathcal{S})$, which has an exponential number of configurations (wrt. k). Note that due to interferences, each of these configurations can now have modified channel contents. We need to store at most $|\mathbb{P}|^n|Ch||\Sigma|^k$ configurations, where n is the maximum number of local states of a FIFO automata, following ideas from [37] and [9]. Hence, the problem can be decided in polynomial space when k is given in unary.

Next, to show that k-WER is decidable, we check for every such reachable configuration, that there exists a receive action from the same channel (note that we do not need to ensure it is the same message).

◀

D Proofs from §5

▶ **Lemma 29.** *It is decidable to check whether a system is crash-handling.*

Proof. To check whether a system $\mathcal{S}$ is crash-handling, we need to check the two properties:

1. Checking if (CH) is satisfied amounts to checking if for all receiving transitions $\tau \in \delta_{\mathsf{p},1}$ such that the sending process is in $\mathbb{P} \setminus \mathcal{R}$, there exists a transition $\tau' \in \delta_{\mathsf{p},2}$ that handles the crash.
2. To check (CB), we need to check every state in a process in $\mathbb{P} \setminus \mathcal{R}$ and ensure it can send crash messages when it crashes.

Both of these are structural checks made on the graph of the automata, hence, checking this is decidable. ◀

▶ **Lemma 30.** *The boundedness problem is undecidable for crash-handling systems.*

Proof. Since every FIFO system is a crash-handling process under the condition that $\mathbb{P} = \mathcal{R}$, this lemma is trivially true. Moreover, every FIFO system $\mathcal{S}$ with $\mathcal{R} \subsetneq \mathbb{P}$ can be translated to a crash-handling system $\mathcal{S}'$ such that $\mathcal{S}$ is k-bounded iff $\mathcal{S}'$ is at most $k+1$-bounded. This can be done by adding a new sink state q_{sink} such that for all receiving transitions $(q, \mathsf{c}?a, q') \in \delta_{\mathsf{p},1}$, we add to $\delta_{\mathsf{p},2}$ a transition $(q, \mathsf{c}?\lightning, q_{\mathsf{sink}})$. Hence, (CH) will be handled. For enforcing (CB), we add to each state q of an unreliable process p the following transition $(q, \mathsf{crash\text{-}broadcast}_{\mathsf{p}}(\lightning), q_{\mathsf{sink}})$. Hence, both conditions are satisfied, and these additional transitions do not add any unboundedness to the channels (and at most one message extra to each channel). Moreover, if the original system is unbounded, then the same execution would be enabled in $\mathcal{S}'$. Hence, $\mathcal{S}$ is k-bounded iff $\mathcal{S}'$ is at most $k+1$-bounded. ◀

▶ **Theorem 32.** *Given a crash-handling system $\mathcal{S}$, it is decidable to check inclusion to the* RSC *class.*

Proof. This amounts to checking the RSC property in automata with internal actions. Intuitively, this amounts to "skipping" the internal actions in the respective NFAs. In order to prove this, we let $\mathsf{Act}_{nr} = \{\mathsf{c}!?m \mid \mathsf{c}!m \in \mathsf{Act}, \mathsf{c}?m \in \mathsf{Act}\} \cup \{\mathsf{c}!m \mid \mathsf{c}!m \in \mathsf{Act}\} \cup \mathsf{Act}_{\tau}$. Then we follow the construction as before. Note that since internal actions do not have a channel associated to them, we do not need to make any further changes. In the conflict graph, they are considered as nodes with only process edges between them, hence, do not form cycles and can be ignored. ◀

▶ **Theorem 33.** *Given a crash-handling system $\mathcal{S}$ generated from a collection of communicating session automata, it is decidable to check k-*WMC*, and can be done in* PSPACE*.*

Proof. First, we observe that for any $k \in \mathbb{N}$, $RS_k(\mathcal{S})$ and $\rightarrow_k$ are finite. Moreover, there are at most $n \cdot |\mathbb{P}|$ control states in the system, where $n = max(\{|Q_{\mathsf{p}}| \mid \mathsf{p} \in \mathbb{P}\}$.

k-exhaustivity: We check whether $\mathcal{S}$ is not k-exhaustive, i.e., for each sending state q_{p} and send action from q_{p} , we check whether there is a reachable configuration from which this send action cannot be fired. The presence of internal actions and the absence of final states does not alter this proof.

eventual reception: For each receiving state q_{p}, we check whether there is a reachable configuration from which one receive action of p is enabled, followed by a send action that matches another receive. We proceed as in the case for k-exhaustivity with additional space to remember whether we are looking for the receiving state or for a matching send action. Note that the presence of the internal actions does not affect this property either. This is because they do not modify the channel bounds (and hence, are not bounded by k), and do not increase the size of $RS_k(\mathcal{S})$.

Therefore, the proofs can directly follow from [37, Theorem 2]. ◀

E Proofs from §6

Before we construct the resulting FIFO automata, we first need to show that the set of states is finite.

▶ **Lemma E.1.** *Given a local type T, the set $\{T' \mid T' \in T\}$ is finite.*

Proof. Let us consider each of the four conditions to build the set $\{T' \mid T' \in T\}$ from T. In cases (1), (2) and (4), we see that T' is a strict prefix of T. Moreover, in case (3), we do not add any element to the set. Therefore, since the length of T is finite, the set $\{T' \mid T' \in T\}$ is finite. ◀

Hence, we can conclude that the automata constructed from T has finitely many states.

▶ **Lemma 39.** *Assume T_{p} is a local type. Then $\mathcal{A}(T_{\mathsf{p}})$ is deterministic, directed and has no mixed states. Moreover, $T_{\mathsf{p}} \approx \mathcal{A}(T_{\mathsf{p}})$, i.e. $\forall \phi, \phi \in \mathsf{executions}(T_{\mathsf{p}}) \Leftrightarrow \phi \in \mathsf{executions}(\mathcal{A}(T_{\mathsf{p}}))$.*

Proof. For the determinism, we note that all m_i in $\mathsf{q}!\{m_i.T_i\}_{i\in I}$ and $\mathsf{q}?\{m_i.T_i\}_{i\in I}$ are distinct. Apart from this, there is only a unique local action that can be taken from a state in case of crash. Therefore, the automaton is deterministic. Directedness is by the syntax of branching and selection types, and the fact that the internal action crash leads to a state without interacting with any other participant. The message broadcast, although not explicit, can be viewed as a sequence of send transitions, thereby making the system directed. Finally, for the absence of mixed states, we can check a state is either sending or receiving state as one state represents either branching and selection type, along with stop, end which are receiving states, and all the intermediate states (between a crash until the stop state) are sending states.

We now show that the translation preserves the semantics.

We show that $T \xrightarrow{\tau} T'$ if $\mathrm{strip}(T) \xrightarrow{\tau} \mathrm{strip}(T')$. Base case: Considering a transition of size 0, it trivially holds as $q_0 = \mathrm{strip}(T_0)$. Let us assume it holds for a transition of size k. Now, let us consider a single transition from $T \xrightarrow{a} T'$. If the transition a belongs to [LR1] or [LR2] (resp. [LR5]) that leads to T', then there exists a transition in 3a (resp. 3d) that leads to $\mathrm{strip}(T')$. Similarly, the correspondence holds for transitions from [LR4] to 3b. Note that rule [LR2] is implicitly applied because of the strip() function. Finally, [LR3] corresponds to 3c.

The reverse direction follows as above. The only change is for the crash-handling behaviour. Here, we modify the condition as follows: if $\mathrm{strip}(T) \xrightarrow{\tau} q$, then there exists $T \xrightarrow{\tau'} T'$ such that there is a unique, deterministic sequence of transitions τ'' such that $q \xrightarrow{\tau''} \mathrm{strip}(T')$. For all cases except 3c, the proof above can be adapted (with $q = \mathsf{T}'$ and $\tau'' = \varepsilon$). For 3c. we see that for every sequence of transitions taken, there is a unique continuation that leads to stop, and the concatenation of $\tau.\tau'' = \tau$ and leads to stop.

◀

▶ **Theorem 40.** *The FIFO system generated from the translation of crash-stop session types is a crash-handling system. Moreover, it is decidable to check inclusion to the* RSC *and k-*WMC *classes.*

Proof. This can be seen by assuming $Q_{\mathsf{p},1} = \{T' \mid T' \in T_0, T' \neq \mathbf{t}, T' \neq \mu\mathbf{t}.T\} \setminus \{\mathsf{end}, \mathsf{stop}\}$. Moreover, $Q_{\mathsf{p},2} = \{\mathsf{stop}, \mathsf{end}\}$ and $Q_{\mathsf{p},3} = \{q_{\mathsf{crash}}\} \cup \{q_{\mathsf{send},\mathsf{r}} \mid \mathsf{r} \in \mathbb{P} \setminus \{\mathsf{p}\}\}$. Moreover, rules in 3a correspond to $\delta_{\mathsf{p},1}$ and 3b (CH), 3c (CB), 3d (CR) constitute $\delta_{\mathsf{p},2}$. With this correspondence, we see that all the conditions are satisfied, hence, it is a crash-handling system.

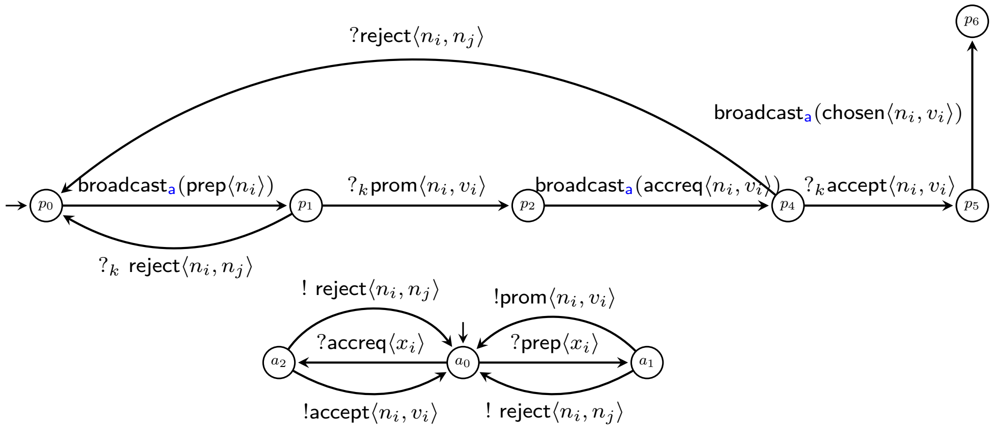


Figure 5 FIFO automata of the proposer and acceptor respectively in Paxos (broadcast$_a$(m) refers to broadcasting message m to all acceptors, ?$_k m$ refers to receiving at least k messages m).

From Lemmas 39 and the above result, we see that the FIFO system generated from the translation of crash-stop local types is a crash-handling system and a collection of communicating session automata. Moreover, from Theorems 32 and 33, it is decidable to check inclusion into the RSC and k-WMC classes. Therefore, we can check the inclusion for collection of local types generated from crash-stop session types. ◀

F Paxos Protocol

In this section, we implement a basic version of the single-decree Paxos protocol, which was originally described in [35]. The Paxos algorithm has been used to implement a fault-tolerant distributed system, which is essentially a consensus algorithm aimed to ensure that network agents can agree on a single proposed value. We model this protocol using FIFO systems with faulty channels, and go on to explore if it belongs to any of the above-mentioned classes of communicating systems.

The protocol.

We assume a subset of processes to be *proposers*, i.e. processes that can propose values. The consensus algorithm ensures that exactly one value among the proposed values is chosen. A correct implementation of the protocol must ensure that:

- Only a value that has been proposed will be chosen.
- Only one single value is chosen by the network.
- A process never knows that a value has been chosen unless the value has actually been chosen.

The Paxos setting assumes the customary asynchronous, non-Byzantine model, in which:

- Agents operate at arbitrary speed, may fail by crashing, and may restart. However, it is assumed that agents maintain persistent storage that survives crashes.
- Messages can take arbitrarily long to be delivered, can be duplicated, and can be lost or delivered out of order, but they are not corrupted.

Paxos agents implement three roles: i) a proposer agent proposes values towards the network for reaching consensus; ii) an acceptor accepts a value from those proposed, whereas a majority of acceptors accepting the same value implies consensus and signifies protocol termination; and iii) a learner discovers the chosen consensus value.

The implementation of the protocol may proceed over several rounds. A successful round has two phases: Prepare and Accept. The protocol ensures that in the case where a consensus value v has already been chosen among the majority of the network agents, broadcasting a new proposal request with a higher proposal number will result in choosing the already chosen consensus value v. Following this fact, we assume for simplicity that a learner has the same implementation as a proposer.

- Requirement 1: An acceptor must accept the first proposal that it receives, i.e. for the acceptor, there must be a path from the initial state, which accepts the first proposal it gets.
- Requirement 2: If multiple proposals are chosen, they all have the same proposal value.

We model a bounded-version of Paxos with a FIFO system which can have any of the above-mentioned errors except corruption. The automata in Fig 5 show an example implementation of a proposer and an acceptor, with a majority of k agents needed for consensus. The action $!!\mathsf{msg}$ refers to broadcasting the message msg across all channels, and $?_k\mathsf{msg}$ refers to receiving k msg messages. Both these actions can be unrolled and expressed as a combination of simple actions. Moreover, we assume that for each value of n_i there is a copy of the same set of transitions. And since we cannot compare values in finite automata, we sequentially order the automata with increasing values of n_i. Note that this model is a CSA, and hence, we can test the `kmc` tool and the ReSCu tool on an implementation. We verify that a 2-bounded Paxos with 2 proposers and 3 acceptors (2-Paxos2P3A) is k-MC and RSC in the presence of lossiness.