# A model of actors and grey failures[⋆]

Laura Bocchi[1], Julien Lange[2], Simon Thompson[1], and A. Laura Voinea[1]

[1] University of Kent, Canterbury, UK
[2] Royal Holloway, University of London, Egham, 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, 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 to inject in the system. Our failure model captures not only fail-stop node and link failures, but also grey failures (e.g., partial, 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 accurately capture. Several works contribute towards a better account of unpredictability, for example in the context of process calculi (also including session types) by extending calculi to model node failures [19,41], link failures [2], a combination of link and node failures [6], as well as programmatic constructs to deal with failures like escapes [15], interrupts [27], exceptions [20], and timeouts [31,7,32]. Most existing models assume a fail-stop model of failure, where processes are either working or permanently stopped, and their state 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, overload). 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 [24]. The symptoms

[⋆] This work has been partially supported by EPSRC project EP/T014512/1 (STARDUST) and the BehAPI project funded by the EU H2020 RISE under the Marie Sklodowska-Curie action (No: 778233).

of grey failure tend to be ambiguous. In a distributed system, processes may have different perceptions as to the state of health of the system (aka *differential observation*) [28]. Grey failures tend to be behind many service incidents in cloud systems and traditional fault tolerance mechanisms tend to be ineffective or counterproductive [28]. Diagnosis is challenging and lengthy, for example the work in [33] 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 [45], 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 [48].

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. To capture the ambiguity of symptoms of grey failure we assume actors have no knowledge on the state of health of other actors. However, actors can observe the presence (or absence) of messages in their own mailboxes and hence the effects of failure in terms of missed communications. In Erlang, a key mechanism for detecting 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 its effects 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.

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 6 discusses conclusions and related 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) [25], a well understood extension of CCS with discrete time and timeouts. Delays are processes of the form $\mathbf{sleep}.P$ 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 [25]). Concretely, we define the systems behaviour as a reduction relation with two kinds of actions: communication actions $\rightharpoonup$ and time actions $\rightsquigarrow\!\triangleright$. 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 $\mathtt{n}[\,P\,](M)(t)$, where $\mathtt{n}$ 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 $\mathtt{n}_1$ and $\mathtt{n}_2$:

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

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 $\mathtt{n}_1$ is deliberately idling and $\mathtt{n}_2$ 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 $\mathtt{n}_2$:

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

*Mailboxes.* Each pair of actors can communicate via two unidirectional links. For example, $(\mathtt{n}_1, \mathtt{n}_2)$ denotes the link for communications from $\mathtt{n}_1$ to $\mathtt{n}_2$. An interaction involve 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 $\mathbf{R}_c = \mathtt{n}_1[\,!\mathtt{a}.0\,](\emptyset)(t) \parallel \mathtt{n}_2[\,?\mathtt{a}.P\ \mathtt{after}\ \ 2\ Q\,](\mathtt{b})(t)$. Step (I), the sending of a message, is illustrated below on $\mathbf{R}_c$:

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

$1.(\mathtt{n}_1,\mathtt{n}_2,\mathtt{a})$ models a latent message in link $(\mathtt{n}_1,\mathtt{n}_2)$ with content $\mathtt{a}$. 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:

$$\mathbf{R}'_c \rightsquigarrow\!\!\triangleright \mathtt{n}_1[\,0\,](\emptyset)(t+1) \parallel (\mathtt{n}_1,\mathtt{n}_2,\mathtt{a}) \parallel \mathtt{n}_2[\,?\mathtt{a}.P\ \mathtt{after}\ \ 1\ Q\,](\emptyset)(t+1) \tag{2}$$

These floating messages $(\mathtt{n}_1,\mathtt{n}_2,\mathtt{a})$ with no latency are similar to messages in the ether [47], in the global mailbox [29], or to the floating messages in [30].

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 $\mathtt{n}_2$:

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

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

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

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) [13] and Multiparty Session Types [26,18], 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. Note that, in the interaction above, $\mathtt{n}_2$ processes message $\mathtt{a}$ because it matches pattern $\mathtt{a}$, although an older message $\mathtt{b}$ is present in the mailbox. Alternative models, like Mailbox CFSMs [5,10], 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 in an unexpected 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 [38].

*Localities and failures.* The actor construct is similar to that used to model locality for processes [16], and also studied in relation to failures [6,42,21,22] 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 (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 $t$:

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

the floating message gets lost which in turn would end up causing a timeout in $\mathtt{n}_2$. Similarly, in case of node failure, node $\mathtt{n}_1$ in system $\mathbf{R}_c$, seen earlier in (1), would go into a crashed node state before sending the message, hence triggering a timeout in $\mathtt{n}_2$:

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

*Assumptions.* When a node crashes and comes back up again later on, it will come up with the same node identifier. We assume the 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 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. 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 $\pi$-calculus restriction.

## 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 $\varDelta$ function:

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

mapping each discrete time $t \in \mathbb{N}$, node $\mathtt{n} \in \mathcal{N}$, and link $(\mathtt{n}_1, \mathtt{n}_2) \in \mathcal{N} \times \mathcal{N}$ 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 $\circlearrowright$ indicates a node or link slowdown. The failure scenarios covered by $\varDelta$ include node crash, message loss, slow processes or slow networks. If *node* $\mathtt{n}$ *is down* at time $t$, written $\varDelta(t)(\mathtt{n}) = \downarrow$, then it will perform no action until it is resumed, if ever. If $\mathtt{n}$ is resumed at time $t'$, then its state at time $t'$ will be set to the initial state (see Definition 5 for the formal definition). If *link* $(\mathtt{n}_1, \mathtt{n}_2)$ *is down* at time $t$, written $\varDelta(t)(\mathtt{n}_1, \mathtt{n}_2) = \downarrow$, then any message in transit on that link at time $t$ will be lost. If *node* $\mathtt{n}$ *is slow* at time $t$, written $\varDelta(t)(\mathtt{n}) = \circlearrowright$, then any actions of the process running in $\mathtt{n}$ are delayed for one time step, and may resume at time $t+1$ if $\varDelta(t+1)(\mathtt{n}) = \uparrow$. If *link* $(\mathtt{n}_1, \mathtt{n}_2)$ *is slow* at time $t$, written $\varDelta(t)(\mathtt{n}_1, \mathtt{n}_2) = \circlearrowright$, 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. The delay is in effect added to the average network latency, which we model as a constant. Failures can be permanent or transient, as shown below by examples.

Systems

| | | | |
|---|---|---|---|
| $\mathbf{R}$ | $::=$ | $\mathtt{n}[\,P\,](M)(t)$ | node |
| | $\mid$ | $(\mathtt{n}_1, \mathtt{n}_2, m)(t)$ | floating message |
| | $\mid$ | $u.(\mathtt{n}_1, \mathtt{n}_2, m)(t)$ | latent message |
| | $\mid$ | $\mathtt{n}[\,\downarrow\,](\emptyset)(t)$ | crashed node |
| | $\mid$ | $\emptyset$ | empty |
| | $\mid$ | $\mathbf{R} \,\|\, \mathbf{R}$ | parallel |

Processes

| | | | |
|---|---|---|---|
| $P$ | $::=$ | $!\{\mathtt{n_i}\; m_i.P_i\}_{i \in I}$ | send |
| | $\mid$ | $?\{p_i.P_i\}_{i \in I}\ \mathtt{after}\ P$ | receive-timeout |
| | $\mid$ | $\mathbf{sleep}.P$ | sleep |
| | $\mid$ | $\mu\mathtt{t}.P$ | fixed-point |
| | $\mid$ | $\mathtt{t}$ | recursive variable |
| | $\mid$ | $\mathbf{0}$ | inaction |

Values

| | | | |
|---|---|---|---|
| $V$ | $::=$ | $a$ | atom |
| | $\mid$ | $\mathtt{n}$ | node id |
| | $\mid$ | $X$ | variable |

Message

| | | | |
|---|---|---|---|
| $m$ | $::=$ | $\widetilde{V}$ | message tuple |

Mailbox

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

Receive Patterns

| | | | |
|---|---|---|---|
| $E$ | $::=$ | $X \mid a$ | pattern element |
| $p$ | $::=$ | $\widetilde{E}$ | pattern tuple |

Fig. 1: Syntax

*Example 1 (Permanent failures).* Permanent node failure after a certain point in time, say $t = 10$, can be modelled by the $\Delta_1$ definition below. Function $\Delta_2$ shows a transient periodic structural failure of node $\mathtt{n}$, with each period having 100 time units of healthy state and 100 of down state. One could similarly model transient degrading failure by setting uptimes when $t = n^2$ for $(n \in \mathbb{N})$.

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

## 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 $\mathtt{n}[\,P\,](M)(t)$, messages (floating or latent), crashed nodes $\mathtt{n}[\,\downarrow\,](\emptyset)(t)$, empty systems $\emptyset$, and parallel compositions of systems $\mathbf{R} \,\|\, \mathbf{R}$. Term $\mathtt{n}[\,P\,](M)(t)$ denotes the state of node $\mathtt{n} \in \mathcal{N}$ where $P$ is the process running in $\mathtt{n}$, 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 [38] through the derivations in Figure 2.

Given a pattern $\widetilde{E}$ and a message (tuple) $\widetilde{V}$, $(\widetilde{E}, \widetilde{V}) \vdash_{\text{match}} \sigma$ the match function returns a substitution $\sigma$. Note that the matching is only defined if $\widetilde{E}$ and $\widetilde{V}$ have the same size, and if the pattern and message match. We write $(E, m) \nvdash_{\text{match}}$ when message $m$ does *not* match pattern $E$.

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

$$[\textsc{Var1}]\ (X,a) \vdash_{\text{match}} [a/X] \qquad [\textsc{Var2}]\ (X,\mathtt{n}) \vdash_{\text{match}} [\mathtt{n}/X]$$

$$[\textsc{Atom}]\ (a,a) \vdash_{\text{match}} [a/a] \qquad [\textsc{Tuple}]\ \frac{(E,V) \vdash_{\text{match}} \underline{\sigma} \qquad (\widetilde{E},\widetilde{V}) \vdash_{\text{match}} \sigma}{(E\widetilde{E}, V\widetilde{V}) \vdash_{\text{match}} \underline{\sigma}\sigma}$$

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

Fig. 2: Matching rules

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 $?\{p_i.P_i\}_{i \in I}\ \mathtt{after}\, P$ 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 `after` one time unit if no message matches and executing $P$. Process **sleep**.$P$ consumes a time unit and then continues as $P$. Process $\mu\mathtt{t}.P$ is for recursion, and $\mathtt{t}$ is recursion call. Finally, $\mathbf{0}$ is the idle process.

*Remark 1.* We use notation $?\{p_i.P_i\}_{i \in I}\ \mathtt{after}\ u\, P$ as syntactic sugar for nesting $u$ timeouts[3] and **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 $\Delta$ 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 1 (Cursed system).** *A cursed system is a pair* $(\mathbf{R}, \Delta)$ *where* $\mathbf{R}$ *is a system,* $\Delta$ *is a curse.*

The semantics of cursed systems is given in Def. 2 as a reduction relation over systems that is parametric on $\Delta$. 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 2 (Operational semantics for cursed systems).** *Reduction is the smallest relation on cursed systems over communication actions denoted by* $\rightharpoonup$*, and time actions denoted by* $\rightsquigarrow\!\triangleright$*, that satisfies the rules in Figure 3. We use* $\rightarrow$ *when* $\rightarrow \in \{\rightharpoonup, \rightsquigarrow\!\triangleright\}$*. For readability, in the rules we assume* $\Delta$ *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

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

$$[\textsc{Snd}]\ \frac{\Delta(t)(\mathtt{n}_1) = \uparrow \qquad j \in I}{\mathtt{n}[\,!\{\mathtt{n_i}\ m_i.P_i\}_{i\in I}\,](M)(t) \rightharpoonup \mathtt{n}[\,P_j\,](M)(t) \,||\, L.(\mathtt{n}, \mathtt{n_j}, m_j)(t)}$$

$$[\textsc{Sched}]\ \frac{\Delta(t)(\mathtt{n}_1) = \uparrow \qquad \Delta(t)(\mathtt{n}_2, \mathtt{n}_1) = \uparrow}{(\mathtt{n}_2, \mathtt{n}_1, m)(t) \,||\, \mathtt{n}_1[\,P\,](M)(t) \rightharpoonup \mathtt{n}_1[\,P\,](M \cdot m)(t)}$$

$$[\textsc{Rcv}]\ \frac{\Delta(t)(\mathtt{n}) = \uparrow \qquad j \in I,\ (p_j, m) \vdash_{\text{match}} \sigma \qquad \forall i \in I,\ (p_i, M_1) \not\vdash_{\text{match}}}{\mathtt{n}[\,?\{p_i.P_i\}_{i\in I}\ \mathtt{after}\ P\,](M_1 \cdot m \cdot M_2)(t) \rightharpoonup \mathtt{n}[\,P_j\sigma\,](M_1 \cdot M_2)(t)}$$

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

(a) Actor/Node actions

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

$$[\textsc{Latency}]\ \frac{\Delta(t)(\mathtt{n}_1, \mathtt{n}_2) = \uparrow \qquad u' = \mathtt{max}(u-1, 0)}{u.(\mathtt{n}_1, \mathtt{n}_2, m)(t) \leadsto u'.(\mathtt{n}_1, \mathtt{n}_2, m)(t+1)}$$

$$[\textsc{Timeout}]\ \frac{\Delta(t)(\mathtt{n}) = \uparrow \qquad \forall i \in I,\ (p_i, M) \not\vdash_{\text{match}}}{\mathtt{n}[\,?\{p_i.P_i\}_{i\in I}\ \mathtt{after}\ P\,](M)(t) \leadsto \mathtt{n}[\,P\,](M)(t+1)}$$

(b) Time actions

$$[\textsc{NLate}]\ \frac{\Delta(t)(\mathtt{n}) = \circlearrowright}{\mathtt{n}[\,P\,](M)(t) \leadsto \mathtt{n}[\,P\,](M)(t+1)} \qquad [\textsc{MsgLoss}]\ \frac{\Delta(t)(\mathtt{n}_1, \mathtt{n}_2) = \downarrow \qquad u \geq 0}{u.(\mathtt{n}_1, \mathtt{n}_2, m)(t) \rightharpoonup \emptyset}$$

$$[\textsc{MsgLate}]\ \frac{\Delta(t)(\mathtt{n}_1, \mathtt{n}_2) = \circlearrowright \qquad u \geq 0}{u.(\mathtt{n}_1, \mathtt{n}_2, m)(t) \leadsto u.(\mathtt{n}_1, \mathtt{n}_2, m)(t+1)} \qquad [\textsc{NDown}]\ \frac{\Delta(t)(\mathtt{n}) = \downarrow}{\mathtt{n}[\,P\,](M)(t) \rightharpoonup \mathtt{n}[\,\downarrow\,](\emptyset)(t)}$$

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

(c) Failure actions

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

$$[\textsc{ParTime}]\ \frac{\mathbf{R}_1 \leadsto \mathbf{R}_1' \quad \mathbf{R}_2 \leadsto \mathbf{R}_2' \quad \forall i \in \{1, 2\}.\, \mathbf{R}_i \not\rightharpoonup}{\mathbf{R}_1 \,||\, \mathbf{R}_2 \leadsto \mathbf{R}_1' \,||\, \mathbf{R}_2'}$$

(d) System actions

Fig. 3: Reduction and structural equivalence

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 $\sigma$ that is applied to the continuation process $P_j$ associated with pattern $p_j$; and $m$ is removed from the mailbox. Finally, Rule [Rec] allows a node with a recursive process to proceed with a communication or a time action.

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 [39] 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 $\mathtt{n}[\,\mathbf{0}\,](\mathrm{M})(t)$.

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 initial process $P$ and empty mailbox. $\Sigma$ 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.

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 ($\mathbf{R}_i \not\rightharpoonup$). [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 $\Delta$.

**Definition 3 (Time of a system).** *Let $\underline{t}$ range over $\mathbb{N} \cup \{*\}$. We define the synchronization (partial) function $\delta$:*

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

*$\delta(\underline{t}_1, \underline{t}_2)$ 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:*

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

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

**Definition 4 (Time coherence).** $\mathbf{R}$ *is* time coherent *if* $\mathtt{time}(\mathbf{R})$ *is defined.*

For example, system $\mathtt{n}_1[\,P\,](M)(t) \parallel (\mathtt{n}_1, \mathtt{n}_2, m)(t) \parallel \emptyset$ is time-coherent, while system $\mathtt{n}_1[\,P\,](M)(t) \parallel (\mathtt{n}_1, \mathtt{n}_2, m)(t+1) \parallel \emptyset$ 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 5 (Initial system).** *Let $\Sigma$ be a mapping from $\mathcal{N}$ to processes such that $\Sigma(\mathtt{n})$ is the initial process of $\mathtt{n}$. A system $\mathbf{R}$ is initial if $\mathtt{time}(\mathbf{R}) = 0$ and*

$$\mathbf{R} \equiv \mathtt{n}_1[\,\Sigma(\mathtt{n}_1)\,](\emptyset)(0) \,||\, \dots \,||\, \mathtt{n}_\mathtt{m}[\,\Sigma(\mathtt{n}_m)\,](\emptyset)(0)$$

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

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

**Lemma 1 (Time-coherence invariant).** *If $\mathbf{R}$ is time-coherent and $\mathbf{R} \to \mathbf{R}'$ 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 $\mathit{time}(\mathbf{R})$ is defined (as 0) yields the following property. We let $\to^*$ be the transitive closure of the reduction relation.

*Property 1.* Let $\mathbf{R}$ be initial, if $\mathbf{R} \to^* \mathbf{R}'$ then $\mathbf{R}'$ is time-coherent.

We assume any system $\mathbf{R}$ to start off as initial and hence, by Prop. 1, 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 the set of reachable states from system to be finite. We start by defining a non-instantaneous process.

**Definition 6 (Non-instantaneous process).** *We define function* $\texttt{ninst}(P)$ *inductively as follows:*

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

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

*Property 2 (Non-zenoness).* Let $\mathbf{R}$ be non-instantaneous. If $\mathbf{R} \to^* \mathbf{R}'$ 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 $\mathbf{R}'$. Hereafter we assume systems to be non-instantaneous, and hence non-Zeno.

## 5 Properties of cursed systems

In this section we define a behavioural relation between cursed systems, as a weak barbed bisimulation [44]. 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 keep receiving the messages they expect. Mailbox-based (rather than point-to-point) communication and pattern matching allow us to capture e.g., multiple-producers 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 observe 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. On the basis of this equivalence we define *recoverability* and *augmentation*.

We start by defining weak barbed bisimulation for cursed systems.

**Definition 7 (Barb).** *The ready actions of* $P$ *are defined inductively as follows:*

$$\begin{array}{ll} \texttt{rdy}(!\{\mathtt{n_i}\, m_i.P_i\}_{i \in I}) = \{!\,\mathtt{n}_i\, m_i\}_{i \in I} & \texttt{rdy}(?\{p_i.P_i\}_{i \in I}\ \texttt{\textit{after}}\ P) = \{?\, p_i\}_{i \in I} \\ \texttt{rdy}(0) = \texttt{rdy}(\mathtt{t}) = \texttt{rdy}(\textbf{sleep}.P) = \emptyset & \texttt{rdy}(\mu \mathtt{t}.P) = \texttt{rdy}(P) \end{array}$$

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

$$\begin{array}{ll} \mathtt{n}[\,P\,](M)(t) \downarrow x & \textit{if } !\,\mathtt{n}'m \in \texttt{rdy}(P) \wedge x = !\,\mathtt{n}'m \ \vee \ ?\,p \in \texttt{rdy}(P) \wedge x = ?\,\mathtt{n}\,p \\ (\mathtt{n}_1, \mathtt{n}_2, m) \downarrow !\,\mathtt{n}_2\, m & \\ (\mathbf{R}_1 \parallel \mathbf{R}_2) \downarrow x & \textit{if } \mathbf{R}_1 \downarrow x \textit{ or } \mathbf{R}_2 \downarrow x \end{array}$$

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

Barbs abstract from 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. We observe $m$ and $p$ to retain expressiveness with respect to channel-based scenarios, as discussed in Section 5.3.

**Definition 8 (Weak barbed bisimulation).** *Recall* $\rightarrow \in \{\rightharpoonup, \rightsquigarrow\!\triangleright\}$. *A weak (time-abstract) barbed bisimulation is a symmetric 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* $(\mathbf{R}_2, \Delta_2) \rightarrow^* (\mathbf{R}'_2, \Delta_2)$ *and* $(\mathbf{R}'_1, \Delta_1)\,\mathcal{S}(\mathbf{R}'_2, \Delta_2)$.
2. *If* $\mathbf{R}_1 \downarrow x$ *for some* $x$, *then* $(\mathbf{R}_2, \Delta_2) \rightarrow^* (\mathbf{R}'_2, \Delta_2)$ *and* $\mathbf{R}'_2 \downarrow x$.

*and the symmetric of (1) and (2).* $(\mathbf{R}_1, \Delta_1)$ *is 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* $(\mathbf{R}_1, \Delta_1)\,\mathcal{S}(\mathbf{R}_2, \Delta_2)$.

### 5.1 Resilience and recoverability

We define resilience as the ability of a system to behave 'normally' despite failures injection. Let $\uparrow$ be the function that assigns $\uparrow$ to all nodes and links at any time.

**Definition 9 (Resilience).** *Initial* $(\mathbf{R}, \Delta)$ *is resilient if* $(\mathbf{R}, \uparrow) \approx (\mathbf{R}, \Delta)$.

The definition of resilience sets the behaviour of a system without curses as a model of the *expected behaviour*. In some cases, e.g. when looking at retry strategies, while the system may be affected by failures, one may want to observe that it *eventually* recovers. To this aim, we define $n$-recoverability as the ability of a system to display the expected behaviour after time $n$.

**Definition 10 ($n$-Recoverability).** *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* time$(\mathbf{R}') = n$ *implies* $(\mathbf{R}, \uparrow) \approx (\mathbf{R}', \Delta)$.

Basically, a system is resilient if it is 0-recoverable. We give some examples of resilience and $n$-recoverability, where we fix the latency $L = 1$.

*Example 2 (Resilience).* Consider the cursed system $(\mathbf{R}, \Delta)$ with:

$$\mathbf{R} = \mathtt{n}_1[\,\mathbf{sleep}.\,!\,\mathtt{n}_2\,a.0\,](\emptyset)(0) \,||\, \mathtt{n}_2[\,?\,a.\mathbf{sleep}.0 \;\mathtt{after}\; 5\,0\,](\emptyset)(0)$$

and $\Delta(\mathtt{n}_1, \mathtt{n}_2)$ injecting network delays at time 1 and 2 and $\uparrow$ otherwise. $(\mathbf{R}, \Delta)$ is resilient; the timeout of 5 is good for networks delays of 2 time units. However, $(\mathbf{R}, \Delta)$ would not be resilient for longer networks delays.

*Example 3 (n-recoverability).* Consider cursed system $(\mathbf{R}, \Delta)$ with:

$$\mathbf{R} = \mathtt{n}_1[\,\mathbf{sleep}.!\mathtt{n}_2\,a.0\,](\emptyset)(0) \,||\, \mathtt{n}_2[\,\mu\mathtt{t}.?a.\mathbf{sleep}.0 \;\mathtt{after}\; 4\,\mathtt{t}\,](\emptyset)(0)$$

and $\Delta(\mathtt{n}_1, \mathtt{n}_2)$ injecting network delays at time 1, 2, and 3 (and $\uparrow$ otherwise). $(\mathbf{R}, \Delta)$ is 5-resilient. Note that any behaviour by $\mathtt{n}_1$ before 5 is disregarded, even in cases where some communication occurred.

By Def. 10, checking resilience and $n$-recoverability is reduced to the problem of checking weak barbed bisimulation. Note that, in Def. 10, 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 Prop. 2, a system can perform only a finite number of actions at any given time.

### 5.2 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.

**Definition 11 (Augmentation).** *System $\mathbf{R}_{\mathtt{I}}$ is an augmentation of $\mathbf{R}$ if $time(\mathbf{R}_{\mathtt{I}}) = time(\mathbf{R})$ and: (i) $(\mathbf{R}, \uparrow) \approx (\mathbf{R}_{\mathtt{I}}, \uparrow)$* (transparency)*; (ii) there exist $\Delta$ and $n$ such that $(\mathbf{R}_{\mathtt{I}}, \Delta)$ is $n$-recoverable and $(\mathbf{R}, \Delta)$ is not $n$-recoverable* (improvement)*. Moreover, we say that an augmentation is* preserving *if, for all $n$ and $\Delta$, $(\mathbf{R}, \Delta)$ is $n$-recoverable implies $(\mathbf{R}_{\mathtt{I}}, \Delta)$ is $n$-recoverable.*

*Example 4 (Augmentation).* Consider the small producer-consumer system $\mathbf{R}$ below, composed of a producer node $\mathtt{n_p}$, a queue node $\mathtt{n_q}$, and a consumer node $\mathtt{n_c}$. 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 $L = 1$.

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

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

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

### 5.3 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 8. 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 12 (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 $!\,\mathtt{n}\,p \in N$, $(p, m) \nvdash_{\text{match}}$. If $\mathbf{R} \downarrow_N \; x$ we say that $\mathbf{R}$ has a $N$-scoped* barb *on $x$.*

We extend Def. 8 using $\downarrow_N$ instead of $\downarrow$, obtaining scoped weak-barbed bisimulation $\approx_N$, and Def. 11 to use $\approx_N$. This setting allow us to analyse producer consumer scenarios, or more complex ones, like the Circuit Breaker pattern [40] widely used in distributed systems.

*Example 5 (Circuit breaker).* Consider system $(\mathbf{R}, \mathit{\Delta})$ 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 $\mathtt{n_s}$:

$$\begin{aligned} \mathbf{R} = \;& \mathtt{n_c}[\,\mu\mathtt{t}.\,!\,\mathtt{n_s}\,\mathit{request}.\,?\,\mathit{reply}.\mathbf{sleep}.\mathtt{t}\ \mathtt{after}\ 4\,\mathbf{0}\,](\emptyset)(0) \parallel \\ & \mathtt{n_s}[\,\mu\mathtt{t}.\,?\,\mathit{request}.\mathbf{sleep}.\,!\,\mathtt{n_c}\,\mathit{reply}.\mathtt{t}\ \mathtt{after}\ 4\,\mathtt{t}\,](\emptyset)(0) \end{aligned}$$

$$\begin{aligned} \mathbf{R_I} = \;& \mathtt{n_c}[\,\mu\mathtt{t}.\,!\,\mathtt{n_s}\,\mathit{request}.\,?\{\mathit{reply}.\mathbf{sleep}.\mathtt{t},\ \mathit{ko}.P_f\}\ \mathtt{after}\ 8\,\mathbf{0}\,](\emptyset)(0) \parallel \\ & \mathtt{n_s}[\,\mu\mathtt{t}.\,?\,X_1.!\mathtt{n_1}\,X_1.\,?\,X_2.!\mathtt{n_c}\,X_2.\mathtt{t}\ \mathtt{after}\ 4\,P'_f\ \mathtt{after}\ 4\,\mathtt{t}\,](\emptyset)(0) \parallel \\ \mathtt{n_1}[\,\mu\mathtt{t}.\,?\{&\mathit{request}.\mathbf{sleep}.!\mathtt{n_s}\,\mathit{reply}.\mathtt{t},\ \mathit{ruok}.\mathbf{sleep}.!\mathtt{n_s}\,\mathit{imok}.\mathtt{t}\}\ \mathtt{after}\ 6\,\mathtt{t}\,](\emptyset)(0) \\ P_f = \;& \mu\mathtt{t}'.\,?\,\mathit{retry}.\mathbf{sleep}.\mathtt{t}\ \mathtt{after}\ 5\,\mathtt{t}' \end{aligned}$$

$$P'_f = \;!\,\mathtt{n_c}\,\mathit{ko}.\mathbf{sleep}.\mu\mathtt{t}'.\,!\,\mathtt{n_s}\,\mathit{ruok}.\,?\,\mathit{imok}.\mathbf{sleep}.\,!\,\mathtt{n_c}\,\mathit{retry}.\mathtt{t}\ \mathtt{after}\ 3\,\mathtt{t}'$$

with a $\mathit{\Delta}(\mathtt{n_c}, \mathtt{n_s})$ injecting link slow ↻ 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. 12 with: $N = \{!\,\mathtt{n_s}\,\mathit{ruok},\ ?\,\mathtt{n_s}\,\mathit{imok},\ ?\,\mathtt{n_s}\,\mathit{reply},\ ?\,\mathtt{n_1}\,\mathit{request},\ !\,\mathtt{n_s}\,\mathit{reply},\ ?\,\mathtt{n_1}\,\mathit{ruok},\ !\,n_s\,\mathit{imok},$ $!\,n_c\,\mathit{ko},\ !\,n_c\,\mathit{retry},\ ?\,n_c\,\mathit{ko},\ ?\,n_c\,\mathit{retry}\}$. 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.

## 6 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 $\Sigma$ to model checkpointing of intermediate node states. Note that $\Sigma$ 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 [23,46], 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 [1], 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 [1] and Real-Time Maude [43] 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 [21], Francalanza and Hennessy introduced a behavioural theory for D$\pi$F, a distributed $\pi$-calculus with with nodes and links failures. For a subset of D$\pi$F, they also developed a notion of fault-tolerance up to $n$-faults [22], which is preserved by contexts, and which is related to our notion of resilience. The behavioural theory in [21] 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 the network, e.g., as to accessibility and reachability of other nodes. The scoped barbs in Section 5.3 have the similar purpose of hiding augmentations while observing their effects on recoverability. However, because of asynchronous communication over mailboxes (while D$\pi$F 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. D$\pi$F studies partial failures but does not consider transient failures and time. On the other hand, D$\pi$F 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 [31,7,32], mailboxes [38], localities [41][6][16]) 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. [9], where different observation criteria are studied (bisimulation, traces and abstract traces) following the axiomatic approach typical of the process algebra ACP [8]. An alternative approach has been followed by Amadio et al. [4] who defined asynchronous bisimulation for the $\pi$-calculus [36]. 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 [37]. 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 [34] have subsequently studied barbed congruence in the context of the Asynchronous Localised $\pi$-calculus (AL$\pi$), a fragment of the asynchronous $\pi$-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 [17] and by Boreale et al. [11,12]. These papers consider an asynchronous variant of CCS [35]. Testing discriminates less than our equivalence, concerning choice, and observes divergent behaviours which we abstract from. Lanese et al. [30] look at bisimulation for Erlang, focussing on the management of process ids. Besides the aforementioned work by Francalanza and Hennessy [21,22], several works look at distributed process algebras with unreliable communication due to faults in the underlying network. Riely and Hennessy [41] study behavioural equivalence over process calculi with locations. Amadio [3] extends the $\pi$-calculus with located actions, in the context of a higher-order distributed programming language. Fournet et al. [19] 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 [6] augment the asynchronous $\pi$-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. [14] 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.

## References

1. Aceto, L., Cimini, M., Ingolfsdottir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using timed rebeca. EPTCS **58**, 1–19 (2011). https://doi.org/10.4204/eptcs.58.1
2. Adameit, M., Peters, K., Nestmann, U.: Session types for link failures. In: Proc. FORTE. pp. 1–16. Springer International Publishing (2017). https://doi.org/10.1007/978-3-319-60225-7_1
3. Amadio, R.M.: An asynchronous model of locality, failure, and process mobility. In: Garlan, D., Le Métayer, D. (eds.) Proc. Coordination Models and Languages. pp. 374–391. Springer (1997). https://doi.org/10.1007/3-540-63383-9_92
4. Amadio, R.M., Castellani, I., Sangiorgi, D.: On bisimulations for the asynchronous pi-calculus. Theor. Comput. Sci. **195**(2), 291–324 (1998). https://doi.org/10.1016/S0304-3975(97)00223-5
5. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. Proc. ACM Program. Lang. **47**(POPL), 191–202 (2012). https://doi.org/10.1145/2103656.2103680
6. Berger, M., Honda, K.: The two-phase commitment protocol in an extended $\pi$-calculus. ENTCS **39**(1), 21–46 (2003). https://doi.org/10.1016/S1571-0661(05)82502-2
7. Berger, M., Yoshida, N.: Timed, distributed, probabilistic, typed processes. In: Proc. APLAS. LNCS, vol. 4807, pp. 158–174. Springer (2007). https://doi.org/10.1007/978-3-540-76637-7_11
8. Bergstra, J.A., Klop, J.W.: Process algebra for synchronous communication. Inf. Control. **60**(1-3), 109–137 (1984). https://doi.org/10.1016/S0019-9958(84)80025-X
9. de Boer, F.S., Klop, J.W., Palamidessi, C.: Asynchronous communication in process algebra. In: Proc. LICS. pp. 137–147. IEEE Computer Society (1992). https://doi.org/10.1109/LICS.1992.185528
10. Bollig, B., Giusto, C.D., Finkel, A., Laversa, L., Lozes, É., Suresh, A.: A unifying framework for deciding synchronizability. In: Proc. CONCUR. LIPIcs, vol. 203, pp. 14:1–14:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.CONCUR.2021.14
11. Boreale, M., Nicola, R.D., Pugliese, R.: A theory of "may" testing for asynchronous languages. In: Proc. FoSSaCS. LNCS, vol. 1578, pp. 165–179. Springer (1999). https://doi.org/10.1007/3-540-49019-1_12
12. Boreale, M., Nicola, R.D., Pugliese, R.: Trace and testing equivalence on asynchronous processes. Inf. Comput. **172**(2), 139–164 (2002). https://doi.org/10.1006/inco.2001.3080
13. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983). https://doi.org/10.1145/322374.322380
14. Cano, M., Castellani, I., Di Giusto, C., Pérez, J.A.: Multiparty Reactive Sessions. Research Report 9270, INRIA (Apr 2019), `https://hal.archives-ouvertes.fr/hal-02106742`
15. Capecchi, S., Giachino, E., Yoshida, N.: Global escape in multiparty sessions. MSCS **26**(2), 156–205 (2016). https://doi.org/10.1017/S0960129514000164
16. Castellani, I.: Process algebras with localities. In: Handbook of Process Algebra, pp. 945–1045. North-Holland / Elsevier (2001). https://doi.org/10.1016/b978-044482830-9/50033-3
17. Castellani, I., Hennessy, M.: Testing theories for asynchronous languages. In: Proc. FSTTCS. LNCS, vol. 1530, pp. 90–101. Springer (1998). https://doi.org/10.1007/978-3-540-49382-2_9

18. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. MSCS **26**(2), 238–302 (2016). https://doi.org/10.1017/S0960129514000188
19. Fournet, C., Gonthier, G., Lévy, J., Maranget, L., Rémy, D.: A calculus of mobile agents. In: Proc. CONCUR. LNCS, vol. 1119, pp. 406–421. Springer (1996). https://doi.org/10.1007/3-540-61604-7_67
20. Fowler, S., Lindley, S., Morris, J.G., Decova, S.: Exceptional asynchronous session types: session types without tiers. Proc. ACM Program. Lang. **3**(POPL), 1–29 (2019). https://doi.org/10.1145/3290341
21. Francalanza, A., Hennessy, M.: A theory for observational fault tolerance. JLAMP **73**(1-2), 22–50 (2007). https://doi.org/10.1007/11690634_2
22. Francalanza, A., Hennessy, M.: A theory of system behaviour in the presence of node and link failure. Inf. Comput. **206**(6), 711–759 (2008). https://doi.org/10.1016/j.ic.2007.12.002
23. Graf, S., Sifakis, J.: A modal characterization of observational congruence on finite terms of CCS. Inf. Control. **68**(1-3), 125–145 (1986). https://doi.org/10.1016/S0019-9958(86)80031-6
24. Gunawi, H.S., Suminto, R.O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., Srinivasan, D., Panda, B., Baptist, A., Grider, G., Fields, P.M., Harms, K., Ross, R.B., Jacobson, A., Ricci, R., Webb, K., Alvaro, P., Runesha, H.B., Hao, M., Li, H.: Fail-slow at scale: Evidence of hardware performance faults in large production systems. ACM Trans. Storage **14**(3), 23:1–23:26 (2018). https://doi.org/10.1145/3242086
25. Hennessy, M., Regan, T.: A process algebra for timed systems. Inf. Comput. **117**(2), 221–239 (1995). https://doi.org/10.1006/inco.1995.1041
26. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9:1–9:67 (2016). https://doi.org/10.1145/2827695
27. Hu, R., Neykova, R., Yoshida, N., Demangeon, R., Honda, K.: Practical interruptible conversations. In: Runtime Verification. pp. 130–148. Springer (2013). https://doi.org/10.1007/978-3-642-40787-1_8
28. Huang, P., Guo, C., Zhou, L., Lorch, J.R., Dang, Y., Chintalapati, M., Yao, R.: Gray failure: The achilles' heel of cloud-scale systems. In: Proc. HotOS. pp. 150–155. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3102980.3103005
29. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. JLAMP **100**, 71–97 (2018). https://doi.org/10.1016/j.jlamp.2018.06.004
30. Lanese, I., Sangiorgi, D., Zavattaro, G.: 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. LNCS, vol. 11665, pp. 71–91. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-21485-2_6
31. Laneve, C., Zavattaro, G.: Foundations of web transactions. In: Proc. FoSSaCS. LNCS, vol. 3441, pp. 282–298. Springer (2005). https://doi.org/10.1007/978-3-540-31982-5_18
32. López, H.A., Pérez, J.A.: Time and exceptional behavior in multiparty structured interactions. In: Proc. WS-FM. LNCS, vol. 7176, pp. 48–63. Springer (2011). https://doi.org/10.1007/978-3-642-29834-9_5
33. Lou, C., Huang, P., Smith, S.: Understanding, detecting and localizing partial failures in large system software. In: NDSI. pp. 559–574. USENIX Association (2020), https://www.usenix.org/conference/nsdi20/presentation/lou

34. Merro, M., Sangiorgi, D.: On asynchrony in name-passing calculi. In: Proc. ICALP. LNCS, vol. 1443, pp. 856–867. Springer (1998). https://doi.org/10.1007/BFb0055108
35. Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)
36. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, I. Inf. Comput. **100**(1), 1–40 (1992). https://doi.org/10.1016/0890-5401(92)90008-4
37. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Proc. ICALP. LNCS, vol. 623, pp. 685–695. Springer (1992). https://doi.org/10.1007/3-540-55719-9_114
38. Mostrous, D., Vasconcelos, V.T.: Session typing for a Featherweight Erlang. In: Proc. Coordination Models and Languages. pp. 95–109. Springer (2011). https://doi.org/10.1007/978-3-642-21464-6_7
39. Murgia, M.: Input urgent semantics for asynchronous timed session types. JLAMP **107**, 38–53 (2019). https://doi.org/doi.org/10.1016/j.jlamp.2019.04.001
40. Nygard, M.T.: Release it!: design and deploy production-ready software. Pragmatic Bookshelf (2018)
41. Riely, J., Hennessy, M.: Distributed Processes and Location Failures. In: Proc. ICALP. LNCS, vol. 1256, pp. 471–481. Springer (1997)
42. Riely, J., Hennessy, M.: Distributed processes and location failures. Theor. Comput. Sci. **266**(1-2), 693–735 (2001). https://doi.org/10.1016/S0304-3975(00)00326-1
43. Sabahi-Kaviani, Z., Khosravi, R., Ölveczky, P.C., Khamespanah, E., Sirjani, M.: Formal semantics and efficient analysis of Timed Rebeca in Real-Time Maude. Science of Computer Programming **113**, 85–118 (2015). https://doi.org/10.1016/j.scico.2015.07.003
44. Sangiorgi, D., Walker, D.: The $\pi$-calculus: a Theory of Mobile Processes. Cambridge University Press (2001)
45. Sankar, K.: *Programming Erlang - Software for a Concurrent World* by Joe Armstrong, Pragmatic Bookshelf, 2007, p. 536. ISBN-10: 193435600x. J. Funct. Program. **19**(2), 259–261 (2009). https://doi.org/10.1017/S0956796809007163
46. Steffen, B.: Characteristic formulae. In: Proc. ICALP. LNCS, vol. 372, pp. 723–732. Springer (1989). https://doi.org/10.1007/BFb0035794
47. Svensson, H., Fredlund, L., Earle, C.B.: A unified semantics for future Erlang. In: Proc. ACM SIGPLAN workshop on Erlang. pp. 23–32. ACM (2010). https://doi.org/10.1145/1863509.1863514
48. Wyatt, D.: Akka Concurrency. Artima Incorporation, Sunnyvale, CA, USA (2013)