A MODEL OF ACTORS AND GREY FAILURES

LAURA BOCCHI ●^{a}, JULIEN LANGE ●^{b}, SIMON THOMPSON ●^{a}, AND A. LAURA VOINEA ●^{a,c}

^{a} University of Kent, Canterbury, UK
   e-mail address: l.bocchi@kent.ac.uk, s.j.thompson@kent.ac.uk

^{b} Royal Holloway, University of London, Egham, UK
   e-mail address: julien.lange@rhul.ac.uk

^{c} University of Glasgow, Glasgow, UK
   e-mail address: laura.voinea@glasgow.ac.uk

ABSTRACT. Existing models for the analysis of concurrent processes tend to focus on fail-stop failures, where processes are either working or permanently stopped, and their state (working/stopped) is known. In fact, systems are often affected by grey failures: failures that are latent, possibly transient, and may affect the system in subtle ways that later lead to major issues, such as crashes, limited availability or overload.

We introduce a model of actor-based systems with grey failures, based on two interlinked layers: an actor model, given as an asynchronous process calculus with discrete time, and a failure model that represents failure patterns that can be injected into the system. Our failure model captures not only fail-stop node and link failures, but also grey failures, which might be partial or transient.

We give a behavioural equivalence relation based on weak barbed bisimulation to compare systems on the basis of their ability to recover from failures, and on this basis we define some desirable properties of reliable systems. By doing so, we reduce the problem of checking reliability properties of systems to the problem of checking bisimulation.

1. INTRODUCTION

Many real-world computing systems are affected by non-negligible degrees of unpredictability, such as unexpected delays and failures, which are not straightforward to capture accurately. Several works contribute towards a formal account of unpredictability, for example in the context of process calculi – potentially including session types – by extending calculi to model node failures [FGL$^+$96, RH97], link failures [APN17], and a combination of link and node failures [BH03]; these calculi also add a variety of program constructs to deal with failures including escapes [CGY16], interrupts [HNY$^+$13], exceptions [FLMD19], and timeouts [LZ05, BY07, LP11]. Most existing models assume a fail-stop model of failure,

where processes are either working or permanently stopped, and their state of being either working or stopped is known. In fact, systems are often affected by grey failures: failures that are latent, possibly transient, and may affect the system in subtle ways that later lead to major issues, such as crashes, limited availability and overload. The symptoms of grey failure tend to be ambiguous. Several kinds of grey failure have been studied in the last decade such as transient failure (e.g., a component is down at periodic intervals), partial failure (only some sub-components are affected), or slowdown [GSS$^+$18]. In a distributed system, processes may have different perceptions as to the state of health of the system (aka *differential observation*) [HGZ$^+$17]. Grey failures tend to be behind many service incidents in cloud systems, and in these situations traditional fault tolerance mechanisms tend to be ineffective or counterproductive [HGZ$^+$17]. Diagnosis can be challenging and lengthy: for example, the work in [LHS20] estimates a median time for the diagnosis of partial failures to be 6 days and 5 hours. One of the main causes of late diagnosis is ambiguity of the symptoms and hence difficulty in correlating failures with their effects.

In this paper we make a first step towards a better understanding of the correlation between failures and symptoms via static formal analysis. We focus on the distributed actor model of Erlang [Arm13], which is known for its effectiveness in handling failures and has been emulated in many other languages, e.g., the popular Akka framework for Scala [Wya13].

We define a formal model of actor-based systems with grey failures, which we call *'cursed systems'*. More precisely, we introduce two interlinked models: (1) a *model of systems*, which are networks of distributed actors; (2) a *model of (grey) failures* that allows us to characterise *'curses'* as patterns of grey failures to inject in the system. This model of failures can represent node failures (with loss of messages in the node's mailbox), node slowdowns, link failures (with loss of the message in transit), and link slowdowns. The aforementioned instances of failure can be specified at the granularity of single nodes and links, to capture total and partial failures, and at the granularity of (discrete) time instants, to capture permanent, transient, and periodic failures. For example, a failed node can be in a failed state for a while before being restarted. The model of systems allows one to specify whether a node is restarted from the initial state (reset) or from a checkpoint. To capture the ambiguity of symptoms of grey failure we assume that actors have no knowledge of the state of health of other actors. However, actors can observe the presence (or absence) of messages in their own mailboxes and hence can infer the effects of failure from the communications that they have (not) received. In Erlang, a key mechanism for detecting and dealing with failure is the use of timeouts, which are one of the main ingredients of our system model.

Modelling failures as a separate layer allows us to compare systems recovery strategies with respect to specific failure patterns. This is a first step towards analysing the resilience of systems to failures, and assessing the effect of failure on different parts of the system. We introduce a behavioural equivalence, based on weak barbed bisimulation, to compare systems affected by failures. We show that reliability properties of interest, namely resilience and recoverability, can be reduced to the problem of checking weak barbed bisimulation between systems with failures. Furthermore, we introduce a notion of augmentation, based on weak barbed bisimulation, to model and analyse the improvement of a system with respect to its recoverability against certain kinds of failure.

Synopsis. The paper is structured as follows. In Section 2, we give an informal overview of the system model, and compare it with related work. Next we introduce the models of failure (Section 3) and systems (Section 4). In Section 5 we give a behavioural equivalence between

systems with failures, and show how it is used to model properties of interest. Section 7 describes prospective applications and promising directions of this work. Section 8 discusses conclusions and related work.

Extensions with respect to the conference paper. This work is an extended version of the conference paper appeared in COORDINATION 2022 [BLTV22] with the following additional contributions:

- The syntax and semantics of systems have been extended to model checkpoints. We have added Section 4.2 to show that the extension can still express the systems in [BLTV22] (Proposition 4.11). In a new section, Section 5.3, we show that the notion of n-recoverability given in [BLTV22] is not suitable for systems with checkpoints. We have therefore added, in Section 5.3, a more relaxed notion of n-recoverability for checkpointing systems.
- We found a flaw in the definition of n-recoverability given in [BLTV22]. We have fixed in this extended version. In Section 5.1 we give an amended definition of n-recoverability alongside the one given in [BLTV22], and discuss the differences by examples.
- In [BLTV22] we informally stated a relationship between two reliability properties we defined in that work: 'resilience is equivalent to 0-recoverability'. In a new section of this extended version, Section 5.2, we give the formal proof of this equivalence based on our amended definition of n-recoverability.
- The examples in Section 5 have been improved to reflect the feedback from the presentation of the conference. For instance, Example 5.6 has been framed to show the role of redundancy in fault-tolerance and how we can express it with our framework.
- Following feedback at the conference presentation, we have extended the section of related works. In particular, we have added a comparison with the works in [Gär99] and [DCMA17]. Due to the particular relevance of the work in [Gär99], we have also added a new subsection, Section 5.4, with a more technical discussion on how our work relates to the more general definitions of fault-tolerance given in [Gär99].
- We have added a Section 7 with a discussion of prospective applications of this work.

2. INFORMAL OVERVIEW

Actor-based systems are modelled using a process calculus with three key elements, following the actor model of Erlang: (1) time and timeouts, (2) asynchronous communication based on mailboxes with pattern-matching, and (3) actor nodes and injected failures.

Time and timeouts. Timeouts are essential for an actor to decide when to trigger a recovery action. Time is also crucial to observe the effects of failure patterns including quantified delays or down-times of nodes and links. We based our model of time on the Temporal Process Language (TPL) [HR95], a well understood extension of CCS with discrete time and timeouts. Delays are processes of the form **sleep**.P that behave as P after one time unit. Timeouts are modelled after the idiomatic `receive...after` pattern in Erlang. Concretely, the Erlang pattern below (left) is modelled as the process below (right):

```
receive
    Pattern1 -> P1;
    ...
    PatternN -> PN
after
    m -> Q
end
```

$?\{p_1.P_1, \ldots, p_N.P_N\} \texttt{ after } m \; Q$

where $p_1, \ldots, p_N$ is a set of patterns, each associated with a continuation P_i, with $i \in \{1, \ldots, N\}$, and Q is the timeout handler, executed if none of the patterns can be matched with a message in the mailbox within m time units. Following TPL, an action can be either a time action or an instantaneous communication action, and time actions can happen only when communication actions are not possible (maximal progress [HR95]). Concretely, we define the systems behaviour as a reduction relation with two kinds of actions: communication actions $\rightharpoonup$ and time actions $\rightsquigarrow$. While TPL is synchronous and only prioritises synchronisations over delays, we model *asynchronous* communications and prioritise any send or receive action over time actions. Thus, in our model, by maximal progress, communications have priority over delays.

The state of an actor at a time t is modelled as $\texttt{n}[\, P \,](M)(t)$, where $\texttt{n}$ is the actor identifier (unique in the system), M the mailbox, and P the process run by that actor. System $\mathbf{R}_t$ below is the parallel composition of actors $\texttt{n}_1$ and $\texttt{n}_2$:

$$\mathbf{R}_t = \texttt{n}_1[\, \mathbf{sleep}.!\texttt{n}_2\, \texttt{a}.0 \,](\emptyset)(t) \parallel \texttt{n}_2[\, ?\texttt{a}.P \texttt{ after } 1 \; Q \,](\emptyset)(t)$$

Although each actor in $\mathbf{R}_t$ has its own local time t explicitly represented, which makes it easy to inject failures compositionally, our semantics keeps the time of parallel components synchronized (as in TPL). In $\mathbf{R}_t$, node $\texttt{n}_1$ is deliberately idling and $\texttt{n}_2$ is temporarily blocked on a receive/timeout action, so no communication can happen, and thus only a time action is possible, updating both actors' times and triggering the timeout in $\texttt{n}_2$:

$$\mathbf{R}_t \rightsquigarrow \texttt{n}_1[\, !\texttt{n}_2\, \texttt{a}.0 \,](\emptyset)(t+1) \parallel \texttt{n}_2[\, Q \,](\emptyset)(t+1)$$

Mailboxes. Each pair of actors can communicate via two unidirectional links. For example, $(\texttt{n}_1, \texttt{n}_2)$ denotes the link for communications from $\texttt{n}_1$ to $\texttt{n}_2$. An interaction involves three steps: (I) the sending actor sends the message by placing it in the appropriate link, (II) the message reaches the receiver's mailbox, and (III) the receiving actor processes the message. These three steps allows us to capture e.g., effects of failures in senders versus receivers, on nodes versus links, and to model latency. Consider the system

$$\mathbf{R}_c = \texttt{n}_1[\, !\texttt{a}.0 \,](\emptyset)(t) \parallel \texttt{n}_2[\, ?\texttt{a}.P \texttt{ after } 2 \; Q \,](\texttt{b})(t)$$

Step (I), the sending of a message, is illustrated below on $\mathbf{R}_c$:

$$\mathbf{R}_c \rightharpoonup \texttt{n}_1[\, 0 \,](\emptyset)(t) \parallel 1.(\texttt{n}_1, \texttt{n}_2, \texttt{a}) \parallel \texttt{n}_2[\, ?\texttt{a}.P \texttt{ after } 2 \; Q \,](\emptyset)(t) = \mathbf{R}_c' \qquad (2.1)$$

$1.(\texttt{n}_1, \texttt{n}_2, \texttt{a})$ models a latent message in link $(\texttt{n}_1, \texttt{n}_2)$ with content $\texttt{a}$. Prefix 1 is the average network latency (assumed to be a constant). Due to latency, the message can only be added to the receiver's mailbox after one time step:

$$\mathbf{R}_c' \rightsquigarrow \texttt{n}_1[\, 0 \,](\emptyset)(t+1) \parallel (\texttt{n}_1, \texttt{n}_2, \texttt{a}) \parallel \texttt{n}_2[\, ?\texttt{a}.P \texttt{ after } 1 \; Q \,](\emptyset)(t+1) \qquad (2.2)$$

These floating messages $(\mathtt{n}_1, \mathtt{n}_2, \mathtt{a})$ with no latency are similar to messages in the ether [SFE10], in the global mailbox [LNPV18], or to the floating messages in [LSZ19].

Step (II) is the reception of the message, and happens as illustrated below (omitting the idle actor $\mathtt{n}_1$), where message $\mathtt{a}$ is added to the mailbox of $\mathtt{n}_2$:

$$(\mathtt{n}_1, \mathtt{n}_2, \mathtt{a}) \parallel \mathtt{n}_2[\,?\mathtt{a}.P \text{ after } 1 \text{ } Q\,](\emptyset)(t+1) \rightharpoonup \mathtt{n}_2[\,?\mathtt{a}.P \text{ after } 1 \text{ } Q\,](\mathtt{a})(t+1)$$

Step (III) is the processing of the message, as illustrated below:

$$\mathtt{n}_2[\,?\mathtt{a}.P \text{ after } 1 \text{ } Q\,](\mathtt{a})(t+1) \rightharpoonup \mathtt{n}_2[\,P\,](\emptyset)(t+1)$$

where message $\mathtt{a}$ in the mailbox matches the receive pattern (made up of a single atom $\mathtt{a}$) and is therefore processed. Mailboxes give us an expressive model of communication for modern real-world systems. An alternative model of communication is peer-to-peer communication, used e.g., in Communicating Finite State Machines (CFSM) [BZ83] and Multiparty Session Types [HYC16, CDYP16], where a receiver must specify from whom the message is expected. This makes it difficult to accurately capture interactions with public servers, or patterns like multiple producers-one consumer.

In the interaction above, note that $\mathtt{n}_2$ processes message $\mathtt{a}$ because it matches pattern $\mathtt{a}$; this would be the case even if there were an older message $\mathtt{b}$ in the mailbox, if that message did not match that pattern $\mathtt{a}$. Alternative models, like Mailbox CFSMs [BBO12, BGF$^+$21], typically do not model the selective receive pattern (e.g., pattern-matching in Erlang) shown above. Without selective receive, participants can easily get stuck if messages are received out of order. One can encode peer-to-peer communication over FIFO unidirectional channels by using pattern matching with selective receive: using the sender's identifier in the message and in the receive pattern. A similar communication model to ours was proposed in [MV11].

Localities and failures. The actor construct is similar to that used to model locality for processes [Cas01], and also studied in relation to failures [BH03, RH01, FH07, FH08] but using a fail-stop untimed model. We use actor nodes to model the effects of injected failures on specific nodes and links.

Referring to system $\mathbf{R}'_c$ in (2.1), by placing floating messages into a link with latency before they reach the receiver's mailbox we can observe the effects of link failure as message loss. Assume link $(\mathtt{n}_1, \mathtt{n}_2)$ is down at time t:

$$\mathbf{R}'_c \rightharpoonup \mathtt{n}_1[\,0\,](\emptyset)(t) \parallel \mathtt{n}_2[\,?\mathtt{a}.P \text{ after } 2 \text{ } Q\,](\emptyset)(t)$$

the floating message gets lost which in turn would end up causing a timeout in $\mathtt{n}_2$. Similarly, in the case of node failure, node $\mathtt{n}_1$ in system $\mathbf{R}_c$, seen earlier in (2.1), would go into a crashed node state before sending the message, hence triggering a timeout in $\mathtt{n}_2$:

$$\mathbf{R}_c \rightharpoonup \mathtt{n}_1[\,\downarrow\,](\emptyset)(t) \parallel \mathtt{n}_2[\,?\mathtt{a}.P \text{ after } 2 \text{ } Q\,](\emptyset)(t)$$

Assumptions. When a node crashes and comes back up again later on, it will come up with the same node identifier. This is consistent with Distributed Erlang, where by default all nodes are named; on the other hand, if we were resuscitating processes, we would need to name them for this to be possible. For simplicity, we assume nodes are not created at run-time, focusing on fixed topologies. Extending the language with the capability of creating new nodes is relatively straightforward, and can be done in a similar way to π-calculus restriction. We assume that behaviour within a node is sequential: actors can be composed in

parallel but processes cannot, hence limiting communication to distributed communications between nodes.

We choose to focus on inter-node communication on its own, because there already exist good strategies (e.g, in Erlang and Elixir) for dealing with in-node failure through the use of a supervision hierarchy, supervision strategies, and let-it-crash philosophy. Messages in transit when a node goes down remain in transit and may enter the mailbox after this node is resumed.

We allow a restricted (external) version of choice, based on the communication patterns found in Erlang. Free, or completely unrestricted choice, while central to many process algebras, for example CCS, tends to be less used in practice.

3. A model of failures

Let $\mathcal{N}$ be the set of node identifiers in a system. The model of failures is defined to be the Δ function:

$$\Delta : \mathbb{N} \times (\mathcal{N} \cup \mathcal{N} \times \mathcal{N}) \mapsto \{\downarrow, \uparrow, \circlearrowleft\}$$

mapping each discrete time $t \in \mathbb{N}$, node $\mathbf{n} \in \mathcal{N}$, and link $(\mathbf{n_1}, \mathbf{n_2}) \in \mathcal{N} \times \mathcal{N}$ to a value representing the state of health of that node or link, at that time. The symbol $\uparrow$ denotes the "healthy" state, $\downarrow$ identifies the failure of a node or link, and $\circlearrowleft$ indicates a node or link slowdown.

The failure scenarios covered by Δ include node crash, message loss, slow processes or slow networks. If *node* $\mathbf{n}$ *is down* at time t, written $\Delta(t)(\mathbf{n}) = \downarrow$, then it will perform no action until it is resumed, if ever. If $\mathbf{n}$ is resumed at time t', then its state at time t' will be set to the initial state (see Definition 4.6 for the formal definition). If *link* $(\mathbf{n_1}, \mathbf{n_2})$ *is down* at time t, written $\Delta(t)(\mathbf{n_1}, \mathbf{n_2}) = \downarrow$, then any message in transit on that link at time t will be lost. If *node* $\mathbf{n}$ *is slow* at time t, written $\Delta(t)(\mathbf{n}) = \circlearrowleft$, then any actions of the process running in $\mathbf{n}$ are delayed for one time step, and may resume at time $t + 1$ if $\Delta(t+1)(\mathbf{n}) = \uparrow$. If *link* $(\mathbf{n_1}, \mathbf{n_2})$ *is slow* at time t, written $\Delta(t)(\mathbf{n_1}, \mathbf{n_2}) = \circlearrowleft$, then the delivery of any message in transit on that link at time t will not happen at that time, and so will be delayed by at least one time unit. This delay is in addition to the network latency, which is modelled as a constant. Failures can be permanent or transient, as shown below by examples.

Example 3.1 (Permanent and transient failures)**.** Permanent node failure after a certain point in time, say $t = 10$, can be modelled by the definition Δ_1 below. Function Δ_2 shows a transient periodic structural failure of node $\mathbf{n}$, with each period having 100 time units of healthy state and 100 of down state.

$$\Delta_1(t)(\mathbf{n}) = \begin{cases} \uparrow & \textit{if } t < 10 \\ \downarrow & \textit{otherwise} \end{cases} \qquad \Delta_2(t)(\mathbf{n}) = \begin{cases} \uparrow & \textit{if } t \texttt{ div } 100 \texttt{ mod } 2 = 0 \\ \downarrow & \textit{otherwise} \end{cases}$$

One could similarly model transient degrading failure by setting uptimes when $t = n^2$ for $(n \in \mathbb{N})$.

Systems			Values			
$\mathbf{R}$	$::=$	$\mathtt{n}[\,P\,]_Q(M)(t)$ node	V	$::=$	a	atom
	$\mid$	$(\mathtt{n_1},\mathtt{n_2},m)(t)$ floating message		$\mid$	$\mathtt{n}$	node id
	$\mid$	$u.(\mathtt{n_1},\mathtt{n_2},m)(t)$ latent message		$\mid$	X	variable
	$\mid$	$\mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t)$ crashed node	Message			
	$\mid$	$\emptyset$ empty	m	$::=$	$\widetilde{V}$	message tuple
	$\mid$	$\mathbf{R}\,\|\,\mathbf{R}$ parallel				

Mailbox
$$M \quad ::= \quad \emptyset \;\mid\; M \cdot m$$

Processes				Receive Patterns			
P	$::=$	$!\{\mathtt{n_i}\, m_i.P_i\}_{i\in I}$	send				
	$\mid$	$?\{p_i.P_i\}_{i\in I}\,\mathtt{after}\,P$	receive-timeout	E	$::=$	$X \mid a$	pattern element
	$\mid$	$\mathbf{sleep}.P$	sleep	p	$::=$	$\widetilde{E}$	pattern tuple
	$\mid$	$\mathbf{save}.P$	check-point				
	$\mid$	$\mu \mathtt{t}.P$	fixed-point				
	$\mid$	$\mathtt{t}$	recursive variable				
	$\mid$	$\mathbf{0}$	inaction				

FIGURE 1. Syntax

[VAR1] $(X,a) \vdash_{\mathrm{match}} [a/X]$ [VAR2] $(X,\mathtt{n}) \vdash_{\mathrm{match}} [\mathtt{n}/X]$

$$[\textsc{Tuple}] \quad \frac{(E,V) \vdash_{\mathrm{match}} \underline{\sigma} \qquad (\widetilde{E},\widetilde{V}) \vdash_{\mathrm{match}} \sigma}{(E\widetilde{E}, V\widetilde{V}) \vdash_{\mathrm{match}} \underline{\sigma}\sigma}$$

[ATOM] $(a,a) \vdash_{\mathrm{match}} [a/a]$

$$[\textsc{Mbox1}] \quad \frac{(E,m) \vdash_{\mathrm{match}} \sigma}{(E, m\cdot M) \vdash_{\mathrm{match}} \sigma} \qquad\qquad [\textsc{Mbox2}] \quad \frac{(E,m) \nvdash_{\mathrm{match}} \qquad (E,M) \vdash_{\mathrm{match}} \sigma}{(E, m\cdot M) \vdash_{\mathrm{match}} \sigma}$$

FIGURE 2. Matching rules

4. CALCULUS FOR CURSED SYSTEMS

This section presents the model for actor based systems. The syntax of the calculus is given in Figure 1.

Systems are nodes $\mathtt{n}[\,P\,]_Q(M)(t)$, messages (floating or latent), crashed nodes $\mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t)$, empty systems $\emptyset$, and parallel compositions of systems $\mathbf{R}\,\|\,\mathbf{R}$. The term $\mathtt{n}[\,P\,](M)(t)$ denotes the state of node $\mathtt{n} \in \mathcal{N}$ at time t where P is the process running in $\mathtt{n}$, Q is the saved checkpoint process, and M is the mailbox of $\mathtt{n}$. A mailbox is a (possibly empty) list of messages. A message m is a tuple of values, which can be atoms a, node ids $\mathtt{n}$ or variables X. Messages are read from a mailbox via pattern matching.

We define the pattern matching function in the style of [MV11] through the derivations in Figure 2. Given a pattern $\widetilde{E}$ and a message (tuple) $\widetilde{V}$, $(\widetilde{E},\widetilde{V}) \vdash_{\mathrm{match}} \sigma$ the match function returns a substitution σ. Note that the match is only defined if $\widetilde{E}$ and $\widetilde{V}$ have the same size, and if the pattern and message match. We write $(E,m) \nvdash_{\mathrm{match}}$ when message m does *not*

match pattern E. Juxtaposition denotes concatenation of pattern and value tuples, and, since we assume that variables appear uniquely in pattern tuples, $\underline{\sigma}\sigma$ is the union of the two substitutions.

A floating message $(\mathtt{n_1}, \mathtt{n_2}, m)(t)$ represents a message m in link $(\mathtt{n_1}, \mathtt{n_2})$. Latent messages $u.(\mathtt{n_1}, \mathtt{n_2}, m)(t)$ are floating messages which can only reach the receiver's mailbox after a latency u. We assume all sent messages have a latency defined as a constant L, which abstracts the average network latency.

Looking at processes, a term of the form $!\{\mathtt{n_i}\, m_i.P_i\}_{i \in I}$ chooses to send to node $\mathtt{n_i}$ a message m_i and continues as P_i. Term $?\{p_i.P_i\}_{i \in I}\, \texttt{after}\, P$ tries to pattern match a message from the mailbox against one of the patterns p_i, and continues as P_i given that the matching succeeds for p_i, timing out $\texttt{after}$ one time unit if no message matches and executing P. Process $\mathbf{sleep}.P$ consumes a time unit and then continues as P. Process $\mathbf{save}.P$ saves the current state as a checkpoint process. Process $\mu\mathtt{t}.P$ is for recursion, and $\mathtt{t}$ is the recursive call. Finally, $\mathbf{0}$ is the idle process.

Remark 4.1. We use notation $?\{p_i.P_i\}_{i \in I}\, \texttt{after}\, u\, P$ as syntactic sugar for nesting u timeouts[1] and $\mathbf{sleep}\, u.P$ for the sequential composition of u delays with continuation P.

Recall (Section 3) that we fix the set of system's nodes $\mathcal{N}$, and the domain of Δ is $\mathcal{N} \cup (\mathcal{N} \times \mathcal{N})$, that is the set of nodes and links between pairs of nodes. Our unit of analysis is a *cursed system* defined below.

Definition 4.2 (Cursed system). A cursed system is a pair $(\mathbf{R}, \Delta)$ where $\mathbf{R}$ is a system, Δ is a curse.

The semantics of cursed systems is given in Def. 4.3 as a reduction relation over systems that is parametric on Δ. We write $\mathbf{R}_1 \equiv \mathbf{R}_2$ to mean that the systems $\mathbf{R}_1$ and $\mathbf{R}_2$ are the same up-to associativity and commutativity of $\|$, plus $0.(\mathtt{n_1}, \mathtt{n_2}, m)(t) \equiv (\mathtt{n_1}, \mathtt{n_2}, m)(t)$ and $\mathbf{R} \parallel \emptyset \equiv \mathbf{R}$.

Definition 4.3 (Operational semantics for cursed systems). Reduction is the smallest relation on cursed systems over communication actions denoted by $\rightharpoonup$, and time actions denoted by $\rightsquigarrow$, that satisfies the rules in Figure 3. We use $\rightarrow$ when $\rightarrow \in \{\rightharpoonup, \rightsquigarrow\}$. For readability, in the rules we assume Δ fixed and write $\mathbf{R} \rightarrow \mathbf{R}'$ instead of $(\mathbf{R}, \Delta) \rightarrow (\mathbf{R}', \Delta)$.

The first set of rules in Figure 3a is for actors actions, happening at a time t, when the nodes and links are in a healthy state i.e. $\Delta(t)(\mathtt{n}) =\uparrow$. In rule [Snd], $\mathtt{n}$ chooses to send a message m_j to node $\mathtt{n_j}$, and continues as P_j. Modelling asynchronous communication, a latent message $L.(\mathtt{n}, \mathtt{n_j}, m_j)(t)$ is introduced in the system, where L is the network latency constant. Rule [Sched] delivers a floating message to the receiver's mailbox. Rule [Rcv], retrieves the first message m in the mailbox that matches one of the receive patterns p_j. The match function returns a substitution σ that is applied to the continuation process P_j associated with pattern p_j; and m is removed from the mailbox. Rule [Checkpoint] saves the current state P as a checkpoint process for that node $\mathtt{n}$. Finally, Rule [Rec] allows a node with a recursive process to proceed with a communication or a time action.

---

[1]As $Q(u)$ where $Q(0) = ?\{p_i.P_i\}_{i \in I}\, \texttt{after}\, P$ and $Q(i+1) = ?\{p_i.P_i\}_{i \in I}\, \texttt{after}\, Q(i)$.

$$[\textsc{Snd}] \ \frac{\Delta(t)(\mathtt{n}) = \uparrow \qquad j \in I}{\mathtt{n}[\,!\{\mathtt{n_i}\,m_i.P_i\}_{i \in I}\,]_Q(M)(t) \rightharpoonup \mathtt{n}[\,P_j\,]_Q(M)(t) \,\|\, L.(\mathtt{n},\mathtt{n_j},m_j)(t)}$$

$$[\textsc{Sched}] \ \frac{\Delta(t)(\mathtt{n_1}) = \uparrow \qquad \Delta(t)(\mathtt{n_2},\mathtt{n_1}) = \uparrow}{(\mathtt{n_2},\mathtt{n_1},m)(t) \,\|\, \mathtt{n_1}[\,P\,]_Q(M)(t) \rightharpoonup \mathtt{n_1}[\,P\,]_Q(M \cdot m)(t)}$$

$$[\textsc{Rcv}] \ \frac{\Delta(t)(\mathtt{n}) = \uparrow \qquad j \in I,\ (p_j,m) \vdash_{\text{match}} \sigma \qquad \forall i \in I,\ (p_i, M_1) \nvdash_{\text{match}}}{\mathtt{n}[\,?\{p_i.P_i\}_{i \in I}\,\texttt{after}\,P\,]_Q(M_1 \cdot m \cdot M_2)(t) \rightharpoonup \mathtt{n}[\,P_j\sigma\,]_Q(M_1 \cdot M_2)(t)}$$

$$[\textsc{Checkpoint}] \ \frac{-}{\mathtt{n}[\,\mathbf{save}.P\,]_Q(M)(t) \rightharpoonup \mathtt{n}[\,P\,]_P(M)(t)}$$

$$[\textsc{Rec}] \ \frac{\Delta(t)(\mathtt{n}) = \uparrow \qquad \mathtt{n}[\,P[\mu\mathtt{t}.P/\mathtt{t}]\,]_Q(M)(t) \rightarrow \mathtt{n}[\,P'\,]_Q(M)(t')}{\mathtt{n}[\,\mu\mathtt{t}.P\,]_Q(M)(t) \rightarrow \mathtt{n}[\,P'\,]_Q(M)(t')}$$

(A) Actor/Node actions

$$[\textsc{Sleep}] \ \frac{\Delta(t)(\mathtt{n}) = \uparrow}{\mathtt{n}[\,\mathbf{sleep}.P\,]_Q(M)(t) \rightsquigarrow \mathtt{n}[\,P\,]_Q(M)(t+1)}$$

$$[\textsc{Latency}] \ \frac{\Delta(t)(\mathtt{n_1},\mathtt{n_2}) = \uparrow \qquad u > 0}{u.(\mathtt{n_1},\mathtt{n_2},m)(t) \rightsquigarrow (u-1).(\mathtt{n_1},\mathtt{n_2},m)(t+1)}$$

$$[\textsc{Timeout}] \ \frac{\Delta(t)(\mathtt{n}) = \uparrow \qquad \forall i \in I,\ (p_i, M) \nvdash_{\text{match}}}{\mathtt{n}[\,?\{p_i.P_i\}_{i \in I}\,\texttt{after}\,P\,]_Q(M)(t) \rightsquigarrow \mathtt{n}[\,P\,]_Q(M)(t+1)}$$

(B) Time actions

$$[\textsc{NLate}] \ \frac{\Delta(t)(\mathtt{n}) = \circlearrowright}{\mathtt{n}[\,P\,]_Q(M)(t) \rightsquigarrow \mathtt{n}[\,P\,]_Q(M)(t+1)} \qquad [\textsc{MsgLoss}] \ \frac{\Delta(t)(\mathtt{n_1},\mathtt{n_2}) = \downarrow \qquad u \geq 0}{u.(\mathtt{n_1},\mathtt{n_2},m)(t) \rightharpoonup \emptyset}$$

$$[\textsc{MsgLate}] \ \frac{\Delta(t)(\mathtt{n_1},\mathtt{n_2}) = \circlearrowright \qquad u \geq 0}{u.(\mathtt{n_1},\mathtt{n_2},m)(t) \rightsquigarrow u.(\mathtt{n_1},\mathtt{n_2},m)(t+1)} \qquad [\textsc{NDown}] \ \frac{\Delta(t)(\mathtt{n}) = \downarrow}{\mathtt{n}[\,P\,]_Q(M)(t) \rightharpoonup \mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t)}$$

$$[\textsc{DownLate}] \ \frac{\Delta(t)(\mathtt{n}) = \downarrow}{\mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t) \rightsquigarrow \mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t+1)} \qquad [\textsc{NUp}] \ \frac{\Delta(t)(\mathtt{n}) = \uparrow}{\mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t) \rightharpoonup \mathtt{n}[\,Q\,]_Q(\emptyset)(t)}$$

(C) Failure actions

$$[\textsc{Str}] \ \frac{\mathbf{R}_1 \equiv \mathbf{R}_1' \qquad \mathbf{R}_1 \rightarrow \mathbf{R}_2 \qquad \mathbf{R}_2 \equiv \mathbf{R}_2'}{\mathbf{R}_1' \rightarrow \mathbf{R}_2'} \qquad [\textsc{ParCom}] \ \frac{\mathbf{R}_1 \rightharpoonup \mathbf{R}_1'}{\mathbf{R}_1 \,\|\, \mathbf{R}_2 \rightharpoonup \mathbf{R}_1' \,\|\, \mathbf{R}_2}$$

$$[\textsc{ParTime}] \ \frac{\mathbf{R}_1 \rightsquigarrow \mathbf{R}_1' \quad \mathbf{R}_2 \rightsquigarrow \mathbf{R}_2' \quad \mathbf{R}_1 \,\|\, \mathbf{R}_2 \nrightarrow}{\mathbf{R}_1 \,\|\, \mathbf{R}_2 \rightsquigarrow \mathbf{R}_1' \,\|\, \mathbf{R}_2'}$$

(D) System actions

FIGURE 3. Reduction rules

Time actions. The second set of rules, in Figure 3b, is for time-passing reduction in absence of failures. Rules [SLEEP] and [TIMEOUT] model reduction of time consuming and receiving with timeout processes, respectively. Rule [TIMEOUT] can only be applied if none of the messages in the mailbox is matching any of the patterns $\{p_i\}_{i \in I}$ yielding an urgent receive semantics [Mur19] reflecting the receive primitive in Erlang. Rule [LATENCY] allows time passing for latent messages. Note that, by setting $u' = \mathtt{max}(u - 1, 0)$, if a receiver node crashes, all latent/floating messages remain in the link until the node is able to receive them, i.e. in a healthy state. We omit the rules for state-preserving time passing for idle nodes and $\mathtt{n}[\,\mathbf{0}\,](M)(t)$.

Failure actions. The third set of rules, in Figure 3c, models the effects of failures injected at time t. Rule [NLATE] models a delay, injected by $\Delta(t)(\mathtt{n}) = \circlearrowright$, in the execution of the process P in a node $\mathtt{n}$: a time unit elapses without any action in P. Rule [MSGLOSS] models a lossy link at time t, injected by $\Delta(t)(\mathtt{n_1}, \mathtt{n_2}) = {\downarrow}$, and permanently deletes a message $u.(\mathtt{n_1}, \mathtt{n_2}, m)(t)$ in transit. Rule [MSGLATE] models a slow link, injected by $\Delta(t)(\mathtt{n_1}, \mathtt{n_2}) = \circlearrowright$, by allowing time to pass but without decreasing the latency u of the message. Rule [NDOWN] models an instantaneous node that crash injected by $\Delta(t)(\mathtt{n}) = {\downarrow}$, and erases the process and mailbox of the node. Rule [DOWNLATE] allows time to pass for a crashed node. In rule [NUP] a crashed node is restarted with its saved checkpoint process Q and empty mailbox. Σ is a mapping from $\mathcal{N}$ to processes, that gives the initial process of each actor node. We assume that the node identifier is unchanged when restarting the node.

Runtime System actions. The last set of rules given in Figure 3d models system actions. In rule [PARCOM] a communication action of system part $\mathbf{R_1}$ is reflected in the composite system $\mathbf{R_1} \,\|\, \mathbf{R_2}$. In rule [PARTIME] time actions need to be reflected in all the parts of a system. A whole system can have a time action only if all parts of the system have no communication or failure actions to perform at the current time ($\mathbf{R}_i \nrightarrow$). [STR] is for communication and time actions of structurally equivalent systems.

4.1. **Basic properties of systems reductions.** In the remainder of this section we discuss two properties of cursed systems: time-coherence (the semantics keeps clocks synchronized) and non-Zenoness. We start by defining the time of a system. All definitions below apply straightforwardly to cursed systems by fixing a Δ.

Definition 4.4 (Time of a system). Let $\underline{t}$ range over $\mathbb{N} \cup \{*\}$. We define the synchronization (partial) function δ:

$$\delta(*, \underline{t}) = \delta(\underline{t}, *) = \underline{t} \quad \delta(*, *) = * \quad \delta(\underline{t}, \underline{t}) = \underline{t}$$

$\delta(\underline{t}_1, \underline{t}_2)$ returns a time or a wildcard $*$, and is undefined if $\underline{t}_1 \neq \underline{t}_2$ and neither $\underline{t}_1$ nor $\underline{t}_2$ is a wildcard. We define $\mathtt{time}(\mathbf{R})$ as a partial function over systems:

$$\mathtt{time}(\mathbf{R}) = \begin{cases} * & \mathbf{R} = \emptyset \\ t & \mathbf{R} = \mathtt{n}[\,P\,]_Q(M)(t) \text{ or } \mathbf{R} = \mathtt{n}[\,{\downarrow}\,]_Q(M)(t) \text{ or} \\ & \mathbf{R} = (\mathtt{n_1}, \mathtt{n_2}, m)(t) \text{ or } \mathbf{R} = u.(\mathtt{n_1}, \mathtt{n_2}, m)(t) \\ \delta(\mathtt{time}(\mathbf{R}_1), \mathtt{time}(\mathbf{R}_2)) & \mathbf{R} = \mathbf{R}_1 \,\|\, \mathbf{R}_2 \end{cases}$$

We can now define time-coherence of a system, holding when all its components have the same time.

Definition 4.5 (Time coherence). $\mathbf{R}$ is *time coherent* if $\mathtt{time}(\mathbf{R})$ is defined.

For example, system $\mathtt{n_1}[\,P\,]_{Q_1}(M)(t) \parallel (\mathtt{n_1}, \mathtt{n_2}, m)(t) \parallel \emptyset$ is time-coherent, while system $\mathtt{n_1}[\,P\,]_{Q_1}(M)(t) \parallel (\mathtt{n_1}, \mathtt{n_2}, m)(t+1) \parallel \emptyset$ is not.

The time function is also useful to characterise systems where all actors are coherently at time 0 and in their initial state.

Definition 4.6 (Initial system). Let Σ and Γ be mappings from $\mathcal{N}$ to processes such that $\Sigma(\mathtt{n})$ is the initial process of $\mathtt{n}$ and $\Gamma(\mathtt{n})$ is the initial checkpoint of $\mathtt{n}$. Note that by the definition of processes $\downarrow$ is not a process, and so nodes are never crashed in the initial state. A system $\mathbf{R}$ is initial if $\mathtt{time}(\mathbf{R}) = 0$ and

$$\mathbf{R} \equiv \mathtt{n_1}[\,\Sigma(\mathtt{n_1})\,]_{\Gamma(\mathtt{n_1})}(\emptyset)(0) \parallel \ldots \parallel \mathtt{n_m}[\,\Sigma(\mathtt{n_m})\,]_{\Gamma(\mathtt{n_m})}(\emptyset)(0)$$

with $\{1, \ldots, m\} = \mathcal{N}$. A cursed system $(\mathbf{R}, \Delta)$ is *initial* if $\mathbf{R}$ is initial.

We assume any system $\mathbf{R}$ to start off as initial and hence, by Prop. 4.8, to be time-coherent.

Next we show that the reduction over systems preserves time-coherence, hence all reachable systems are coherent.

Lemma 4.7 (Time-coherence invariant). *If $\mathbf{R}$ is time-coherent and $\mathbf{R} \rightarrow \mathbf{R}'$ then $\mathbf{R}'$ is time-coherent.*

The proof of the lemma is straightforward, by induction on the derivation. In fact, the only rule that updates the time of a parallel composition is [PARTIME] which requires time passing for all parallel processes. The fact that if $\mathbf{R}$ is initial then $time(\mathbf{R})$ is defined (as 0) yields the following property. We let $\rightarrow^*$ be the transitive closure of the reduction relation.

Proposition 4.8. *Let $\mathbf{R}$ be initial, if $\mathbf{R} \rightarrow^* \mathbf{R}'$ then $\mathbf{R}'$ is time-coherent.*

We assume any system $\mathbf{R}$ to start off as initial and hence, by Prop. 4.8, to be time-coherent.

Next, we give a desirable property for timed models: non-Zenoness. This prevents an infinite number of communication actions at any given time (Zeno behaviours). Besides yielding a more natural abstraction of a real world system, non-Zenoness simplifies analysis; for example, we can assume that the set of states reachable without time passing is finite. We start by defining a non-instantaneous process.

Definition 4.9 (Non-instantaneous process). We define function $\mathtt{ninst}(P)$ inductively as follows:

$$\mathtt{ninst}(P) = \begin{cases} \bigwedge_{i \in I} \mathtt{ninst}(P_i) & \text{if } P = !\{\mathtt{n_i}\,m_i.P_i\}_{i \in I} \text{ or } P = ?\{p_i.P_i\}_{i \in I}\,\mathtt{after}\,Q \\ \mathtt{ninst}(Q) & \text{if } P = \mu X.Q \\ \mathtt{true} & \text{if } P = \mathbf{sleep}\,.Q \\ \mathtt{false} & \text{if } P = X \text{ or } P = 0 \end{cases}$$

We say that P is non-instantaneous if $\mathtt{ninst}(P) = \mathtt{true}$. We say that $\mathbf{R}$ is non-instantaneous if all nodes in $\mathbf{R}$ run non-instantaneous processes.

Proposition 4.10 (Non-Zenoness). *Let $\mathbf{R}$ be non-instantaneous. If $\mathbf{R} \rightarrow^* \mathbf{R}'$ then there is a finite number of $\mathbf{R}''$ such that $\mathbf{R}' \rightharpoonup \mathbf{R}''$.*

The proof is straightforward by induction on the structure of $\mathbf{R}'$. Intuitively, any non-instantanous actor can only make a finite number of instantaneous actions at any given time, and hence at time $\mathtt{time}(\mathbf{R}')$. Hereafter we assume systems to be non-instantaneous, and hence non-Zeno.

4.2. **Reset vs Checkpointing Systems.** We call *reset systems* those systems obtained using the grammar for systems but without the save processes **save**.P, and where $\Gamma = \Sigma$. Reset systems model systems where each node reacts to (presumed) failure by restarting the execution from the initial state. More formally:

Proposition 4.11 (Reset systems)**.** *If* $\mathbf{R}$ *is reachable from an initial reset system then for all* $\mathtt{n}[\,P\,]_Q(M)(t)$ *and* $\mathbf{R}'$ *such that* $\mathbf{R} = \mathtt{n}[\,P\,]_Q(M)(t)\,||\,\mathbf{R}'$ *we have* $Q = \Sigma(\mathtt{n})$.

The property above is proved straightforwardly by coinduction, showing that having checkpoint $\Sigma(\mathtt{n})$ in all nodes is a property of initial reset systems and an invariant of reset systems preserved by reduction (by case analysis on the reduction rules).

Reset systems are common in Erlang: robustness is provided by a supervision hierarchy which explicitly describes the ways in which parts of the system are restarted when they or other parts fail. Restarts can escalate: if a component repeatedly restarts, then its parent process may itself have to be restarted.

While Erlang provides no explicit mechanism for checkpointing, it is possible to save state periodically using bulk storage known as ETS-tables. These provide global storage from which state can be retrieved, always assuming that the tables themselves are preserved. Disk-based ETS-tables (DETS-tables) provide more permanent storage, but with an associated time cost.

In fact, in the short version of this article [BLTV22] we focussed on a formalisms that corresponds to reset systems. Here, we explore a more general setting, to show a more interesting relationship of our work to the ones in [Gär99][DCMA17], and particularly to the notion of *non-masking* fault-tolerance therein.

5. PROPERTIES OF CURSED SYSTEMS

In this section we define a behavioural relation between cursed systems, as a weak barbed bisimulation, which is the standard choice since we have a reduction semantics [SW01]. The aim is to compare the systems' abilities to preserve 'normal' functionality when they are affected by failures. We abstract from the fact that some parts of the system may be deadlocked, as long as healthy actors can keep receiving the messages they expect. Mailbox-based (rather than point-to-point) communication and pattern matching allow us to capture e.g., multiple-producer scenarios where a consumer can receive the expected feeds as long as *some* producers are healthy.

Our behavioural relation also abstracts from time, to disregard the delays introduced by recovering actions, and only observes the effects of such delays (we do not focus on efficiency). Essentially, two systems are equivalent when actors receive the same messages,

abstracting from senders, in a time-abstract way.[2] On the basis of this equivalence we define *recoverability* and *augmentation*.

We start by defining weak barbed simulation for cursed systems.

Definition 5.1 (Barb). The ready actions of P are defined inductively as follows:

$$\mathtt{rdy}(!\{\mathtt{n_i}\, m_i.P_i\}_{i\in I}) = \{!\,\mathtt{n}_i\, m_i\}_{i\in I} \qquad \mathtt{rdy}(?\{p_i.P_i\}_{i\in I}\ \mathtt{after}\ P) = \{?\,p_i\}_{i\in I}$$

$$\mathtt{rdy}(0) = \mathtt{rdy}(\mathtt{t}) = \mathtt{rdy}(\mathbf{sleep}.P) = \emptyset \qquad \mathtt{rdy}(\mu\mathtt{t}.P) = \mathtt{rdy}(P)$$

Let $\mathbf{R} \downarrow x$ be the least relation satisfying the rules below.

$$\mathtt{n}[\,P\,](M)(t) \downarrow x \qquad \text{if } !\,\mathtt{n}'m \in \mathtt{rdy}(P) \wedge x = !\,\mathtt{n}'m\ \vee\ ?\,p \in \mathtt{rdy}(P) \wedge x = ?\,\mathtt{n}\,p$$

$$(\mathtt{n}_1, \mathtt{n}_2, m) \downarrow\ !\,\mathtt{n}_2\, m$$

$$(\mathbf{R}_1 \parallel \mathbf{R}_2) \downarrow x \qquad \text{if } \mathbf{R}_1 \downarrow x \text{ or } \mathbf{R}_2 \downarrow x$$

If $\mathbf{R} \downarrow x$ we say that $\mathbf{R}$ has a *barb* on x.

Barbs abstract from (i.e., do not include in the model of observation) the sender of a message. This allows us to disregard the identity of the senders, following mailbox-based communications in actor-based systems. Scenarios where the identity of the sender is important can be encoded by using node identifiers as message content.[3] We observe m and p to retain expressiveness with respect to channel-based scenarios, as discussed in Section 6.1.

Example 5.2 (Examples on barbs). Consider a system $\mathbf{R}_R$ with a consumer node $\mathtt{c}$ receiving data $\mathtt{d}$ from two replicas $\mathtt{r1}$ and $\mathtt{r2}$. If the messages from both replicas are delayed then the consumer notifies a monitor node $\mathtt{m}$ (omitted here for simplicity):

$$\mathbf{R}_R = \mathtt{c}[\,\mu\mathtt{t}.?\mathtt{d.t.}\ \mathtt{after}\ 2\ !\mathtt{m}\,\mathtt{fail}\,](\emptyset)(0) \parallel \mathtt{r1}[\,\mu\mathtt{t}.!\mathtt{c}\,\mathtt{d.t}\,](\emptyset)(0) \parallel \mathtt{r2}[\,\mu\mathtt{t}.!\mathtt{c}\,\mathtt{d.t}\,](\emptyset)(0)$$

Regarding our choice of barbs in this example, the consumer needs to receive regular feeds $\mathtt{d}$, no matter whether they are from $\mathtt{r1}$ or $\mathtt{r2}$. Abstracting away from the identity of the sending replica is directly captured by our definition of barbs. In fact, the set of barbs of $\mathbf{R}_R$ is $\{!\,\mathtt{c}\,\mathtt{d}, ?\,\mathtt{c}\,\mathtt{d}\}$.

A system defined in the same way as $\mathbf{R}_R$ but with only one replica, e.g. obtained by removing node $\mathtt{r2}$, or with one of the replicas down, e.g. obtained by substituting node $\mathtt{r2}$ with $\mathtt{r2}[\downarrow](\emptyset)(0)$, would have the same set of barbs as $\mathbf{R}_R$, namely $\{!\,\mathtt{c}\,\mathtt{d}, ?\,\mathtt{c}\,\mathtt{d}\}$.

It is worth noting that if the identity of the sender does matter, it can be observed by encoding the identity into the messages being sent by the sender:

$$\mathbf{R}'_R =\ \ \mathtt{c}[\,\mu\mathtt{t}.\{?\mathtt{n1}\,\mathtt{d.t.}, ?\mathtt{n2}\,\mathtt{d.t.}\ \mathtt{after}\ 2\ !\mathtt{m}\,\mathtt{fail}\,](\emptyset)(0) \parallel$$
$$\mathtt{r1}[\,\mu\mathtt{t}.!\mathtt{c}\,\mathtt{n1}\,\mathtt{d.t}\,](\emptyset)(0) \parallel \mathtt{r2}[\,\mu\mathtt{t}.!\mathtt{c}\,\mathtt{n2}\,\mathtt{d.t}\,](\emptyset)(0)$$

The set of barbs of $\mathbf{R}'_R$ is $\{!\,\mathtt{c}\,\mathtt{n1}\,\mathtt{d}, !\,\mathtt{c}\,\mathtt{n2}\,\mathtt{d}, ?\,\mathtt{c}\,\mathtt{n1}\,\mathtt{d}, ?\,\mathtt{c}\,\mathtt{n2}\,\mathtt{d}\}$. If node $\mathtt{r2}$ was removed or crashed, the set of barbs would be affected, becoming $\{!\,\mathtt{c}\,\mathtt{n1}\,\mathtt{d}, ?\,\mathtt{c}\,\mathtt{n1}\,\mathtt{d}, ?\,\mathtt{c}\,\mathtt{n2}\,\mathtt{d}\}$, and making it possible to distinguish among senders.

---

[2] Abstracting from timing and message senders is an assumption of our model that we adopted for the sake of generality: it allows us to capture scenarios where the timing and order of the messages does not matter (e.g., multiple producers). On the other hand the model can encode scenarios where such orders matter. It can, for example, support Erlang-style actor behaviour. Erlang does not guarantee temporal order of messages between different processes in general, however between any two processes it does guarantee that messages sent directly between them will be received in the same order. Erlang behaviour can be encoded in our model if messages are extended to include the identity of the sender and a counter (e.g., as atoms) to guarantee message origin and ordering.

[3] This is precisely how sender information is communicated in Erlang.

Definition 5.3 (Weak barbed simulation). Recall $\rightarrow \in \{\twoheadrightarrow, \rightsquigarrow\}$. A weak (time-abstract) barbed simulation is a binary relation $\mathcal{S}$ between cursed systems such that $(\mathbf{R}_1, \Delta_1) \, \mathcal{S} (\mathbf{R}_2, \Delta_2)$ implies:

(1) If $(\mathbf{R}_1, \Delta_1) \rightarrow (\mathbf{R}'_1, \Delta_1)$ then there exists $\mathbf{R}'_2$ such that $(\mathbf{R}_2, \Delta_2) \rightarrow^* (\mathbf{R}'_2, \Delta_2)$ and $(\mathbf{R}'_1, \Delta_1) \, \mathcal{S} (\mathbf{R}'_2, \Delta_2)$.

(2) If $\mathbf{R}_1 \downarrow x$ for some x, then there exists $\mathbf{R}'_2$ such that $(\mathbf{R}_2, \Delta_2) \rightarrow^* (\mathbf{R}'_2, \Delta_2)$ and $\mathbf{R}'_2 \downarrow x$.

We say $(\mathbf{R}_1, \Delta_1)$ is weak barbed similar to $(\mathbf{R}_2, \Delta_2)$, written $(\mathbf{R}_1, \Delta_1) \lesssim (\mathbf{R}_2, \Delta_2)$, if there exists some weak barbed simulation $\mathcal{S}$ such that $(\mathbf{R}_1, \Delta_1) \, \mathcal{S} (\mathbf{R}_2, \Delta_2)$.

Definition 5.4 (Weak barbed bisimulation). We say that $\mathcal{S}$ is a weak barbed bisimulation if $\mathcal{S}$ and $\mathcal{S}^{-1}$ are weak barbed simulations. We say $(\mathbf{R}_1, \Delta_1)$ is weak barbed bisimilar to $(\mathbf{R}_2, \Delta_2)$, written $(\mathbf{R}_1, \Delta_1) \approx (\mathbf{R}_2, \Delta_2)$, if there exists some weak barbed bisimulation $\mathcal{S}$ such that $(\mathbf{R}_1, \Delta_1) \, \mathcal{S} (\mathbf{R}_2, \Delta_2)$.

The rules in Figure 3 embody mailbox-based communication. These rules allow us to observe messages 'in flight', and so allow us to observe that some such messages have been affected by a curse, reflecting an insecure communication medium. Note, however, that it is *not* possible to directly observe the contents of mailboxes; this can only be done indirectly – and in general, partially – by observing the behaviour of receive statements.

If we fix a system, we can use Definition 5.3 to compare behaviours of that system with different curses, namely to determine when a system maintains its (desirable) behaviour even when cursed. Namely, Definition 5.3 provides a means to study the ability of a systems to tackle failure, or its *resilience*. Formally (Definition 5.5) we define resilience as the ability of a system to behave 'normally' despite failure injection. In the following, we let $\uparrow$ be the curse function that assigns $\uparrow$ to all nodes and links at all times.

Definition 5.5 (Resilience). Initial $(\mathbf{R}, \Delta)$ is resilient if $(\mathbf{R}, \uparrow) \approx (\mathbf{R}, \Delta)$.

Example 5.6 (Resilience). Consider the system $\mathbf{R}$ below and curse Δ for which (p, c) is down at time 1

$$\mathbf{R} = \mathsf{p}[\,\mathbf{sleep}\,.!\mathsf{c}\,\mathtt{item}.0\,](\emptyset)(0) \parallel \mathsf{c}[\,?\mathtt{item}.0\,\mathtt{after}\ 3\ 0\,](\emptyset)(0)$$

Fix the latency constant as 1 time unit. System $\mathbf{R}$ is not resilient with respect to Δ since $(\mathbf{R}, \Delta) \not\lesssim (\mathbf{R}, \uparrow)$. Intuitively, observe that $(\mathbf{R}, \uparrow)$ reaches the terminated state

$$(\mathsf{p}[\,0\,](\emptyset)(2) \parallel \mathsf{c}[\,?\mathtt{item}.0\,\mathtt{after}\ 1\ 0\,](\mathtt{item})(2), \uparrow)$$

with no barbs, whereas $(\mathbf{R}, \Delta)$ finally gets stuck in the state

$$(\mathsf{p}[\,0\,](\emptyset)(3) \parallel \mathsf{c}[\,0\,](\emptyset)(3) \parallel (\mathsf{p}, \mathsf{c}, \mathtt{item}), \Delta)$$

with an orphan message and barb $?\mathsf{c}\,\mathtt{item}$.

Different strategies can be applied to modify $\mathbf{R}$ so that it correctly handles the delays specified by Δ. One is to tune the timeouts in the code so that it can handle the curse Δ. Concretely, consider the following variant $\mathbf{R}'$ of $\mathbf{R}$, that increases the timeout value of one unit:

$$\mathbf{R}' = \mathsf{p}[\,\mathbf{sleep}\,.!\mathsf{c}\,\mathtt{item}.0\,](\emptyset)(0) \parallel \mathsf{c}[\,?\mathtt{item}.0\,\mathtt{after}\ 4\ 0\,](\emptyset)(0)$$

One can verify that $(\mathbf{R}', \Delta) \approx (\mathbf{R}', \uparrow)$ and hence $\mathbf{R}'$ is resilient with respect to Δ.

Example 5.6 shows a non-resilient cursed system $(\mathbf{R}, \Delta)$ and a resilient variant $(\mathbf{R}', \Delta)$ obtained by tuning the timeout in $\mathbf{R}$. In the following Example 5.7 we provide two additional resilient variants of $(\mathbf{R}, \Delta)$ obtained using time-redundancy (e.g., retry strategies) and space-redundancy (e.g., replication). Redundancy has been shown [Gär99] to be a necessary condition for fault-tolerance. Resilience gives a tool to assess whether a 'redundant' system is indeed attaining the intended fault-tolerance.

Example 5.7 (Resilience and redundancy)**.** Consider $(\mathbf{R}, \Delta)$ from Example 5.6 and, again, fix the latency constant as 1 time unit. We define a variant of $\mathbf{R}$, called $\mathbf{R}_T$, where time-redundancy is attained by retrying the communication once more in case of timeout

$$\mathbf{R}_T = \mathtt{p}[\,\mathbf{sleep}\,.!\mathtt{c}\,\mathtt{item}.0\,](\emptyset)(0) \parallel \mathtt{c}[\,?\mathtt{item}.0\ \mathtt{after}\ 3\ (?\mathtt{item}.0\ \mathtt{after}\ 3\ 0)\,](\emptyset)(0)$$

Similarly, we define a variant of $\mathbf{R}$, called $\mathbf{R}_S$, where space-redundancy is applied by adding an extra producer:

$$\begin{aligned}\mathbf{R}_S = \ &\mathtt{p}[\,\mathbf{sleep}\,.!\mathtt{c}\,\mathtt{item}.0\,](\emptyset)(0) \parallel \mathtt{c}[\,?\mathtt{item}.0\ \mathtt{after}\ 3\ 0\,](\emptyset)(0)\\ &\parallel \mathtt{p}'[\,\mathbf{sleep}\,.!\mathtt{c}\,\mathtt{item}.0\,](\emptyset)(0)\end{aligned}$$

One can verify that both $\mathbf{R}_T$ and $\mathbf{R}_S$ are resilient with respect to Δ from Example 5.6.

Our definition of resilience sets the behaviour of a system without curses as a model of *expected behaviour*. By Definition 5.5, any deviation from the expected behaviour, even a temporary one, makes a system non-resilient. This is a very strict characterization of fault-tolerance. For example, resilience is too strong to capture the effects of more complex retry-strategies than those applied in $\mathbf{R}_T$ from Example 5.7, as shown in the Example 5.8 below.

Example 5.8 (Resilience and more complex retry strategies)**.** Consider Δ from Example 5.6, latency of 1 time unit, and a variant $\mathbf{R}_{TT}$ of $\mathbf{R}_T$, where time-redundancy affects both processes:

$$\begin{aligned}\mathbf{R}_{TT} = \ &\mathtt{p}[\,\mu X.\,\mathbf{sleep}\,.!\mathtt{c}\,\mathtt{item}.\,?\{\mathtt{ok}\,.0,\ \mathtt{retry}.X\}\,](\emptyset)(0) \parallel\\ &\mathtt{c}[\,\mu X.\,?\mathtt{item}.\,!\mathtt{p}\,\mathtt{ok}.\,0\ \mathtt{after}\ 3\ (!\mathtt{p}\,\mathtt{retry}.\,X)\,](\emptyset)(0)\end{aligned}$$

System $\mathbf{R}_{TT}$ is not resilient with respect to Δ from Example 5.6 because the nodes add some communications to acknowledge correct interaction or coordinate on a retry iteration.

In the remaining of this section, we study a less restrictive characterization than resilience, which we call recoverability, to allow for some deviation from the expected behaviour as long as the system *eventually* resumes the expected behaviour. In Section 5.1 we discuss recoverability. In Section 5.2 we show a relation between resilience and recoverability. In Section 5.3 we provide a more general account of reliability that can easily capture reset and checkpointing systems. Section 5.1 is based on the notion of n-recoverability first introduced in [BLTV22], which is fixed and improved. Section 5.3 is new.

5.1. **Recoverability for reset-systems.** We define n-recoverability as the ability of a system to display the expected behaviour after time n. The definition from [BLTV22] had several issues that we have amended in this work. The original definition is as follows:

Definition 5.9 (n-Recoverability (from [BLTV22]))**.** Let $n \in \mathbb{N}$ and $(\mathbf{R}, \Delta)$ initial. $(\mathbf{R}, \Delta)$ is *n-recoverable* if $(\mathbf{R}, \Delta) \rightarrow^* (\mathbf{R}', \Delta)$ and $\mathtt{time}(\mathbf{R}') = n$, implies $(\mathbf{R}, \uparrow) \approx (\mathbf{R}', \Delta)$.

Example 5.10 (Counterexample). Fix a latency of 1 time unit and a generic Δ that does not affect any node or link at time 0. System $\mathbf{R}_{ce}$ below reduces to $\mathbf{R}'_{ce}$ after a communication action

$$\mathbf{R}_{ce} = \mathtt{n1}[\, !\mathtt{n2}\,\mathtt{a}.0\,](\emptyset)(0) \parallel \mathtt{n2}[\,\mathbf{sleep}.?\mathtt{a}.0\,](\emptyset)(0)$$
$$\mathbf{R}'_{ce} = \mathtt{n1}[\,0\,](\emptyset)(0) \parallel \mathtt{n2}[\,\mathbf{sleep}.?\mathtt{a}.0\,](\emptyset)(0) \parallel \mathbf{1}.(\mathtt{n}_1, \mathtt{n}_2, \mathtt{a})(0)$$

For $\mathbf{R}_{ce}$ to be 0-recoverable, since $(\mathbf{R}_{ce}, \Delta) \rightharpoonup (\mathbf{R}'_{ce}, \Delta)$, one should have $(\mathbf{R}'_{ce}, \Delta) \approx (\mathbf{R}_{ce}, \uparrow)$ which does not hold for any Δ because of a difference in barbs– hence not even for $\Delta = \uparrow$.

Example 5.10 shows that Definition 5.9 is too strict to capture the intended meaning of n-recoverability. In [BLTV22] for example, 0-recoverability is (wrongly) set to correspond to resilience. Definition 5.9 requires that *all* states at time n are bisimilar to the initial state, and this is too strict since several actions may naturally happen in a time unit.

We provide a weaker definition of n-recoverability, using universal quantification over *paths* of actions at time n and existential quantification on the states on each of these paths, which is set to better represent the intuition. First, we define the concept of n-entry, which is the set of states that are, for some execution, the first state to be reached at time n. Then a n-path is the maximal path from a n-entry where states are at time n.

Definition 5.11 (n-Entry). Let $n \in \mathbb{N}$ and $(\mathbf{R}_0, \Delta)$ be an initial state. If $n = 0$ then (R_0, Δ) is the only 0-entry for itself. If $n > 0$, $(\mathbf{R}, \Delta)$ is a n-entry for $(\mathbf{R}_0, \Delta)$ if there exists an execution $(\mathbf{R}_0, \Delta) \rightarrow^* (\mathbf{R}', \Delta) \rightsquigarrow (\mathbf{R}, \Delta)$ with $\mathtt{time}(\mathbf{R}) = n$.

A n-entry $(\mathbf{R}, \Delta)$ is the first state to be reached at time n. Observe that in Definition 5.11 if $n > 0$ then it is always the case that $\mathtt{time}(\mathbf{R}') = n - 1$. We define an *execution* $(\mathbf{R}_1, \Delta) \rightharpoonup^* (\mathbf{R}_m, \Delta)$ to be a sequence of configurations $(\mathbf{R}_i, \Delta)$, with $1 \leq i \leq m - 1$ such that $(\mathbf{R}_i, \Delta) \rightharpoonup (\mathbf{R}_{i+1}, \Delta)$.

Definition 5.12 (n-Path). Let $n \in \mathbb{N}$ and $(\mathbf{R}_0, \Delta)$ be an initial state. Execution $(\mathbf{R}_1, \Delta) \rightharpoonup^*$ $(\mathbf{R}_m, \Delta)$ is a n-path for $(\mathbf{R}_0, \Delta)$ if: (1) $(\mathbf{R}_1, \Delta)$ is a n-entry for $(\mathbf{R}_0, \Delta)$, and (2) $(\mathbf{R}_m, \Delta)$ cannot make other actions than time actions.

Observe that in Definition 5.12 a n-path $(\mathbf{R}_1, \Delta) \rightharpoonup^* (\mathbf{R}_m, \Delta)$ includes no time actions.

Definition 5.13 (n-Recoverability (new)). Let $n \in \mathbb{N}$ and $(\mathbf{R}_0, \Delta)$ be an initial state. $(\mathbf{R}_0, \Delta)$ is n-recoverable if for all of its n-paths $(\mathbf{R}_1, \Delta) \rightharpoonup^* (\mathbf{R}_m, \Delta)$ there is $i \in \{1, \ldots, m\}$ such that $(\mathbf{R}_i, \Delta) \approx (\mathbf{R}_0, \uparrow)$.

Definition 5.13 says that in any arbitrary n-path there exists a state $(\mathbf{R}_i, \Delta)$ reachable at time n that is weak-barbed bisimilar to $(\mathbf{R}_0, \uparrow)$.

Example 5.14 (n-Recoverability). Consider the system $\mathbf{R}$ below (and any Δ that does not affect the system at times 0 and 1):

$$\mathbf{R} = \mathtt{n}_1[\, !\mathtt{n}_2\,\mathtt{a}.\emptyset\,](\emptyset)(0) \parallel \mathtt{n}_2[\,?\mathtt{a}.\emptyset\,](\emptyset)(0)$$

$(\mathbf{R}, \Delta)$ reduces to the successfully terminated system $(\mathbf{R}', \Delta)$ with

$$\mathbf{R}' = \mathtt{n}_1[\,\emptyset\,](\emptyset)(1) \parallel \mathtt{n}_2[\,\emptyset\,](\emptyset)(1)$$

at time zero.

Example 5.15 (n-recoverability and more complex retry strategies). Consider Δ from Example 5.7, latency of 1 time unit, and a variant $\mathbf{R}_{TT}$ of $\mathbf{R}_T$, where time-redundancy

affects both processes:

$$\begin{aligned} \mathbf{R}_{TT} = \quad &\mathtt{p}[\,\mu X.\,\mathbf{sleep}\,.!\mathtt{c}\,\mathtt{item}.\,?\{\mathtt{ok}\,.0,\,\mathtt{retry}.X\}\,\mathtt{after}\ 5\ 0\,](\emptyset)(0)\ \| \\ &\mathtt{c}[\,\mu X.\,?\mathtt{item}.!\mathtt{p}\,\mathtt{ok}.0\,\mathtt{after}\ 4\ (!\mathtt{p}\,\mathtt{retry}.\,X)\,](\emptyset)(0) \end{aligned}$$

System $\mathbf{R}_{TT}$ is not resilient with respect to Δ from Example 5.6 because the nodes add some communications to acknowledge correct interaction or coordinate on a retry iteration. It is however n-recoverable with $n = 6$.

By Definition 5.13, checking resilience and n-recoverability is reduced to the problem of checking weak barbed bisimulation. Note that, in Definition 5.13, the number of $\mathbf{R}'$ that can be reached from $\mathbf{R}$ is finite, because the execution up to $\mathbf{R}'$ lasts for n time units and, by Proposition 4.10, a system can perform only a finite number of actions in a finite amount of time.

In the following, we show that resilience is equivalent to 0-recoverability. This fact was conjectured for not formally proven in [BLTV22]. This result is given in Section 5.2.

5.2. Resilience is equivalent to 0-recoverability.

Equivalence of resilience and 0-recoverability (Theorem 5.22) is based on two facts:

(1) for all initial systems $\mathbf{R}$, $(\mathbf{R}, \Delta) \gtrsim (\mathbf{R}, \uparrow)$, given in Lemma 5.19, and
(2) for all initial and 0-recoverable systems $\mathbf{R}$, $(\mathbf{R}, \Delta) \lesssim (\mathbf{R}, \uparrow)$, given directly in Theorem 5.22.

Lemma 5.19 is based on a property that we call $\uparrow$-consistency. $\uparrow$-consistency correlates the syntactic structure cursed system and their corresponding counterparts with curse $\uparrow$, as they evolve.

Definition 5.16 ($\uparrow$-consistency)**.** Two systems $\mathbf{R}_\Delta$ and $\mathbf{R}_\uparrow$ are $\uparrow$-*consistent* if there exist $\mathbf{R}$, $\mathbf{R}_u$, $\mathbf{R}_d$, and $\mathbf{R}_f$ such that

$$\mathbf{R}_\uparrow = \mathbf{R} \,\|\, \mathbf{R}_u \,\|\, \mathbf{R}_f \qquad \mathbf{R}_\Delta = \mathbf{R} \,\|\, \mathbf{R}_d$$

and:

- $\mathbf{R}_u$ and $\mathbf{R}_d$ are parallel compositions of the same (possibly empty) set of nodes.
- the nodes in $\mathbf{R}_d$ are all down, i.e., of the form $\mathtt{n}[\,\downarrow\,]_Q(\emptyset)(t)$,
- $\mathbf{R}_f$ is the parallel composition of a (possibly empty) set of latent or floating messages.

Intuitively, $\uparrow$-consistency defines a structural relation between the evolution of a system with and without curses: $\mathbf{R}$ models the parts of the system (if any) that $\mathbf{R}_\uparrow$ and $\mathbf{R}_\Delta$ have in common; $\mathbf{R}_u$ and $\mathbf{R}_d$ are the nodes that are up in $\mathbf{R}_\uparrow$ and down in $\mathbf{R}_\Delta$, respectively (they model the difference between $R_\uparrow$ and R_Δ wrt. crashed nodes); moreover, $\mathbf{R}_\uparrow$ may have some additional floating messages, represented by $\mathbf{R}_f$, that have been lost in $\mathbf{R}_\Delta$.

$\uparrow$-consistency enjoys two properties. The first one, given in Lemma 5.17, is that instantaneous actions preserve $\uparrow$-consistency and does not decrease the number of down nodes in the cursed system. The second one, given in Lemma 5.18, ensures that the barbs of the cursed system are always a subset of those of the uncursed counterpart.

Lemma 5.17. *If* $\mathbf{R}_\Delta$ *and* $\mathbf{R}_\uparrow$ *are* $\uparrow$-consistent *and* $(\mathbf{R}_\Delta, \Delta) \rightharpoonup (\mathbf{R}'_\Delta, \Delta)$ *then*

(1) *there exists* $\mathbf{R}'_\uparrow$ *such that* $(\mathbf{R}_\uparrow, \uparrow) \rightharpoonup (\mathbf{R}'_\uparrow, \uparrow)$ *and* $\mathbf{R}'_\Delta$ *and* $\mathbf{R}'_\uparrow$ *are* $\uparrow$-consistent
(2) *the set of down nodes in* $\mathbf{R}_\Delta$ *is a subset of the set of down nodes in* $\mathbf{R}_\uparrow$.

Proof. By induction on the derivation. In case of actions by [Snd], [Sched], [Rcv], and [Checkpoint] all yield that there is $\mathbf{R}'$ such that $\mathbf{R}'_\uparrow = \mathbf{R}' \| \mathbf{R}_u \| \mathbf{R}_f$ and $\mathbf{R}'_\Delta = \mathbf{R}' \| \mathbf{R}_d$. Moreover, no down nodes are introduced in $\mathbf{R}'_\Delta$ and hence $\mathbf{R}'_\Delta$ and $\mathbf{R}'_\uparrow$ are $\uparrow$-*consistent*. Rule [NUp] cannot be applied at time 0. The only possible failure actions are [MsgDown] and [NDown]. In case for [MsgDown], there exist $\mathbf{R}'$ and $\mathbf{R}'_f$ such that $\mathbf{R}'_\uparrow = \mathbf{R}' \| \mathbf{R}_u \| \mathbf{R}'_f$ and $\mathbf{R}'_\Delta = \mathbf{R}' \| \mathbf{R}_d$ where $\mathbf{R}'$ is as $\mathbf{R}$ but without the lost message, and $\mathbf{R}'_f$ is as $\mathbf{R}_f$ but with the addition of the lost message. The set of down nodes does not change, hence $\mathbf{R}'_\Delta$ and $\mathbf{R}'_\uparrow$ are $\uparrow$-consistent, yielding the thesis for this case. In case of [NDown], $\mathbf{R}'_\uparrow$ is of the form $\mathbf{R}' \| \mathbf{R}'_u \| \mathbf{R}_f$ and $\mathbf{R}'_\uparrow$ is for the form $\mathbf{R}' \| \mathbf{R}'_d$ where $\mathbf{R}'$ is as $\mathbf{R}$ but without the node that went down, $\mathbf{R}'_d$ is as $\mathbf{R}_d$ but with the addition in parallel with the node that went down, similarly for $\mathbf{R}'_u$ but the node in this case is still up. The set of down nodes has increased in $\mathbf{R}'_\Delta$. It follows that $\mathbf{R}'_\Delta$ and $\mathbf{R}'_\uparrow$ are $\uparrow$-*consistent*.

The cases for [Rec], [Str], and [ParCom] are immediate by induction. □

Lemma 5.18 ($\uparrow$-consistency). *If $\mathbf{R}_\Delta$ and $\mathbf{R}_\uparrow$ are $\uparrow$-consistent then $\mathbf{R}_\Delta \downarrow x$ implies $\mathbf{R}_\uparrow \downarrow x$.*

Proof. By $\uparrow$-consistency, there exist $\mathbf{R}$, $\mathbf{R}_u$, $\mathbf{R}_d$, and $\mathbf{R}_f$ such that $\mathbf{R}_\uparrow = \mathbf{R} \| \mathbf{R}_u \| \mathbf{R}_f$ and $\mathbf{R}_\Delta = \mathbf{R} \| \mathbf{R}_d$. By Definition 5.1 the barbs of $\mathbf{R}_\uparrow$ is the union of barbs of $\mathbf{R}$, $\mathbf{R}_u$, and $\mathbf{R}_f$, and the set of barbs of $\mathbf{R}_\Delta$ is the union of the barbs of $\mathbf{R}$ and $\mathbf{R}_d$. We only need to show that $\mathbf{R}_d$ does not have barbs that $\mathbf{R}_\uparrow$ does not have. This follows trivially from Definition 5.16 since $\mathbf{R}_d$ is the parallel composition of down nodes and hence has no barbs. □

We can now prove a more general property of cursed systems at time 0:

Lemma 5.19. *Let $\lesssim_0$ be the restriction of $\lesssim$ obtained considering only communication actions $\rightharpoonup$ in Figure 3 (i.e., no time-consuming actions $\rightsquigarrow$) on systems $\mathbf{R}$ such that $\mathtt{time}(\mathbf{R}) = 0$. It holds that*

$$(\mathbf{R}, \uparrow) \lesssim_0 (\mathbf{R}, \Delta)$$

Proof. By coinduction, observing that initial systems are $\uparrow$-consistent, $\uparrow$-consistency is preserved by communication actions by Lemma 5.17 and ensures that $\mathbf{R}_\Delta \downarrow x$ implies $\mathbf{R}_\uparrow \downarrow x$ by Lemma 5.18. □

We next show an intuitive property that will be useful to show equivalence of resilience and 0-reliability: $(\mathbf{R}, \Delta)$ and $(\mathbf{R}, \uparrow)$ are weak barbed bisimilar at time 0 if no time actions or failures happen (Lemma 5.21). This is proved by coinduction via Lemma 5.20 ensuring that actor/node transitions preserve equivalence of barbs in the evolution of cursed and uncursed systems.

Lemma 5.20. *If $\mathtt{time}(\mathbf{R}) = 0$, $\mathbf{R}$ is fail-free, and $\rightharpoonup$ is an actor/node action then*
(1) $(\mathbf{R}, \Delta) \rightharpoonup (\mathbf{R}', \Delta) \Leftrightarrow (\mathbf{R}, \uparrow) \rightharpoonup (\mathbf{R}', \uparrow)$
(2) $(\mathbf{R}, \Delta) \rightharpoonup (\mathbf{R}', \Delta) \Rightarrow \mathbf{R}'$ is fail-free.

Proof. By induction on the derivation proceeding by case analysis on the last rule used. The base cases, for rules [Snd], [Sched], [Rcv], and [Checkpoint], are mechanical. The inductive cases for rules [Rec], [Str], and [ParCom], are straightforward by inductive hypothesis. □

Lemma 5.21. *Let $\approx_0^\uparrow$ be the restriction of $\approx$ obtained considering only actor/node actions in Figure 3 (i.e., no failure and no time-consuming actions) and $\mathtt{time}(\mathbf{R}) = 0$ with $\mathbf{R}$ fail-free. It holds that*

$$(\mathbf{R}, \Delta) \approx_0^\uparrow (\mathbf{R}, \uparrow)$$

Proof. This lemma holds by coinduction, observing that $\mathtt{time}(\mathbf{R}) = 0$ and hence, by Lemma 5.20, $(\mathbf{R}, \Delta) \rightharpoonup (\mathbf{R}', \Delta)$ if and only if $(\mathbf{R}, \uparrow) \rightharpoonup (\mathbf{R}', \uparrow)$ with $\mathbf{R}'$ fail-free.   □

We are not able to state the main results: equivalence of resilience and 0-recoverability.

Theorem 5.22 (0-recoverability and resilience). *An initial cursed system $(\mathbf{R}, \Delta)$ is resilient if and only if it is 0-recoverable.*

Proof. The only if case is immediate since resilience implies the existence of a state, the initial one, such that $(\mathbf{R}, \uparrow) \approx (\mathbf{R}, \Delta)$. For the if case, we assume $(\mathbf{R}, \Delta)$ to be 0-*recoverable*: for all path of executions of $(\mathbf{R}, \Delta)$ at time 0 (i.e., 0-paths of $(\mathbf{R}, \Delta)$) there exists a state $(\mathbf{R}', \Delta)$ in that path such that $time(\mathbf{R}') = 0$ and $(\mathbf{R}, \uparrow) \approx (\mathbf{R}', \Delta)$. Fix a path $(\mathbf{R}, \Delta) \rightarrow^* (\mathbf{R}', \Delta)$. Consider a generic intermediate state $(\mathbf{R}'', \Delta)$ such that $(\mathbf{R}, \Delta) \rightarrow^* (\mathbf{R}'', \Delta) \rightarrow^* (\mathbf{R}', \Delta)$. The reductions to $(\mathbf{R}'', \Delta)$ can be by either (i) one of the Actor/node actions, or by (ii) one of the instantaneous failure actions ([MsgLoss] or [NodeDown]).

Observe that, for all $\mathbf{R}$ and Δ, the relation $((\mathbf{R}, \Delta), (\mathbf{R}, \uparrow))$ is a bisimulation if we consider a restriction of the reduction relation that only uses actions that is actor/node actions with no failure and no time-consuming actions (Lemma 5.21, using the fact that $\mathbf{R}$ is initial and hence fail-free). So, if only actions (i) are possible in the reduction of $(\mathbf{R}, \Delta)$ then $(\mathbf{R}'', \Delta) \approx (\mathbf{R}'', \uparrow)$ by Lemma 5.21. If there are only (i) actions at time 0, since any state reachable from $(\mathbf{R}, \Delta)$ at time 0 is bisimilar to the corresponding state reached by $(\mathbf{R}, \uparrow)$ then $(\mathbf{R}, \Delta)$ is resilient. Hence done.

The argument proceeds similarly in case of (ii) actions that preserve the corresponding barbs of the system with Δ and the one with $\uparrow$. Assume therefore that, possibly after a number of barb-preserving reductions by (i) or (ii), the system with Δ and the one with $\uparrow$ reach states $(\mathbf{R}'', \Delta)$ and $(\mathbf{R}''', \uparrow)$, respectively, where $\mathbf{R}''$ and $\mathbf{R}'''$ have different barbs. By Lemma 5.18 it can only be the case $\mathbf{R}'' \not\downarrow x$ and $\mathbf{R}''' \downarrow x$. By hypothesis we have that $(\mathbf{R}'', \Delta) \rightharpoonup^* (\mathbf{R}', \Delta) \approx (\mathbf{R}, \uparrow)$. We also know that $(\mathbf{R}, \uparrow) \rightharpoonup^* (\mathbf{R}''', \uparrow)$ with $\mathbf{R}''' \downarrow x$. Hence $(\mathbf{R}', \Delta)$ can reach a state that has barb x and is bisimilar to $(\mathbf{R}''', \uparrow)$. This shows $(\mathbf{R}, \Delta) \gtrsim (\mathbf{R}, \uparrow)$. The fact that $(\mathbf{R}, \Delta) \lesssim (\mathbf{R}, \uparrow)$ by Lemma 5.19 yields the thesis $(\mathbf{R}, \Delta) \approx (\mathbf{R}, \uparrow)$.   □

5.3. **Recoverability for checkpointing systems.** The definition of recoverability in the previous section formalises a system restarting from the initial state, and does not capture checkpointing systems that recover to intermediate states. In this section we add definition of bisimulation up to a particular time, and also a notion of n-recoverability for checkpointing systems. This is illustrated with an example of a system that is not n-recoverable but that is n-checkpoint recoverable.

We introduce a notion of weak barbed simulation up to n where n is a relative time, up to which we want to compare behaviour (ignoring what happens afterwards).

Definition 5.23 (Weak barbed simulation up to n). Recall $\rightarrow \in \{\rightharpoonup, \leadsto\}$. A weak barbed simulation up to n is a set of binary relations $\mathcal{S}^r$ for $r \leq n$ between cursed systems such that:

(1) $r = 0$: $(\mathbf{R}_1, \Delta_1) \, \mathcal{S}^0 (\mathbf{R}_2, \Delta_2)$ for all $\mathbf{R}_1$, Δ_1, $\mathbf{R}_2$ and Δ_2;
(2) $r > 0$ and $(\mathbf{R}_1, \Delta_1) \, \mathcal{S}^r (\mathbf{R}_2, \Delta_2)$ implies:

(a) If $(\mathbf{R}_1, \Delta_1) \to (\mathbf{R}_1', \Delta_1)$ and $s = time(\mathbf{R}_1') - time(\mathbf{R}_1) \leq r$, then there exists $\mathbf{R}_2'$ such that $(\mathbf{R}_2, \Delta_2) \to^* (\mathbf{R}_2', \Delta_2)$ and $(\mathbf{R}_1', \Delta_1) \, \mathcal{S}^{(r-s)} (\mathbf{R}_2', \Delta_2)$.

(b) If $\mathbf{R}_1 \downarrow x$ for some x, then there exists $\mathbf{R}_2'$ such that $(\mathbf{R}_2, \Delta_2) \to^* (\mathbf{R}_2', \Delta_2)$ and $\mathbf{R}_2' \downarrow x$.

We say $(\mathbf{R}_1, \Delta_1)$ is weak barbed similar to $(\mathbf{R}_2, \Delta_2)$ up to n, written $(\mathbf{R}_1, \Delta_1) \lesssim^n (\mathbf{R}_2, \Delta_2)$, if there exists some weak barbed simulation up to n, $\mathcal{S}^n$, such that $(\mathbf{R}_1, \Delta_1) \, \mathcal{S}^n (\mathbf{R}_2, \Delta_2)$. By point (1), weak barbed simulation up to 0 holds for all pairs of systems, whereas weak barbed simulation is morally equivalent to weak barbed simulation up to ∞.

Definition 5.24 (Weak barbed bisimulation up to n). We say that $\mathcal{S}^n$ is a weak barbed bisimulation up to n if $\mathcal{S}^n$ and $\mathcal{S}^{n-1}$ are weak barbed simulations up to n. We say that $(\mathbf{R}_1, \Delta_1)$ and $(\mathbf{R}_2, \Delta_2)$ are weak barbed bisimilar up to n, written $(\mathbf{R}_1, \Delta_1) \approx^n (\mathbf{R}_2, \Delta_2)$, if there exists some weak barbed bisimulation up to n, $\mathcal{S}^n$, such that $(\mathbf{R}_1, \Delta_1) \, \mathcal{S}^n (\mathbf{R}_2, \Delta_2)$.

It is a straightforward consequence of these definitions that if two systems are (bi-)similar up to n then they are (bi-)similar up to r for any $r < n$, and two systems are (bi-)similar if and only if they are (bi-)similar up to n for all n.

We now define a variant of n-recoverability that, based on weak barbed bisimulation up to n, aims to characterise recoverability for systems that use checkpoints to recover from failures.

Definition 5.25 (n-checkpoint-recoverable). For all $\mathbf{R}''$ reachable from $(\mathbf{R}, \uparrow)$ such that $\text{time}(\mathbf{R}'') < n$ and $(\mathbf{R}, \uparrow) \approx^{\text{time}(\mathbf{R}'')} (\mathbf{R}, \Delta)$ there exists $\mathbf{R}'$ such that $\text{time}(\mathbf{R}') < n$ and $(\mathbf{R}, \Delta) \to^* (\mathbf{R}', \Delta)$ and $(\mathbf{R}', \Delta) \approx (\mathbf{R}'', \uparrow)$.

Informally, a cursed system $(\mathbf{R}, \Delta)$ that behaves correctly up to a state at time t, can always reach a later state $(\mathbf{R}', \Delta)$ which is bisimilar to the state $(\mathbf{R}'', \uparrow)$. Suppose that $(\mathbf{R}, \uparrow) \approx^t (\mathbf{R}, \Delta)$, where $t \leq n$. Then for any $\mathbf{R}''$ reachable from $(\mathbf{R}, \uparrow)$ such that $\text{time}(\mathbf{R}'') = t$ there exists $\mathbf{R}'$ that by time n displays the remaining behaviour of the correct (uncursed) system, that is $(\mathbf{R}'', \uparrow)$. In contrast to Definition 5.13 this definition does not require that $(\mathbf{R}', \Delta)$ exhibits the complete behaviour of $(\mathbf{R}, \uparrow)$ but only the behaviour after a certain point, for example from a checkpoint onwards. In contrast to Definition 5.13, Definition 5.25 does not require the recovered system $(\mathbf{R}', \Delta)$ to exhibit the complete behaviour of the uncursed system since its initial state (i.e., have the same behaviour of $(\mathbf{R}, \uparrow)$) but only the behaviour of $(\mathbf{R}, \uparrow)$ after a certain point (i.e., $(\mathbf{R}'', \uparrow)$) for example from a checkpoint onwards.

Example 5.26 (n-checkpoint-recoverability). Consider a variant $\mathbf{R}_{TT}$ of $\mathbf{R}_T$ from Example 5.7, latency of 1 time unit, and Δ that curses node $\mathbf{p}$ to go down at time 2:

$$\mathbf{R}_{TT} = \;\; \mathbf{p}[\,?order.\mathbf{save}.\mathbf{sleep}.!\mathbf{c}\,item.\,?ok.0 \text{ after } 3\;0\,](\emptyset)(0) \;\|$$
$$\mathbf{c}[\,!\mathbf{p}\,order.\,?item.\,!\mathbf{p}\,ok.0 \text{ after } 3\;(!\mathbf{p}\,failed.\,?item.\,!\mathbf{p}\,ok.0 \text{ after } 5\;0)\,](\emptyset)(0)$$

System $\mathbf{R}_{TT}$ used checkpointing to restart from an intermediate state in the event of failure. In the case of the node $\mathbf{p}$ to going down at time 2, the system reduces in a number of steps to:

$$\mathbf{R}'_{TT} = \;\; \mathbf{p}[\,\mathbf{sleep}.!\mathbf{c}\,item.\,?ok.0 \text{ after } 3\;0\,](\emptyset)(3) \;\|$$
$$\mathbf{c}[\,?item.\,!\mathbf{p}\,ok.0 \text{ after } 5\;0)\,](\emptyset)(3) \;\|\; 1.(\mathbf{c},\mathbf{p},failed)(3)$$

It is this state that makes system $\mathbf{R}_{TT}$ not n-recoverable with respect to Δ both because node $\mathbf{c}$ sends message $failed$ in its timeout process and the communication of the $order$

message does not get repeated when the rest of interaction is repeated. While the `failed` message is not read by p, it adds an additional barb to the cursed system $(\mathbf{R}_{TT}, \Delta)$ that is not matched by the uncursed system $(\mathbf{R}_{TT}, \uparrow)$.

The system is however n-checkpoint-recoverable with $n = 4$, $\mathbf{R}'_{TT}$ further reduces to

$$\texttt{p[!c item.\,?ok.0\ after}\ 2\ 0](\texttt{failed})(4) \parallel \texttt{c[?item.\,!p ok.0\ after}\ 4\ 0)](\emptyset)(4)$$

from which state the cursed system exhibits the behaviour of the uncursed system from time 2 (or from the checkpoint) onwards.

5.4. **Fault-tolerance: a more general perspective.** In [Gär99], the author gives a theoretical definition of the problem of fault tolerance along two dimensions: safety (the system does not reach bad states, although it can possibly stop due to faults) and liveness (the system eventually reaches good states, hence in case of bad behaviour it eventually recovers). In this context, the guarantee of both safety and liveness is called *masking fault-tolerance*, of only safety is called *fail-safe fault-tolerance*, and of only liveness is called *non-masking fault-tolerance*.

In our framework, we can characterise these three kinds of fault-tolerance by using our simulation relation:

- $(\mathbf{R}, \Delta) \approx (\mathbf{R}, \uparrow)$ - *masking fault-tolerance*: $\mathbf{R}$ cursed by Δ has all and only the behaviour of healthy system $\mathbf{R}$.
- $(\mathbf{R}, \Delta) \lesssim (\mathbf{R}, \uparrow)$ - *fail-safe*: $\mathbf{R}$ cursed by Δ has only the behaviour of healthy system $\mathbf{R}$.
- $(\mathbf{R}, \Delta) \gtrsim (\mathbf{R}, \uparrow)$ - *non-masking fault tolerant*: $\mathbf{R}$ cursed by Δ has only the behaviour of healthy system $\mathbf{R}$.

Resilience, given in Definition 5.5, corresponds to the safety and liveness combination of masking fault-tolerance. We have shown in Example 5.7 that masking-failure can be attained by using space redundancy (e.g., replication of nodes as in the multiple producers scenario) and time redundancy (e.g., retry-strategies). In fact, the author in [Gär99] substantiates that redundancy is a necessary condition for fault tolerance. Fail-safe fault tolerance, while easier to attain, is not the most desirable property in many real-world scenarios: a systems that just stops to prevent 'bad' actions, may not be a suitable model when you want eventually consistency despite perturbations to the ideal course of actions. Intuitively, both fail-safe fault tolerance and non-masking fault tolerance for cursed system can be expressed by using the notion of weak barbed simulation given in Definition 5.3, as shown above. The formulation of *non-masking fault tolerant* as $(\mathbf{R}, \Delta) \gtrsim (\mathbf{R}, \uparrow)$ is very general. In principle, this definition consider fault tolerant any system that performs an infinite sequence of actions among which, sometimes, a correct action happens to make the system progress. Practically we would want to see that the behaviour added is not random, but follows a sensible pattern of restart, reset or another benign behaviour. In this paper we have put most emphasis on non-masking fault tolerance, but focussing on more stringent definitions of non-masking fault-tolerance: some unforeseen sequence of actions may be visible at some point, but after some recovery actions, at a time that is not later than n, the system will revert to the required behaviour by restarting from the beginning (n-recoverability for reset systems) or from the point of failure (n-recoverability for checkpointing systems). These definitions are intentionally non-general, with the aim of capturing known recovery patterns. We leave as a future work the extension of n-recoverability to cater for periodic failures.

A similar approach, of characterising masking/fail-safe/non-masking fault tolerance using simulation was followed by [DCMA17] but with a clear distinction of good versus faulty states (using coloured Kripke structures). More on the relationship with [DCMA17] is discussed in Section 8.

6. AUGMENTATION OF CURSED SYSTEMS

Augmentation of a cursed system is the result of adding or modifying some behaviour in the initial system to improve the system's ability of handling failures. The following definition applies to reset systems as it is based on n-recoverability. A corresponding notion of augmentation could be given for checkpointing systems by using, in Definition 6.1, n-checkpoint-recoverability instead of n-recoverability. In the remaining of this section we focus on reset systems.

Definition 6.1 (Augmentation). $\mathbf{R_I}$ is an augmentation of $\mathbf{R}$ if $time(\mathbf{R_I}) = time(\mathbf{R})$ and:

i) transparency: $(\mathbf{R}, \uparrow) \approx (\mathbf{R_I}, \uparrow)$

ii) improvement: there exist Δ and n such that $(\mathbf{R_I}, \Delta)$ is n-recoverable and $(\mathbf{R}, \Delta)$ is not n-recoverable.

Moreover, we say that an augmentation is *preserving* if, for all n and Δ, $(\mathbf{R}, \Delta)$ is n-recoverable implies $(\mathbf{R_I}, \Delta)$ is n-recoverable.

Example 6.2 (Augmentation). Consider the small producer-consumer system $\mathbf{R}$ below, composed of a producer node $\mathbf{n_p}$, a queue node $\mathbf{n_q}$, and a consumer node $\mathbf{n_c}$. The producer recursively sends items to the queue and sleeps for a time unit. The queue expects to receive an item within three time units that then gets sent to the consumer. In case of a timeout the queue loops back to the beginning and awaits an item from the producer. The consumer recursively receives items from the queue. We fix the latency of the system to $L = 1$.

$$\mathbf{R} = \mathbf{n_q}[\mu t.\, ?\, item.\mathbf{sleep}.\, !\, \mathbf{n_c}\, item.t \text{ after } 3\, t](\emptyset)(0) \,\|$$
$$\mathbf{n_p}[\, \mu t.\, !\, \mathbf{n_q}\, item.\mathbf{sleep}.t\,](\emptyset)(0) \,\|\, \mathbf{n_c}[\, \mu t.\, ?\, item.\mathbf{sleep}.t \text{ after } 4\, t\,](\emptyset)(0)$$

$$\mathbf{R_I} = \mathbf{R} \,\|\, \mathbf{n_{p'}}[\, \mu t.\, !\, \mathbf{n_q}\, item.\mathbf{sleep}.t\,](\emptyset)(0)$$

The augmented producer-consumer $\mathbf{R_I}$ adds behaviour to the system by having a second producer node $\mathbf{n_{p'}}$. $\mathbf{R_I}$ improves the resilience to a producer node or its link failing or being slow. For example the curse function $\Delta(\mathbf{n_p})$ injecting node delay for the producer node between time 1 and 3 and $\uparrow$ otherwise impacts the first system $\mathbf{R}$ but not its augmented counterpart $\mathbf{R_I}$. $\mathbf{R}$ is 4-recoverable while $\mathbf{R_I}$ is 0-recoverable. Moreover, $\mathbf{R_I}$ preserving augmentation of system $\mathbf{R}$.

6.1. Augmentation with scoped barbs.

Augmentations often need to introduce additional behaviour into actors. One may want to disregard part of 'behind the scenes' augmentation when comparing the behaviour of cursed systems using the relation in Definition 5.4. For simplicity, instead of adding scope restriction to the calculus, we extend barbs with scopes to hide behaviour of some nodes or links. With mailboxes, all interactions to a node are directed to the one mailbox. Defining scope restriction only on node identifiers would be less expressive than scope restriction based on channels, e.g., it would not be possible to hide specific communications to a node, while in channel-based calculi one can

use ad-hoc hidden channels. To retain expressiveness, we define scope restriction that takes into account *patterns* in the communication between nodes.

Definition 6.3 (Scoped barb). Let N be a finite set of elements of the form $!\,\mathtt{n}\,p$ or $?\,\mathtt{n}\,p$ where $\mathtt{n} \in \mathcal{N}$ and p is a pattern. $\mathbf{R} \downarrow_N x$ if: (1) $\mathbf{R} \downarrow x$, (2) $x \notin N$, and (3) if $x = !\,\mathtt{n}\,m$ then for all $!\,\mathtt{n}\,p \in N$, $(p, m) \not\vdash_{\text{match}}$. If $\mathbf{R} \downarrow_N x$ we say that $\mathbf{R}$ has a N-scoped *barb* on x.

We extend Def. 5.4 using $\downarrow_N$ instead of $\downarrow$, obtaining scoped weak-barbed bisimulation $\approx_N$, and Def. 6.1 to use $\approx_N$. This setting allow us to analyse producer consumer scenarios, or more complex ones, like the Circuit Breaker pattern [Nyg18] widely used in distributed systems.

Example 6.4 (Circuit breaker). Consider system $(\mathbf{R}, \Delta)$ with a client $\mathtt{n_c}$ and a service $\mathtt{n_s}$, and its augmentation $\mathbf{R_I}$ with a circuit breaker running on node $\mathtt{n_s}$:

$$\mathbf{R} = \quad \mathtt{n_c}[\,\mu\mathtt{t}.!\,\mathtt{n_s}\,request.?\,reply.\mathbf{sleep}.\mathtt{t}\ \mathtt{after}\ 4\,\mathbf{0}\,](\emptyset)(0) \,||$$
$$\mathtt{n_s}[\,\mu\mathtt{t}.?\,request.\mathbf{sleep}.!\,\mathtt{n_c}\,reply.\mathtt{t}\ \mathtt{after}\ 4\,\mathtt{t}\,](\emptyset)(0)$$

$$\mathbf{R_I} = \quad \mathtt{n_c}[\,\mu\mathtt{t}.!\,\mathtt{n_s}\,request.?\{reply.\mathbf{sleep}.\mathtt{t}, ko.P_f\}\ \mathtt{after}\ 8\,\mathbf{0}\,](\emptyset)(0) \,||$$
$$\mathtt{n_s}[\,\mu\mathtt{t}.?\,X_1.!\mathtt{n_1}\,X_1.?\,X_2.!\mathtt{n_c}\,X_2.\mathtt{t}\ \mathtt{after}\ 4\,P_f'\ \mathtt{after}\ 4\,\mathtt{t}\,](\emptyset)(0) \,||$$
$$\mathtt{n_1}[\,\mu\mathtt{t}.?\{request.\mathbf{sleep}.!\mathtt{n_s}\,reply.\mathtt{t}, ruok.\mathbf{sleep}.!\mathtt{n_s}\,imok.\mathtt{t}\}\ \mathtt{after}\ 6\,\mathtt{t}\,](\emptyset)(0)$$

$$P_f = \quad \mu\mathtt{t}'.?\,retry.\mathbf{sleep}.\mathtt{t}\ \mathtt{after}\ 5\,\mathtt{t}'$$
$$P_f' = \quad !\,\mathtt{n_c}\,ko.\mathbf{sleep}.\mu\mathtt{t}'.!\,\mathtt{n_s}\,ruok.?\,imok.\mathbf{sleep}.!\,\mathtt{n_c}\,retry.\mathtt{t}\ \mathtt{after}\ 3\,\mathtt{t}'$$

with a $\Delta(\mathtt{n_c}, \mathtt{n_s})$ injecting link slow $\circlearrowleft$ at times 1, 2, and 3 and healthy otherwise, and latency to $L = 1$. The impact of failure on the $\mathbf{R}$ makes it unrecoverable, as the link delay cascades to node $\mathtt{n_c}$. We augment $\mathbf{R}$ with a circuit breaker process which runs on the previous server node $\mathtt{n_s}$ that monitors for failure, prevents faults in one part of the system and controls the retries to the service node now $\mathtt{n_1}$. The node $\mathtt{n_s}$ forwards messages between nodes $\mathtt{n_c}$ and $\mathtt{n_1}$, and in case of a timeout checks the health of $\mathtt{n_s}$ and tells node $\mathtt{n_c}$ when it can safely retry the request. When comparing $\mathbf{R}$ and $\mathbf{R_I}$ for resilience, recoverability or transparency we wish to abstract from the additional behaviour introduced by the circuit breaker pattern for which we use Def. 6.3 with: $N = \{!\,\mathtt{n_s}\,ruok, ?\,\mathtt{n_s}\,imok, ?\,\mathtt{n_s}\,reply, ?\,\mathtt{n_1}\,request, !\,\mathtt{n_s}\,reply, ?\,\mathtt{n_1}\,ruok, !\,\mathtt{n_s}\,imok,$
$!\,n_c\,ko, !\,n_c\,retry, ?\,n_c\,ko, ?\,n_c\,retry\}$. This effectively hides the entire behaviour of $\mathtt{n_1}$ and node $\mathtt{n_s}$'s health checking behaviour. Using the extended definition we find that for the same curse function system $\mathbf{R_I}$ is 0-recoverable. Similarly, for the curse function delays link $(\mathtt{n_s}, \mathtt{n_1})$ at times 1, 2, and 3, $\mathbf{R_I}$ is 0-recoverable.

7. Prospective applications

This work is a first step towards an analysis of mailbox systems with failures and has the purpose of clarifying the problem space. An informal validation of the relevance of the work was attained through interaction with our industry partners, in particular Erlang Solutions Ltd. and Actyx AG, as well as in applying it to a collection of real-world case studies and patterns, such as the circuit breaker in Section 6.4.

To support analysis and development of real-world systems, we aim to build on the current work. In this section we discuss two potential applications; their development goes beyond the scope of the formal setting given in the current work.

7.1. **Analysis of cursed systems.** Encoding the models of failures and systems into verification tools like UPPAAL is fairly straightforward. We provide a prototype encoding here. A straightforward encoding only supports analysis of a system against a specific Δ 'traces', and so, by repeating this analysis, to a limited set of curse cases.

As a more powerful development, we are working on generalizing the notion of Δ to a symbolic entity, that can finitely characterise infinite patterns, together with a tractable algorithm to determine simulation that is parametric with respect to this symbolic Δ. This feature would allow it to be determined whether a system model is resilient with respect to a given set of curses, or synthesise the curses that a system can or cannot deal with.

Code generation or synthesis would, in turn support top-down or bottom-up development, (respectively). Existing approaches to code generation provide seamless links between process-calculi-based models and Erlang code. For example, the tool described in [BOV23], which presents a proof of concept of a theoretical advance, can generate Erlang `gen_statem` code from a process-calculus specification and extract specifications from Erlang `gen_statem` code. The circular transformation described above is possible thanks to the code structure induced by Erlang `gen_statem` itself, which yields modular code that is structured as a finite state machine and hence has a straightforward correspondence with its model. A similar approach could be taken to our modelling of failure scenarios by supporting a richer process calculus that includes time and timeouts.

7.2. **Test support.** A second direction is to use property-based testing (PBT), as implemented by QuickCheck [CH00], initially for Haskell and Erlang, and subsequently for a variety of other languages. Property-based testing replaces unit tests by tests of logical properties of the system under test (SUT), expressed in a universal fragment of first-order logic. A universal property is evaluated at a randomly generated set of values, and any counter-example is systematically shrunk to a simplest such example, according to some size metric. Successful application of property-based testing therefore depends on three things: being able to express relevant properties of a system in a logical form; being able to generate values from relevant domains in a way that optimises coverage of the domain; and being able to "shrink" values in an effective and efficient way. PBT can be seen as a complement to more heavyweight verification approaches: for example, it is worthwhile subjecting a candidate theorem to PBT before embarking on developing a formal proof.

Stateful systems in Erlang [CPS$^+$09] and other languages can subject to PBT using state machine models. The state machine provides an abstract model of the system, and is used to guide testing of the SUT: random sequences of transitions of the state machine exercise the SUT, and shrinking simplifies and shortens counter-example traces.

In the context of the work presented here, QuickCheck can be used to test systems in which failure is modelled explicitly in a state machine model, but could also be extended to include modelling of the symbolic Δ function discussed above – e.g. using logical constraints – and to generate and shrink instances of Δ with particular properties.

8. Conclusion and related work

We introduced a model for actor-based systems with grey failures and investigated the definition of behavioural equivalence for it. We used weak barbed bisimulation to compare systems on the basis of their ability to recover from faults, and defined properties of resilience, recoverability and augmentation. We reduced the problem of checking reliability properties of

systems to a problem of checking bisimulation. We introduced scope restriction for mailboxes based on patterns, which allows us to model relatively complex real-world scenarios like the Circuit Breaker.

As further work we plan to extend the recovery function Σ to model check-pointing of intermediate node states. Note that Σ can already be set as an arbitrary process, but a more meaningful extension would account for the way in which checkpoints are saved. Moreover, we plan to add a notion of intermittent correctness, to model recovery with partial checkpoints rather than re-starting from the initial state, or intermittent expected/unexpected behaviour. Another area of future work is to use the characteristic formulae approach [GS86, Ste89], a method to compute simulation-like relations in process algebras, to generate formulae for the properties introduced and reduce them to a model checking problem that can be offloaded to a model checker.

A related formalism to our model is Timed Rebeca [ACI$^+$11], which is actor-based and features similar constructs for deadlines and delays. Timed Rebeca actors can also use a '*now*' function to get their local times. Extending our calculus with '*now*' and allowing messages to have time as data sort, would allow us to model scenarios e.g., where a node calculates the return-trip time to another node and changes its behaviour accordingly. While Timed Rebeca can encode network delays (adding delays to receive actions – using a construct called '*after*'), it does not model links explicitly. Explicit links and separation between curses and systems make it easier in our calculus to compare systems with respect to recoverability. Rebeca was encoded in McErlang [ACI$^+$11] and Real-Time Maude [SKÖ$^+$15] for verification. We have ongoing work on encoding our model in UPPAAL. Our main challenge in this respect is to formalise a meaningful and manageable set of curses to verify the model against.

In [FH07], Francalanza and Hennessy introduced a behavioural theory for DπF, a distributed π-calculus with with nodes and links failures. For a subset of DπF, they also developed a notion of fault-tolerance up to n-faults [FH08], which is preserved by contexts, and which is related to our notion of resilience. The behavioural theory in [FH07] is based on reduction barbed congruence. The idea is to use a contextual relation to abstract from the behaviour of hidden nodes/links, while still observing their effects on the network, e.g., as to accessibility and reachability of other nodes. The scoped barbs in Section 6.1 have the similar purpose of hiding augmentations while observing their effects on recoverability. However, because of asynchronous communication over mailboxes (while DπF is based on synchronous message passing), our notion of hiding is less structural (i.e., based on nodes and links) and more application-dependent (i.e., based on patterns). At present, we have left pattern hiding out of the semantics, but further investigation towards a contextual relation that works for hidden patterns is promising future work. DπF studies partial failures but does not consider transient failures and time. On the other hand, DπF features mobility which we do not support. In fact, we rely on the assumption of fixed networks: since our observation is based on patterns (and ignores senders) we opted for relying on a stable structure to simplify our reasoning on what augmentation vs recoverability means, leaving mobility issues for future investigation.

Most ingredients of the given model (e.g., timeouts [LZ05, BY07, LP11], mailboxes [MV11], localities [RH97][BH03][Cas01]) have been studied in literature, often in isolation. We investigated the inter-play of these ingredients, focussing on reliability properties. One of the first papers dealing with asynchronous communication in process algebra is by de Boer et al. [dBKP92], where different observation criteria are studied (bisimulation, traces and abstract traces) following the axiomatic approach typical of the process algebra ACP [BK84].

An alternative approach has been followed by Amadio et al. [ACS98] who defined asynchronous bisimulation for the π-calculus [MPW92]. They started from operational semantics (expressed as a standard labelled transition system), and then considered the largest bisimulation defined on internal steps that equates processes only when they have the same observables, and which is closed under contexts. The equivalence obtained in this way is called barbed congruence [MS92]. Notably, when asynchronous communication is considered, barbed congruence is defined assuming as observables the messages that are ready to be delivered to a potential external observer. Merro and Sangiorgi [MS98] have subsequently studied barbed congruence in the context of the Asynchronous Localised π-calculus (ALπ), a fragment of the asynchronous π-calculus in which only output capabilities can be transmitted, i.e., when a process receives the name of a channel, it can only send messages along it, but cannot receive on it. Another line of research deals with applying the testing approach to asynchronous communication; this has been investigated by Castellani and Hennessy [CH98] and by Boreale et al. [BNP99, BNP02]. These papers consider an asynchronous variant of CCS [Mil89]. Testing discriminates less than our equivalence, concerning choice, and observes divergent behaviours which we abstract from. Lanese et al. [LSZ19] look at bisimulation for Erlang, focussing on the management of process ids. Besides the aforementioned work by Francalanza and Hennessy [FH07, FH08], several works look at distributed process algebras with unreliable communication due to faults in the underlying network. Riely and Hennessy [RH97] study behavioural equivalence over process calculi with locations. Amadio [Ama97] extends the π-calculus with located actions, in the context of a higher-order distributed programming language. Fournet et al. [FGL$^+$96] look at locations, mobility and the possibility of location failure in the distributed join calculus. The failure of a location can be detected and recovered from. Berger and Honda [BH03] augment the asynchronous π-calculus with a timer, locations, message-loss, location failure and the ability to save process state. They define a notion of weak bisimulation over networks. Their model however does not include timeout, link delays, or a way of injecting faults. Cano et al. [CCDGP19] develop a calculus and type system for multiparty reactive systems that models time dependent interactions. Their setting is synchronous and their focus is on proving properties as types safety or input timeliness, while ours is comparing asynchronous systems with faults.

In Section 5.4 we discussed two related works: one characterising fault-tolerance using safety and liveness [Gär99], and one [DCMA17] instantiating such characterisation using simulation. One of the main differences of our work with [DCMA17] is the communication model: the work in [DCMA17] uses coloured Kripke structures, whereas we use an asynchronous process calculus with explicit actor-based features. Another difference is the model of failure and the characterization of good versus bad states. Usually, fault-tolerance is studied against a well define model of failure and set of failure scenarios. Systems that can recover from arbitrary failures, called *self-stabilizing* [Gär99] are difficult to build and verify. Hence models for foult-tolerant systems normally have a model of failures within. In our case, this would be a set of Δ functions. In [DCMA17] there is an explicit and statical labelling (colouring) of each state as good or bad. In contrast, we observe deviations from the behaviour of the same system but without failures, so that a bad state is actually one that breaks the bisimulation relation with the corresponding uncursed system. Finally, in [Gär99][DCMA17], there is no clear distinction between the notion of fault – a defect of the system – and its concrete manifestation as a symptom. Our model of failure, that separates

system from curses yields a more agnostic view of what a bad state is that supports modular reasoning on the relationship between causes and symptoms.

References

[ACI⁺11]   Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed rebeca. *EPTCS*, 58:1–19, 2011. `doi:10.4204/eptcs.58.1`.

[ACS98]   Roberto M. Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theor. Comput. Sci.*, 195(2):291–324, 1998. `doi:10.1016/S0304-3975(97) 00223-5`.

[Ama97]   Roberto M. Amadio. An asynchronous model of locality, failure, and process mobility. In *Proc. COORDINATION*, volume 1282 of *LNCS*, pages 374–391. Springer, 1997. `doi:10.1007/ 3-540-63383-9_92`.

[APN17]   Manuel Adameit, Kirstin Peters, and Uwe Nestmann. Session types for link failures. In *Proc. FORTE*, volume 10321 of *LNCS*, pages 1–16. Springer International Publishing, 2017. `doi: 10.1007/978-3-319-60225-7_1`.

[Arm13]   Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, Dallas, TX, 2 edition, 2013.

[BBO12]   Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. *Proc. ACM Program. Lang. (POPL)*, 47:191–202, 2012. `doi:10.1145/2103656.2103680`.

[BGF⁺21]   Benedikt Bollig, Cinzia Di Giusto, Alain Finkel, Laetitia Laversa, Étienne Lozes, and Amrita Suresh. A unifying framework for deciding synchronizability. In *Proc. CONCUR*, volume 203 of *LIPIcs*, pages 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.CONCUR.2021.14`.

[BH03]   Martin Berger and Kohei Honda. The two-phase commitment protocol in an extended π-calculus. *ENTCS*, 39(1):21–46, 2003. `doi:10.1016/S1571-0661(05)82502-2`.

[BK84]   Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Inf. Control.*, 60(1-3):109–137, 1984. `doi:10.1016/S0019-9958(84)80025-X`.

[BLTV22]   Laura Bocchi, Julien Lange, Simon Thompson, and A. Laura Voinea. A model of actors and grey failures. In *Proc. COORDINATION*, volume 13271 of *LNCS*, pages 140–158. Springer, 2022. `doi:10.1007/978-3-031-08143-9_9`.

[BNP99]   Michele Boreale, Rocco De Nicola, and Rosario Pugliese. A theory of "may" testing for asynchronous languages. In *Proc. FoSSaCS*, volume 1578 of *LNCS*, pages 165–179. Springer, 1999. `doi:10.1007/3-540-49019-1_12`.

[BNP02]   Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Trace and testing equivalence on asynchronous processes. *Inf. Comput.*, 172(2):139–164, 2002. `doi:10.1006/inco.2001.3080`.

[BOV23]   Laura Bocchi, Dominic Orchard, and A. Laura Voinea. A Theory of Composing Protocols. *Art Sci. Eng. Program.*, 7(2), 2023. `doi:10.22152/programming-journal.org/2023/7/6`.

[BY07]   Martin Berger and Nobuko Yoshida. Timed, distributed, probabilistic, typed processes. In *Proc. APLAS*, volume 4807 of *LNCS*, pages 158–174. Springer, 2007. `doi:10.1007/ 978-3-540-76637-7_11`.

[BZ83]   Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983. `doi:10.1145/322374.322380`.

[Cas01]   Ilaria Castellani. Process algebras with localities. In *Handbook of Process Algebra*, pages 945–1045. North-Holland / Elsevier, 2001. `doi:10.1016/b978-044482830-9/50033-3`.

[CCDGP19]   Mauricio Cano, Ilaria Castellani, Cinzia Di Giusto, and Jorge A. Pérez. Multiparty Reactive Sessions. Research Report 9270, INRIA, April 2019. URL: `https://hal.archives-ouvertes. fr/hal-02106742`.

[CDYP16]   Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. Global progress for dynamically interleaved multiparty sessions. *MSCS*, 26(2):238–302, 2016. `doi: 10.1017/S0960129514000188`.

[CGY16]   Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *MSCS*, 26(2):156–205, 2016. `doi:10.1017/S0960129514000164`.

[CH98]   Ilaria Castellani and Matthew Hennessy. Testing theories for asynchronous languages. In *Proc. FSTTCS*, volume 1530 of *LNCS*, pages 90–101. Springer, 1998. `doi:10.1007/978-3-540-49382-2_9`.

[CH00]   Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. ICFP*, pages 268–279. ACM, 2000. `doi:10.1145/351240.351266`.

[CPS+09]   Koen Claessen, Michal Palka, Nicholas Smallbone, John Hughes, Hans Svensson, Thomas Arts, and Ulf Wiger. Finding Race Conditions in Erlang with QuickCheck and PULSE. *SIGPLAN Not.*, 44(9):149–160, aug 2009. `doi:10.1145/1631687.1596574`.

[dBKP92]   Frank S. de Boer, Jan Willem Klop, and Catuscia Palamidessi. Asynchronous communication in process algebra. In *Proc. LICS*, pages 137–147. IEEE Computer Society, 1992. `doi:10.1109/LICS.1992.185528`.

[DCMA17]   Ramiro Demasi, Pablo F. Castro, Thomas S. E. Maibaum, and Nazareno Aguirre. Simulation relations for fault-tolerance. *Form. Asp. Comput.*, 29(6):1013–1050, nov 2017. `doi:10.1007/s00165-017-0426-2`.

[FGL+96]   Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proc. CONCUR*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996. `doi:10.1007/3-540-61604-7_67`.

[FH07]   Adrian Francalanza and Matthew Hennessy. A theory for observational fault tolerance. *JLAMP*, 73(1-2):22–50, 2007. `doi:10.1007/11690634_2`.

[FH08]   Adrian Francalanza and Matthew Hennessy. A theory of system behaviour in the presence of node and link failure. *Inf. Comput.*, 206(6):711–759, 2008. `doi:10.1016/j.ic.2007.12.002`.

[FLMD19]   Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. Exceptional asynchronous session types: session types without tiers. *Proc. ACM Program. Lang. (POPL)*, 3:1–29, 2019. `doi:10.1145/3290341`.

[Gär99]   Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999. `doi:10.1145/311531.311532`.

[GS86]   Susanne Graf and Joseph Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Inf. Control.*, 68(1-3):125–145, 1986. `doi:10.1016/S0019-9958(86)80031-6`.

[GSS+18]   Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Trans. Storage*, 14(3):23:1–23:26, 2018. `doi:10.1145/3242086`.

[HGZ+17]   Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proc. HotOS*, pages 150–155, New York, NY, USA, 2017. Association for Computing Machinery. `doi:10.1145/3102980.3103005`.

[HNY+13]   Raymond Hu, Rumyana Neykova, Nobuko Yoshida, Romain Demangeon, and Kohei Honda. Practical interruptible conversations. In *Proc. RV*, pages 130–148. Springer, 2013. `doi:{10.1007/978-3-642-40787-1_8}`.

[HR95]   Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Inf. Comput.*, 117(2):221–239, 1995. `doi:10.1006/inco.1995.1041`.

[HYC16]   Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016. `doi:10.1145/2827695`.

[LHS20]   Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *Proc. NDSI*, pages 559–574. USENIX Association, 2020. URL: `https://www.usenix.org/conference/nsdi20/presentation/lou`.

[LNPV18]   Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for Erlang. *JLAMP*, 100:71–97, 2018. `doi:10.1016/j.jlamp.2018.06.004`.

[LP11]   Hugo A. López and Jorge A. Pérez. Time and exceptional behavior in multiparty structured interactions. In *Proc. WS-FM*, volume 7176 of *LNCS*, pages 48–63. Springer, 2011. `doi:10.1007/978-3-642-29834-9_5`.

[LSZ19]   Ivan Lanese, Davide Sangiorgi, and Gianluigi Zavattaro. Playing with bisimulation in erlang. In *Models, Languages, and Tools for Concurrent and Distributed Programming: Essays Dedicated*

to Rocco De Nicola on the Occasion of His 65th Birthday, volume 11665 of *LNCS*, pages 71–91. Springer International Publishing, 2019. `doi:10.1007/978-3-030-21485-2_6`.

[LZ05]   Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *Proc. FoSSaCS*, volume 3441 of *LNCS*, pages 282–298. Springer, 2005. `doi:10.1007/978-3-540-31982-5_18`.

[Mil89]   Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989. `doi:10.5555/534666`.

[MPW92]   Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. `doi:10.1016/0890-5401(92)90008-4`.

[MS92]   Robin Milner and Davide Sangiorgi. Barbed bisimulation. In *Proc. ICALP*, volume 623 of *LNCS*, pages 685–695. Springer, 1992. `doi:10.1007/3-540-55719-9_114`.

[MS98]   Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proc. ICALP*, volume 1443 of *LNCS*, pages 856–867. Springer, 1998. `doi:10.1007/BFb0055108`.

[Mur19]   Maurizio Murgia. Input urgent semantics for asynchronous timed session types. *JLAMP*, 107:38–53, 2019. `doi:10.1016/j.jlamp.2019.04.001`.

[MV11]   Dimitris Mostrous and Vasco T. Vasconcelos. Session typing for a Featherweight Erlang. In *Proc. COORDINATION*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011. `doi:10.1007/978-3-642-21464-6_7`.

[Nyg18]   Michael T Nygard. *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2018.

[RH97]   James Riely and Matthew Hennessy. Distributed Processes and Location Failures. In *Proc. ICALP*, volume 1256 of *LNCS*, pages 471–481. Springer, 1997. `doi:10.1007/3-540-63165-8_203`.

[RH01]   James Riely and Matthew Hennessy. Distributed processes and location failures. *Theor. Comput. Sci.*, 266(1-2):693–735, 2001. `doi:10.1016/S0304-3975(00)00326-1`.

[SFE10]   Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. A unified semantics for future Erlang. In *Proc. ACM SIGPLAN workshop on Erlang*, pages 23–32. ACM, 2010. `doi:10.1145/1863509.1863514`.

[SKÖ⁺15]   Zeynab Sabahi-Kaviani, Ramtin Khosravi, Peter Csaba Ölveczky, Ehsan Khamespanah, and Marjan Sirjani. Formal semantics and efficient analysis of timed rebeca in real-time maude. *Sci. Comput. Program.*, 113:85–118, 2015. `doi:10.1016/j.scico.2015.07.003`.

[Ste89]   Bernhard Steffen. Characteristic formulae. In *Proc. ICALP*, volume 372 of *LNCS*, pages 723–732. Springer, 1989. `doi:10.1007/BFb0035794`.

[SW01]   Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.

[Wya13]   Derek Wyatt. *Akka Concurrency*. Artima Incorporation, Sunnyvale, CA, USA, 2013.