Towards choreographic-based monitoring*

Adrian Francalanza[1], Claudio Antares Mezzina[2], and Emilio Tuosto[3,4]

[1] University of Malta, Malta
[2] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Italy
[3] Gran Sasso Science Institute, L'Aquila, Italy
[4] University of Leicester, UK

Abstract. Distributed programs are hard to get right because they are required to be open, scalable, long-running, and dependable. In particular, the recent approaches to distributed software based on (micro-) services, where different services are developed independently by disparate teams, exacerbate the problem. Services are meant to be composed together and run in open contexts where unpredictable behaviours can emerge. This makes it necessary to adopt suitable strategies for monitoring the execution and incorporate recovery and adaptation mechanisms so to make distributed programs more flexible and robust. The typical approach that is currently adopted is to embed such mechanisms within the program logic. This makes it hard to extract, compare and debug. We propose an approach that employs formal abstractions for specifying failure recovery and adaptation strategies. Although implementation agnostic, these abstractions would be amenable to algorithmic synthesis of code, monitoring, and tests. We consider message-passing programs (a la Erlang, Go, or MPI) that are gaining momentum both in academia and in industry. We first propose a model which abstracts away from three aspects: the definition of formal behavioural models encompassing failures; the specification of the relevant properties of adaptation and recovery strategy; and the automatic generation of monitoring, recovery, and adaptation logic in target languages of interest. To show the efficacy of our model, we give an instance of it by introducing *reversible choreographies* to express the normal forward behaviour of the system and the condition under which adaptation has to take place. Then we show how it is possible to derive Erlang code directly from the global specification.

1 Introduction

Distributed applications are notoriously complex and guaranteeing their correctness, robustness, and resilience is particularly challenging. These reliability requirements cannot be tackled without considering the problems that are not

* Research partly supported by the EU H2020 RISE programme under the Marie Skłodowska-Curie grant agreement No 778233 and by COST Action IC1405 on Reversible Computation - Extending Horizons of Computing. The second author has been partially supported by the French National Research Agency (ANR), project DCore n. ANR-18-CE25-0007.

generally encountered when developing *non*-distributed software. In particular, the execution and behaviour of distributed applications is characterised by a number of factors, a few of which we discuss below:

- Firstly, communication over networks is subject to *failures* (hardware or software) and to *security* concerns: nodes may crash or undergo management operations, links may fail or be temporarily unavailable, access policies may modify the connectivity of the system.
- Secondly, *openness* —a key requirement of distributed applications— introduces other types of failures. A paradigmatic example are (micro-)service architectures where distributed components dynamically bind and execute together. In this context, failures in the communication infrastructures are possibly aggravated by those due to services' unavailability, their (behavioural) incompatibility, or to unexpected interactions emerging from unforeseen compositions.
- Also, distributed components may belong to different administrative domains; this may introduce unexpected changes to the interaction patterns that may not necessarily emerge at design time. In addition, unforeseen behaviour may emerge because components may evolve independently (e.g., the upgrade of a service may hinder the communication with partner services).
- Another element of concern is that it is hard to determine the causes of errors, which in turn complicates efforts to rectify and/or mitigate the damage via recovery procedures. Since the boundary of an application are quite "fluid", it becomes infeasible to track and confine errors whenever they emerge. These errors are also hard to reproduce for debugging purposes, and some of them may even constitute instances of Heisenbugs [27].

For the above reasons (and others), developers have to harness their software with mechanisms that ensure (some degree of) dependability. For instance, the use of monitors capable of detecting failures and triggering automated countermeasures can avoid catastrophic crashes in distributed settings [24]. The typical mechanisms to foster reliability are redundancy (typically to tackle hardware failures) and exception handling for software reliability. It has been observed (see e.g., [42]) that the use of exception handling mechanisms naturally leads to defensive approaches in software development. For instance, network communications in languages such as Java require to extensively cast code in try-catch blocks in order to deal with possible exceptions due to communications. This muddles the main program logic with auxiliary logic related to error handling. Defensive programming, besides being inelegant, is not appealing; in fact, it requires developers to entangle the application-specific software with the one related to recovery procedures.

We advocate the use of choreographies to specify, analyse, and implement reliable strategies for recovery and monitoring of distributed message-passing applications. We strive towards a setup that teases apart the main program logic from the coordination of error detection, correction and recovery. The rest of the paper motivates our approach: Section 2 further introduces our motivations, Section 3 presents our (abstract) model by posing some research challenges, while

Sections 4 to 6 provide and instance of such model. We draw some conclusions in Section 7.

Disclaimer. This paper gathers the results obtained in [13, 23] with the intent to present them as a whole. In particular, the model presented in Section 3 is taken from [13], while Sections 4 to 6 are adapted from [23]. These results were obtained during the COST Action IC1405 within the case study "Reversible Choreographies via Monitoring in Erlang" of the Working Group 4 on case studies. We thank Carla Ferreira and Ulrik Pagh Schultz for having wisely led such working group.

2 Motivation

We are interested in *message-passing* frameworks, *i.e.,* models, systems, and languages where distributed components coordinate by exchanging messages. One archetypal model of the message-passing paradigm is the *actor model* [5] popularised by industry-strength language implementations such as those found in Akka (for both Scala and Java) [46], Elixir [44], and Erlang [15]. In particular, one effective approach to fault-tolerance is the model adopted by Erlang.

Rather than trying to achieve absolute error freedom, Erlang's approach concedes that failures are hard to rule out completely in the setting of open distributed systems. Accordingly, Erlang-based program development takes into account the possibility of computation going wrong. However, instead of resorting to the usual defensive programming, it adopts the so-called "let it fail" principle. In place of intertwining the software realising the application logic with logic for handling errors and faults, Erlang proposes a supervisory model whereby components (*i.e.,* actors) are monitored within a hierarchy of independently-executing *supervisors* (which can be monitor for other supervisors themselves). When an error occurs within a particular component, it is quarantined by letting that component fail (in isolation); the absence of global shared memory of the actor model facilitates this isolation. Its supervisor is then notified about this failure, creating a traceable event that is useful for debugging. More importantly to our cause, this mechanism also allows the supervisor to take *remedial action* in response to the reported failure. For instance, the failing component may be restarted by the supervisor. Alternatively, other components that may have been contaminated by the error could also be terminated by the supervisor. Occasionally supervisors themselves fail in response to a supervised component failing, thus percolating the error to a higher level in the supervision hierarchy.

Erlang's model is an instance of a programming paradigm commonly termed as Monitor Oriented Programming (MOP) [35, 16]. It neatly separates the application logic from the recovery policy by encapsulating the logic pertaining to the recovery policy within the supervision structure encasing the application. Despite this clear advantage, the solution is not without its shortcomings. For instance, the Erlang supervision mechanism is still inherently tied to the constructs of the host language and it is hard to transfer to other technologies.

Despite it being localised within supervisor code, manual effort is normally still required to disentangle it from the context where it is defined in order to be understood in isolation. Also, the manual construction of logic associated with recovery is itself prone to errors.

We advocate for a recovery mechanism that sits at a higher level of abstraction than the bare metal of the programming language where it is deployed. In particular, we envisage the three challenges outlined below:

1. The explicit identification and design of recovery policies in a technology agnostic manner. This will facilitate the comprehension and understanding of recovery policies and allow for better separation of concerns during program development.
2. The automated code synthesis from high-level policy descriptions. There exsist only a handful of methods for recovery policy specification and these have limited support for the automatic generation of monitors that implement those policies.
3. The evaluation of recovery policies. We require automated techniques that allow us to ascertain the validity of recovery policies with respect to notions of recovery correctness. We are also unaware of many frameworks that permit policies to be compared with one another and thus determine whether one recovery policy is better than (or equivalent to) another one.

To the best of our knowledge, there is a lack of support to take up the first challenge. For instance, Erlang folklore's to recovery policies simply prescribes the "one-for-one" or the "one-for-all" strategies. Recently, Neykova and Yoshida have shown how better strategies are sometimes possible [40]. We note that the approach followed in [40] is based on simple yet effective choreographic models.

The second challenge somehow depends on the support one provides for the design and implementation of recovery strategies. A basic requirement of (good) abstract software models is that an artefact has a clear relationship with the other artefacts that it interacts with, possibly at different levels of abstraction. This constitutes the essence of model-driven design. The preservation of these clearly defined interaction-points (across different abstraction levels) is crucial for sound software refinement. Such a translation from one abstraction level to a more concrete one forms the basis for an actual "compilation" from one model to the other. In cases where such relations have a clear semantics, they can be exploited to verify properties of the design (and the implementation) as well as to transform models (semi-)automatically. In our case, we would expect runtime monitors to be derived from their abstract models, to ease the development process and allow developers to focus on the application logic (such as in [11, 6]).

Finally, the right abstraction level should provide the foundations necessary to develop formal techniques to analyse and compare recovery policies as outlined in our third challenge. The right abstraction level would also permit us to tractably apply these techniques to specific policy instances; these may either have been developed specifically for the policy formalism considered by the technique or obtained via reverse-engineering methods from a technology-specific

application. Possible examples that may be used as starting points for such an investigation are [20], where various pre-orders for monitor descriptions are developed, and [21] where intrinsic monitor correctness criteria such as consistent detections are studied.

3 The Model

We advocate that the development of recovery logic is *orthogonal* to the application logic, and this separation of concerns could induce separate development efforts which are, to a certain degree, independent from one another. Similar to the case for the application logic, we envisage global and local points of view for the recovery logic whereby the latter is attained by projecting the global strategy. Our approach is schematically described in Figure 1. The left-most part of the diagram illustrates the top-down approach of choreographies of the application logic described in Section 4.1. We propose to develop a similar approach for the recovery logic as depicted in the right-most part of Fig. 1, where the triangular shape for monitors evokes that monitors are possibly arranged in a complex structure (as e.g., the *hierarchy* of Erlang supervisors). In fact, we envisage that a local strategy could correspond to a subsystem of monitors as in the case of [10, 6] (unlike the choreographies for the application logic, where each local view typically yields one component).

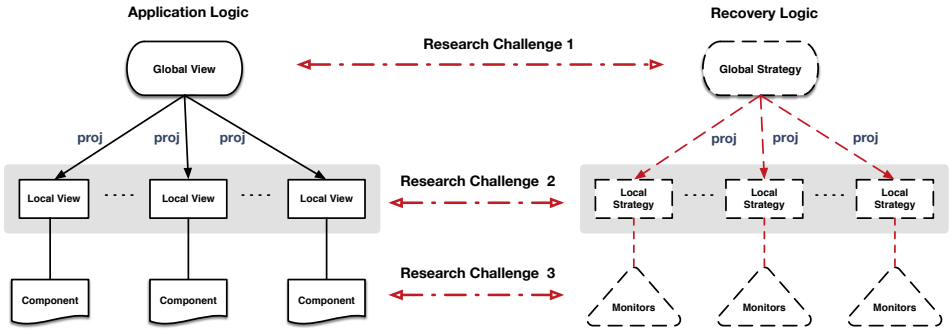


Fig. 1. A Global-Local approach to Adaptation Strategies incorporating the three Research Challenges identified in Section 2

Models to express global and local strategies. Choreographic models should be equipped with features allowing us to design and analyse the recovery logic of systems. This requires, on the one hand, the identification of suitable linguistic mechanisms for expressing global/local strategies and, on the other hand, to define principles of monitors programming by looking at state-of-the-art techniques. For example, the (global) recovery logic should allow us to specify *recov-*

ery points where parties can roll-back if some kind of error is met or *compensations* to activate when anomalous configurations are reached.

A challenge here is the definition of projection operations that enable featuring recovery mechanisms. A first step in this direction is a recent proposal of Mezzina and Tuosto [39] who extend the global graphs reviewed in Section 4.1 with *reversibility guards* to recover the system when it reaches undesired configurations. A promising research direction in this respect is to extend the language of reversibility guards with the patterns featured by adaptEr [10–12] and then define projection operations to automatically obtain adaptEr monitors.

Properties of recovery logic. We should understand general properties of interest of recovery as well as specific ones. One general property could be the fact that the strategy guides the application toward a *safe* state (*i.e.* stability envelope [35]) when errors occur. For example, the recovery strategy could guarantee *causal consistency*, namely that a safe state is one that the execution could have reached, possibly following a different interleaving of concurrent actions. Recovery strategies may be subject to resource requirements that need to be taken into consideration and/or adhered to. One such example would be the minimisation of the number of components that have to be re-started when a recovery procedure is administered, whereby the restarted components are causally related to the error detected. The work discussed in [10, 11] provides another example of resource requirements for recovery strategies: in an asynchronous monitoring setting, component synchronisations are considered to be expensive operations and, as a result, the monitors are expected to use the least number of component synchronisations for the adaptation actions to be administered correctly.

Also, as typical for choreographies, we should unveil the conditions under which a recovery strategy is realisable in a distributed settings. In other words, not all globally-specified recovery policies are necessarily implementable in a choreographed distributed setting; we therefore seek to establish *well-formedness* criteria that allow us to determine when a global recovery policy can be projected (and thus implemented) in a decentralised setup.

Compliance. In the case of recovery strategies, it is unclear when monitors are deemed to be compliant with their local strategy. A central aspect that we should tackle is that of understanding what it actually means for monitors and local strategy to be compliant, and subsequently to give a suitable compliance definition that captures this understanding. One possible approach to address this problem is to emulate and extend what was done for the application logic where several notions of behavioural compliance have been studied (e.g.[14, 8]).

Another potential avenue worth considering is the work on monitorability [22, 2] and enforceability [43, 4] that relates the behaviour of the monitor to that specified by the correctness property of interest; the work in [25] investigates these issues for a target actor calculus that is deeply inspired by the Erlang model. In such cases we would need to extend the concept of monitorability and enforceability to adaptability with respect to the local strategy derived from the global specification.

Once we identify and formalise our notions of compliance, we should study their decidability properties, and investigate approaches to check compliance such as type-checking or behavioural equivalence checking (*e.g.,* via testing preorders or bisimulations [20, 3]).

Seamless integration. A key driving principle of our proposed approach is that the recovery logic should be orthogonal to the application logic. This separation of concerns allows the traditional designers to focus on the application logic and just declare the error conditions to be managed by the recovery logic. The dedicated designers of the recovery logic would then use those error conditions and the structure of the choreography of the application logic to specify a recovery strategy. Finally, the application and recovery logic should be integrated via appropriate code instrumentation mechanisms to cater for reliability. The driving principle we will follow is that of minimising the entanglement between the respective models of the application logic and those of the recovery logic. This principled approach with clearly delineated separation of concerns should also manifest itself at the code level of the systems produced, that will, in turn, improve the maintainability of the resulting systems.

4 An instance

We propose a line of research that aims to combine the run-time monitoring and local adaptation of distributed components with the top-down decomposition approach brought about by choreographic development. Our manifesto may thus be distilled as:

Local Runtime Adaptation + Static Choreography Specifications = Choreographed MOP

Our work stems from two existing bodies of work. On the one hand, our investigation is grounded on the Erlang monitoring framework developed and implemented in [10, 11], which showed that these concepts are realisable. On the other hand, the end point of what we want to achieve is driven by the design of a choreographic model for distributed computation with global views and local projections of [34], reviewed in Section 4.1.

4.1 Global and Local Specifications

A key reason that makes choreographies appealing for the modelling, design, and analysis of distributed applications is that they do not envisage centralisation points. Roughly, in a choreographic model one describes how a few distributed components interact in order to coordinate with each other. There is a range of possible interpretations for choreographies [7]; a widely accepted informal description is the one suggested by W3C's [30]:

> [...] a contract containing a global definition of the common ordering conditions and constraints under which messages are exchanged, is produced that describes, from a **global viewpoint** [...] observable behaviour [...]. Each party can then use the **global definition** to build and test solutions that conform to it. The global specification is in turn realised by combination of the resulting **local systems** [...]

According to this description, a **global** and a **local** view are related as in the left-most diagram in Fig. 1 which evokes the following software development methodology. First, an architect designs the global specification and then uses the global specification to derive, via a 'projection' operation, a local specification for the distributed components. Programmers can then use the local specifications to check that the implementation of their components are compliant with the local specification. The keystones of this process are (*i*) that the global specification can be used to guarantee good behaviour of the system abstracting away from low level details (typically assuming synchronous communications), (*ii*) that projection operation can usually be automatised so to (*iii*) produce local specifications at a lower level of abstraction (where communication are asynchronous) while preserving the behaviour of the global specification.

We remark that the relations among views and systems of choreographies are richer than those discussed here. For instance, local views can also be compiled into template code of components and the projection operation may have an "inverse" (cf. [34]). Those aspects are not in scope here.

We choose two specific formalisms for global and local specifications. More precisely, we adapt to our needs the *global graphs* of [34] for global specifications and Erlang actors to express local views of choreographies.

Global specifications. *Global graphs*, originally proposed in [18] and recently generalised in [45, 28], are a convenient specification language for global views of message-passing systems. They yield both a formal framework and a simple visual representation that we review here, adapting notation and definition from [45].

Hereafter we fix two disjoint sets $\mathcal{P}$ and $\mathcal{M}$; the former is a finite set of *participants* (ranged over by A, B, etc.) and $\mathcal{M}$ is the set of *messages* (ranged over by m, x, etc.). To exchange messages and coordinate with each other, participants use asynchronous point-to-point communication via *channels* following the *actor model* [29, 5]. We remark that global graphs abstract away from data; the messages specified in interactions of global graphs have to be thought of as data types rather than values.

The syntax of global graphs is defined by the grammar

$$\mathsf{G} \;::=\; \mathsf{A}{\to}\mathsf{B}\colon \mathsf{m} \quad | \quad \mathsf{G};\mathsf{G}' \quad | \quad \mathsf{G} \mid \mathsf{G}' \quad | \quad \mathsf{G}{+}\mathsf{G}' \quad | \quad *\mathsf{G}@\mathsf{A}$$

A global graph can be a simple interaction $\mathsf{A}{\to}\mathsf{B}\colon \mathsf{m}$ (for which we require $\mathsf{A} \neq \mathsf{B}$), the sequential composition $\mathsf{G};\mathsf{G}'$ of G and G', the parallel composition (for which the participants of G and of G' are disjoint), a nondeterministic choice $\mathsf{G}{+}\mathsf{G}'$ between G and G', or the iteration $*\mathsf{G}@\mathsf{A}$ of G. The syntax captures the

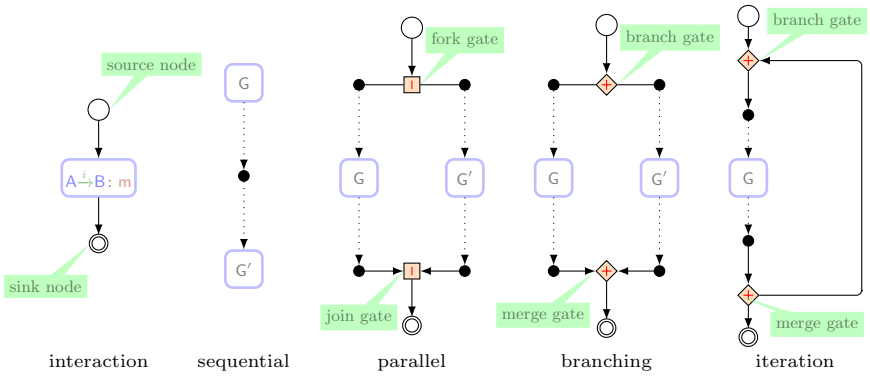


Fig. 2. A visual notation for global graphs

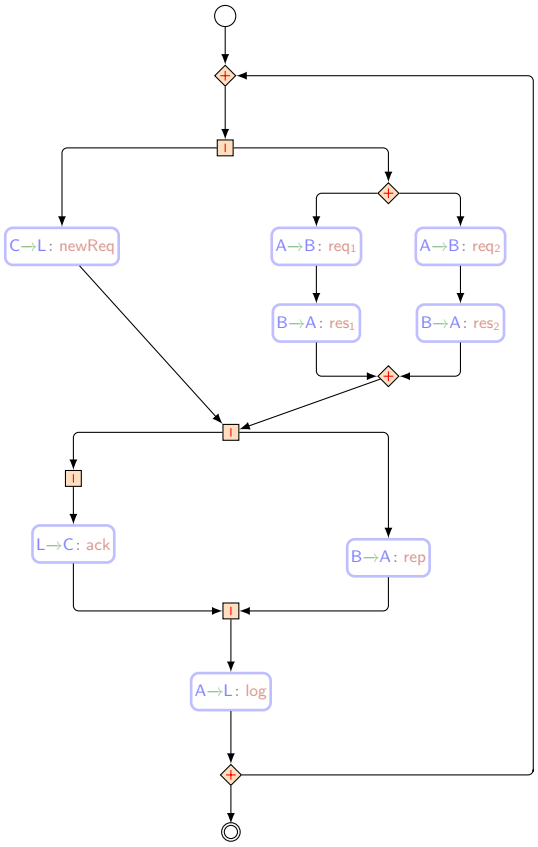
structure of a visual language of distributed workflows illustrated in Fig. 2. Each global graphs G can be represented as a rooted diagram with a single source node and a single sink node respectively represented as $\circ$ and $\circledcirc$. Other nodes are drawn as $\bullet$ and a dotted edge from/to a $\bullet$-node singles out the source/sink nodes the edge connects to. For instance, in the diagram for the sequential composition, the top-most edge identifies the sink node of G and the other edge identifies the source node of G'; intuitively, $\bullet$ is the node of the sequential composition of G and G' obtained by "coalescing" the sink of G with the source of G'. In our diagrams, branches and forks are marked respectively by $\diamond$ and $\square$ nodes; also, to each branch/fork nodes corresponds a "closing" gate merge/join gate.

Example 1. Consider a protocol where iteratively participant C sends a newReq message to a logging service L. In parallel, a C's partner, A, makes either requests of either type req_1 or type req_2 to a service B, which, in turn, replies via two different types of responses, namely res_1 and res_2. Once a request is served, B also sends a report to A, which logs this activity on L. This protocol can be modelled with the graph $\mathsf{G} = *(\mathsf{G}_1 \mid \mathsf{G}'_1);\mathsf{G}_2;\mathsf{G}_3@\mathsf{A}$ where

$$\mathsf{G}_1 = \mathsf{C}\to\mathsf{L}\colon \mathsf{newReq} \qquad \mathsf{G}'_1 = \mathsf{A}\to\mathsf{B}\colon \mathsf{req}_1;\mathsf{B}\to\mathsf{A}\colon \mathsf{res}_1 + \mathsf{A}\to\mathsf{B}\colon \mathsf{req}_2;\mathsf{B}\to\mathsf{A}\colon \mathsf{res}_2$$
$$\mathsf{G}_2 = \mathsf{L}\to\mathsf{C}\colon \mathsf{ack} \mid \mathsf{B}\to\mathsf{A}\colon \mathsf{rep}$$
$$\mathsf{G}_3 = \mathsf{A}\to\mathsf{L}\colon \mathsf{log}$$

The decision to leave or repeat the loop is non-deterministically taken by one of the participants (in this case A) which then communicates to all the others what to do. This will become clearer in Section 6. The diagram in Fig. 3 is the visual counterpart of G. $\diamond$

The (forward) semantics of global graphs can be defined in terms of partial orders of communication events [45, 28]. We do not present this semantics here (the reader is referred to [45, 28]) for space limitations; instead, we give only a brief and informal account through a "token game" similar to the one of Petri



The topmost ◈ gate is the entry point of a loop which simply lets the token to flow. At the first ▯ gate, the token is duplicated, forking the computations along the two threads. In the leftmost thread, the token enables the interaction C→L: newReq; this allows the output event from C (which then waits for the ack message from L) and later the input event of L. The token on the leftmost thread then enables the last interaction L→C: ack. Observe that, after the input of message ack, C can start the next iteration while the other threads may still be completing the current iteration. Concurrently, the token flowing on the rightmost thread reaches another branch gate ◈ which non-deterministically routes the token either on the left or on the right branch. On both branches A and B execute a request-response type of protocol similarly to what C and L run on the leftmost thread. When the token flows through the merge gate at the end of the choice, it enable a last interaction from B to A (which allows B to go the next iteration) and subsequently, the last logging interaction between A and L. Finally, also A and L can repeat the loop.
Note that the body of an iteration is executed at least once.

Fig. 3. The diagram of a global graph and its semantics

nets based on Fig. 3. The token game would start from the source node and flow down along the edges in the diagram as described by the test in Fig. 3.

For the semantics of global graphs to be defined, *well-branchedness* [45, 28] is a key requirement. This is a simple condition guaranteeing that *all* the participants involved in a distributed choice follow a same branch. Well-branchedness requires that each branch in a global graph (*i*) has a unique *active* participant (that is a unique participant taking the decision on which branch to follow) and (*ii*) that any other participant is *passive*, namely that it is either able to ascertain which branch was selected from the messages it receives or it does not play any role in the branching.

Example 2. In the branch of Example 1, A is the active participant while the others are passive; in fact, C and L are not involved in the choice, while B can determine that the left or the right branch was selected depending on which type of request it receives. ◇

Local specifications. We adopt systems of CFSMs [9] as our model of local specifications. A CFSM is a finite-state automaton where transitions represent input or output events from/to other machines. Each machine in the system corresponds to an actor which can send or receive messages to/from other machines. Communications take place on unbound FIFO buffers: for each pair of machines, say A and B, there is a buffer from A to B and one from B to A. Basically, when a machine A is in a state q with a transition to a state q' whose label is an output of message m to B, then m is put in the buffer from A to B and A moves to state q'. Similarly, when B is in a state q with a transition to a state q' whose label is an input of m from B and the m is on the top of the buffer from A to B then B pops m from the buffer and moves to state q'.

Noteworthy, the model of CFSMs is very close to the actor model and CFSMs can be projected from global graphs automatically. Moreover, when the global graph, say G, is *well-formed* then the behaviour of the projected machines faithfully refines the semantics of G [28]. In this paper, we will directly synthesise Erlang code from the global specification, that is we will use Erlang actors to model our local specifications.

5 Global Graphs for Reversibility

We propose a variant of global graphs, dubbed *reversibility-enabling (global) graphs* (REGs for short) that generalises the branching construct to cater for reversibility. We will use REGs to render the recovery model in Section 3.

Example 3. Recall the global graph in Example 1. A possible reversion guard for B could specify that the port required to respond A needs to be available at the time of communication, or that the size of the communication buffer for this port does not exceed a given threshold. At runtime, both conditions may prohibit the respective participants from completing the execution of the specified protocol. By reversing the choice taken (*i.e.* A making requests of either type req_1 or of type req_2), the participants involved can make alternative choices. ◇

The syntax of REGs uses *control points*[1] to univocally identify positions where choices have to be made on how to continue the protocol. Syntactically, control points are written as $_i\mathsf{G}$, where i is a strictly positive integer.

Definition 1 (Reversibility-enabling global graphs). *The set $\mathcal{G}$ of* reversibility-enabling global graphs (REGs) *consists of the terms G derived by the following grammar:*

$$\begin{aligned}\mathsf{G} ::= {} & \mathsf{A}\to\mathsf{B}\colon \mathsf{m} \quad | \quad \mathsf{G};\mathsf{G}' \quad | \quad {}_i(\mathsf{G} \mid \mathsf{G}') \quad | \\ & {}_i\big(\mathsf{G}_1 \ \mathtt{unless}\ \phi_1 \ + \ \mathsf{G}_2 \ \mathtt{unless}\ \phi_2\big) \mid \qquad (1) \\ & {}_i\big(*\mathsf{G}@\mathsf{A}\big) \qquad (2)\end{aligned}$$

that satisfy the following conditions:

[1] Control points can be automatically generated; for simplicity, we explicitly put them in the syntax of REGs.

– *in* $i.(*\mathsf{G}@\mathsf{A})$, A *is the active participant of* G *and*
– *for any two control points* i *and* j *occurring in different positions of a REG it must be the case that the indices are distinct,* $i \neq j$.

In (1)*, the formulas* ϕ_h *(for* $h \in \{1, 2\}$*) are* reversion guards *expressed in terms of boolean expressions.*

In Definition 1, the participant A in (2) decides whether to repeat the body G or exit an iteration. Hereafter, we consider equivalent REGs that differ only in the indices of control points (the indices of control points are, in fact, irrelevant as long as they are unique) and may omit control points when immaterial, e.g. writing $\mathsf{G} \;\mathtt{unless}\; \phi + \mathsf{G}' \;\mathtt{unless}\; \phi'$ instead of $i.(\mathsf{G} \;\mathtt{unless}\; \phi + \mathsf{G}' \;\mathtt{unless}\; \phi')$.

The new branching construct (1) extends the usual branching construct of choreographies to control reversible computations. The semantics of this constructs is rendered by the encoding in Section 6 which realises the following intended behaviour. The execution of $i.(\mathsf{G}_1 \;\mathtt{unless}\; \phi_1 + \mathsf{G}_2 \;\mathtt{unless}\; \phi_2)$ requires first to non-deterministically choose $h \in \{1, 2\}$ and execute the REG G_h. At the end of the execution of G_h then its guard ϕ_h is checked. It the guard is false, then the execution exits the branch and continues executing normally. It the guard is true we may have two sub-cases depending whether the other branch has been already reversed or not. In the first case, then the execution is forced to proceed normally (e.g., there is no alternatives to try), in the second case then the execution of G_h is reversed and the other branch is executed.

Note that, by keeping track of all reversed branches and fully executing the last branch when all the others have been reversed, we can easily generalise to a branching construct $i.(\mathsf{G}_1 \;\mathtt{unless}\; \phi_1 + \cdots + \mathsf{G}_h \;\mathtt{unless}\; \phi_h)$ with $h \geq 2$; for simplicity we just consider $h = 2$ here.

Definition 1 parameterises REGs on the notion of reversion guard. However, our study required us to address crucial design choice on how reversion guards are rendered in a language like Erlang (without a global state). Roughly, reversion guards can be thought of as propositions predicating on the state of the forward execution. A key requirement for a proper projection, however, is that the evaluation of such guards must be "distributable", *i.e.* we want revision guards to be "projectable" from the global view to the components realising the behaviour of the participants. To meet this requirements, we use *local guards*, *i.e.* boolean expression that predicate on the state of a specific participant and assume that a revision guard is a *conjunction of the local guards at each participant*. More concretely, we exploit Erlang's support [1] for accessing the status of a process implementing a participant via system functions such as `process_info` or `system_info`, which return a dictionary with miscellaneous information about a process or a physical node respectively.

Example 4. Consider the following concrete examples of revision guards:

```
queue_len(Threshold, State) ->
   Info = from_list(State),
   {_,Len} = find(message_queue_len, State),
   (Len > Threshold).
```

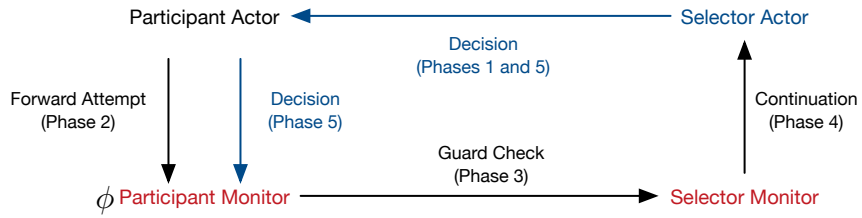


Fig. 4. The instrumentation architecture connecting participant actors, coordinating (selector) actors and their respective monitor actors

```
message_exists(Filter, State) ->
   Info = from_list(State),
   {_,messages} = find(message_queue_len,State),
   Filter(messages).
```

Predicate `queue_len` checks if the size of the mailbox is above a threshold, whereas `message_exists` checks for the presence of a message matching some pattern in a mailbox. Other examples of reversion guards are conditions on PIDs and port identifiers, heap size, or the status of processes (e.g., waiting, running, runnable, suspended). ◇

Our reversible semantics still requires well-branchedness: a REG, say G, is well-branched when the global graph obtained by removing reversion guards from G is well-branched (as defined in Section 4). This guarantees communication soundness in presence of reverse executions.

6 From REGs to Erlang

This section shows how we map REGs into Erlang programs. This mapping corresponds to the definition of *projection* from the global view provided by REGs into Erlang implementations of their local view. Our encoding embraces the principles advocated in [13] and reviewed in Section 3: we strive for a solution yielding a high degree of decoupling between forward and reverse executions. Unsurprisingly, the most challenging aspect concerns how branches are projected. This is done by realising a coordination mechanism which interleaves forward and reversed behaviour, as described in Section 5. In the following, we first describe the architecture of our solution. We then show how forward and reversed executions are rendered in it.

6.1 Architecture

The abstract architecture of our proposal is given in Fig. 4. Each participant of a REG is mapped to a *pair* of Erlang actors, the *participant actor* and the *participant monitor* which liaise with one another in order to realise reversible

distributed choices. The execution of a distributed choice is supported by another pair of (dynamically generated) actors, the *selector actor* which liaises with its corresponding *selector monitor*. The basic idea is that participant and selector actors are in charge of executing the forward logic part of the choice while their respective monitors deal with the reversibility logic.

A key structural invariant of the architecture is that monitors can interact only with their corresponding participant or with the monitors of the selectors currently in execution, as depicted in Figure 4. This organisation is meant to represent the information and control flow of our solution. The coordination protocol required to resolve a distributed choice specified in a REG is made of the following phases:

1. **Inception**: The selector actor (started at a branching point) decides which branch to execute and communicates its decision to the participants involved.
2. **Forward attempt**: Participant actors execute the selected branch accordingly and report their local state at the end of the branch to their participant monitor.
3. **Guards checking**: Participant monitors check their reversion guard and communicate the outcome to the selector monitor.
4. **Continuation**: The selector monitor aggregates the individual outcome of all participant monitors and reports the aggregated result to the selector actor.
5. **Decision**: Based on suggestion forwarded by the selector monitor, the selector actor decides whether to continue forward or reverse the execution and communicates the decision to all participants, which in turn propagate it to their participant monitor.

These phases roughly correspond to the arrows in Figure 4.

6.2 Branching Actors and Monitors

We now describe the behaviour of actors and monitors in a choice, with the help of their automata-like representation in Figure 5. The coordination protocol that we describe here resembles a 2-phase commit protocol where participants report the outcome of local computations to a coordinator that then decides how to continue the execution.

When participant actors (start to) reach a branching point, the inception phase begins. The actor corresponding to the (unique) active participant of the choice spawns the selector actor and waits from the selector message telling which branch to take in the choice; all other participant actors just wait for the selector's decision. The act of spawning the selector arrow by the active participant is represented in Fig. 5 via the gray arrow and the cloud in the automaton of the participant actor. Subsequently, all the actor participants involved in a branch will wait from the selector to instruct them with the branch (either left or right) to take—these are the yellow arrows in the automaton of Figure 5. Upon the receipt of such a message, participant actors first forward

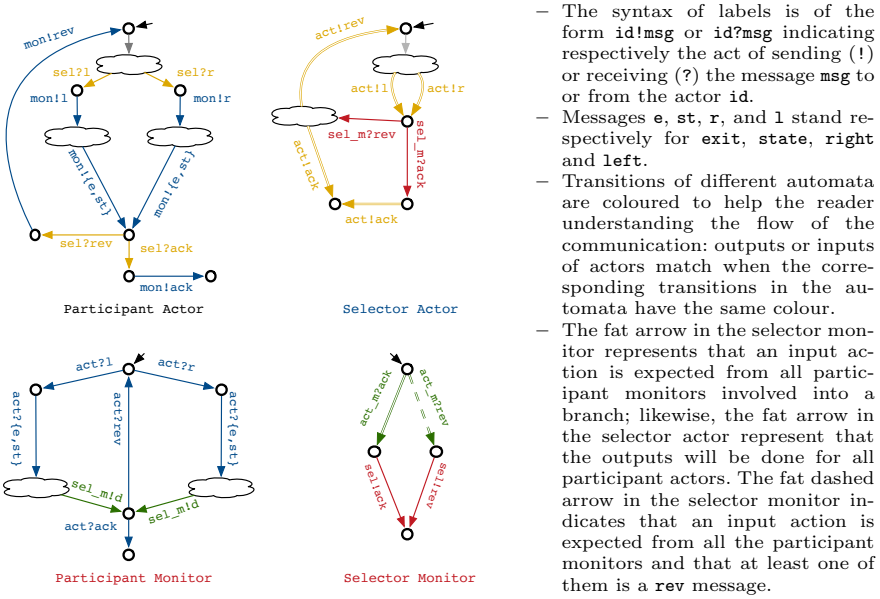


Fig. 5. Automata-like description of actors and monitors for the projection of branches

this message to their monitor and then enter the second phase executing the branch—represented by the cloud in the automaton. Unless the chosen branch diverges, the third phase starts when participant actors finish the branch (possibly at different times) and they signal to their monitor that they are ready to exit the choice. This is signalled by the `exit` message which also carries the local state of execution (described in Section 5). At this point, participant actors take part only in the last phase: they receive from the selector either an `ack` message (confirming that the choice has been resolved) or a `rev` message to reverse the execution. In either case, they propagate the message to their monitor and either "commit" the branch or return to the state that waits for the message dictating the next branch to take. Participant actors behave uniformly but for the active one, which has the additional task of spawning the selector at the very beginning (for non-active participants the grey transition is an internal step not affecting communications).

Each participant monitor waits for the message carrying the local state that its participant actor sends at the end of the second phase in the `exit` message. The state is used to check whether the reversion guard of the branch, say ϕ, holds or not. If ϕ holds for the local state of the participant actor, then the participant monitor sends the selector monitor a request to *reverse* the branch (message `rev`). Otherwise the monitor sends a message to commit the choice (message `exit`). In Figure 5 this is represented by the label `sel_m!d`, where `d` stands for decision and `sel_m` binds to the unique identifier of the selection monitor

implemented as an actor. After this, the monitor waits from its participant actor for the `rev` or the `ack` message sent in the last phase: if `rev` is received the monitor returns to its initial state and leaves the branch otherwise.

The selector actor spawned in the inception phase starts by spawning a selector monitor and then deciding which branch to take initially—represented in Figure 5 by the grey transition and the cloud in the automaton of the selector. After communicating its decision to all participant actors, the selector waits for the request of its monitor and starts phase five of Section 6.1 by deciding whether to reverse the branch or not. The decision process is as follows: if the selector receives an `ack` message then the branch is committed and the selector monitor terminates. Otherwise, the selector participants receive a `rev` message to reverse the branch. If there are branches that have not been taken yet, then the last executed branch is marked as "tried", a branch that has not been attempted yet is selected, and a `rev` message is sent to all participant actors. Otherwise, the decision to commit the branch is taken and the `ack` message is sent to all participant actors. In the former case, the selector returns to its initial state, and terminates otherwise.

The selector monitor participates to the fourth phase. It first gathers all the outcomes from the guard-checking phase from *all* the participant monitors involved into the choice. Recall that a `rev` message is received from any participant monitor whose revision guard becomes true, while an `ack` message is received from any participant monitor whose revision guard does not hold. Then, the selector monitor computes an outcome to be sent to the selector actor: if all received messages are `ack` then an `ack` message is sent to the selector actor, otherwise the monitor sends a `rev` message to the selector actor. In both cases, the selector monitor terminates; a new selector monitor is spawned by the selector actor if the branch is actually reversed.

Iteration is a simplification of a distributed choice: we just generate a selector for an iteration but not its monitor. The reason for not having a monitor for the iterator selector is due to the fact that there is no reversible semantics to be implemented for the iteration. This does not imply that within the body of an iteration a reversible step can not be taken (e.g. there can be an inner choice), but just that iterations are not points at which the computation can be reversed. The selector (instantiated by the active participant of the iteration, similarly to choices) just decides whether to iterate or exit the loop. A participant actor within a loop, after completing an iteration, awaits the decision from the selector actor and continues accordingly.

6.3 Compiling to Erlang

The code generated for the projections from REGs to Erlang is discussed below. We focus on the compiled code for the branches constructs, since the compilation of the other constructs is standard and therefore omitted. Our discussion uses auxiliary functions for which the code is not reported.

```
1  act_A_cp() ->
2    %Pid =  list_to_atom("sel_act_"
3    %++ integer_to_list(cp)),
4    %register("Pid,
5    %          spawn(sel_act,[cp])),
6    receive
7      {cp,left} ->
8          mon_A ! {cp, left}
9            %CODE OF LEFT BRANCH
10         ;
11     {cp,right} ->
12         mon_A ! {cp, right}
13           %CODE OF RIGHT BRANCH
14   end,
15   mon_A!{cp, exit, process_info(self())},
16   receive
17     {cp,ack} -> mon_A ! {cp, ack};
18     {cp,rev} ->
19       mon_A ! {cp, rev},
20       act_A_cp()
21   end.

22 mon_A_cp() ->
23 receive
24   {cp, left} ->
25       %CODE FOR LEFT BRANCH MONITOR%
26     receive{cp, exit, Info} ->
27       G = check_guard(Left_guard, Info)
28     end;
29   {cp, right} ->
30       %CODE FOR RIGHT BRANCH MONITOR%
31     receive{cp, exit, Info} ->
32       G = check_guard(Right_guard, Info)
33     end
34 end,
35 Sel_m = get_selector_monitor(cp),
36 case G of
37   true -> Sel_m ! {cp, rev};
38   _    -> Sel_m ! {cp, ack}
39 end,
40 receive
41   {cp, rev} -> mon_A_cp();
42   {cp, ack} -> ok
43 end.

44 sel_act(Attempt,CP) ->
45   Pid = list_to_atom("sel_mon_"
46       ++ integer_to_list(CP)),
47   register(Pid,
48      spawn(sel_mon, [CP, self()])),
49   Sel =
50     case Attempt of
51       [] -> getBranch();
52       [left] -> right;
53       [right] -> left;
54       _ -> throw("panic....") %this case never happens
55     end,
56     P = participants(CP),
57     foreach(fun(X) -> X!{CP, Sel} end, P),
58     receive {CP,Outcome} ->
59       Decision =
60         case {Outcome,Attempt} of
61          {ack,_} -> ack;
62          {rev,[]} -> rev;
63          {_,_} -> ack
64        end
65     end,
66    foreach(fun(X) -> X!{CP, Decision} end, P),
67      case Decision of
68        rev -> sel_act(Attempt ++ [Sel], CP);
69        _   -> end_branch
70      end.

71 sel_mon(CP, SelPid)->
72    MP = participants(CP),
73    MsgList = lists:map(fun(_) ->
74        receive {CP,M} -> M end end, MP),
75    Msg =
76      case lists:member(rev, MsgList) of
77        true -> rev;
78        _    -> ack
79      end,
80    SelPid ! {CP, Msg}.
```

The code for the participant actor (lines 1-21) is parametrised with respect to `cp`, the value of the control point[2] univocally identifying the point of branch in the REG. The commented lines 2-5 are generated only for the code of the active participant which spawns the selector actor of the branch `CP`. Note that the process is registered under a unique name `sel_act_cp` (which is an atom). This snippet is actually a template which would be filled up with the code generated for the participant communications respectively on the left and on the right branches (*i.e.* the commented lines 9 and 13).

The Erlang process spawned by a participant actor implementing the selector actor executes the function on lines 44-70. This function takes two parameters: the `Attempt` representing the branches chosen so far and the control point `CP` identifying the choice. The former parameter is a list of atoms `left` and `right`; note that the empty list is passed initially when the process is spawned and that (in our case) the size of this list should never exceed 1. As discussed above, the selector chooses a branch (lines 49-55) and communicates its decision to the participants of the branch (lines 56-57, where `participants` is computed at compile time, from the global graph script, and returns the participants of a branch given its control point). Finally, the selector enters the fourth phase of

[2] Note that the value `cp` is statically determined by the compiler.

Section 6.1, waiting for the message from its monitor, and decides accordingly how to continue the execution of the choreographed choice.

As in the case of the participant actor, the snippet of the participant monitor (lines 22-43) does not make explicit the code for the monitoring of the left and right branches (commented lines 25 and 30). The auxiliary function `check_guard` returns the evaluation of the guard for the state provided by the participant (lines 26-28 and 31-33). The function `get_selector_monitor` retrieves the PID of the selector monitor from the control point value `CP`.

The selector monitor, spawned by the selector process, is registered with the name `sel_mon_cp` (lines 45-48) where `cp` is the second actual `CP` when invoking `sel_act`. Note that the invocation to `get_selector_monitor` on line 35 returns the atom `sel_mon_cp`. The snippet for the selector monitor uses the auxiliary function `participants` returning the list of participant actors involved in the branch `cp`. The outcome `Msg` is computed on lines 73-79 and sent to the selector on line 80. The selector monitor awaits a message from all the participant monitors involved in the branch (lines 73-74), and then it decides the message to communicate to the selector actor. If at least one of the messages received is `rev`, then the final message is `rev`, otherwise the final message is `ack`.

7 Conclusions

We have presented a methodology to automate the process of adding recovery strategies to message passing systems specified via a global protocol. In particular, our model abstracts from (1) the definition of formal behavioural models encompassing failures, (2) the specification of the relevant properties of adaptation and recovery strategy, (3) the automatic generation of monitoring, recovery, and adaptation logic in target languages of interest.

In line with the principles advocated by our model, we then have presented a minimally-intrusive extension to global graph choreographies [28] for expressing reversible computation. We showed how these descriptions could be realised into executable actor-based Erlang programs that compartmentalise the reversion logic as Erlang monitors, minimally tainting the application logic.

Related Work. The closest work to ours is [33, 40, 19]. In [33] a reversible semantics for a subset of Erlang is given. The goal of [33] is a debugger based on a fully reversible semantics. To achieve this, they modify the Erlang semantics in order to keep track of the computational history and build an ad-hoc interpreter for it. Our goal is different since we focus on *controlled reversibility* [31]. Our framework automates the derivation of rollback points (namely the exact point at which the execution has to revert) from the recovery logic. Also, the use of monitors avoids any changes to Erlang's run-time support. Choreographies are used in [40] to devise an algorithm that optimises Erlang's recovery policies. More precisely, global views specify dependencies from which a global recovery tables are derived. Such tables tell which are the safe rollback points. The framework then exploits the supervision mechanism of Erlang to pair participants with a monitor. In case of failure, the monitor restarts the actor to

a consistent rollback point. One could combine our approach with the recovery mechanism of [40] so as to generalise our reversible semantics to harness fault tolerance. This is not a trivial task, because the fault-tolerance mechanism of [40] needs to follow a specific protocol, making it unclear whether participants can be automatically derived. In [19] actors are extended with checkpoints primitives, which the programmer has to specify in order to rollback the execution. In order to reach globally-consistent checkpoints severe conditions have to be met. Thanks to the correctness-by-design principle induced by global views, our approach automatically deals with checkpoints, relieving this burden from the programmer.

Other works [41, 37, 38] have investigated the use of monitors to steer reversibility in concurrent systems. In [41] a monitored reversible process algebra is presented where each agent is paired with a monitor. But, unlike our approach, the monitor tells the agent what to do both in the forward and in the reverse way. In [37, 38] the authors investigate the use of monitors to steer reversibility in message oriented systems. Here monitors are used as *memories* storing information about the forward execution of the monitored participants, and this information is then used to reconstruct previous states. As in our approach, in [38] participants and their monitors are derived from a global specification as well. We diverge from [37, 38] in several aspects. Firstly, our monitors do not store any information about the forward computation. Secondly, all the monitors coordinate amongst each other to decide whether to revert a particular computation or not. The coordination mechanism of our monitors is automatically derived. Moreover in our approach reversibility is triggered at run-time when certain conditions (specified at design-time in the recovery logic) are met.

Conclusions. We have presented a method to automatically derive reversible computation as Erlang actors. A key aspect of our approach is the ability to express, from a global point of view, *when* a reverse distributed computation has to take place and not *how*. Starting from a global specification of the system, branches can be decorated with conditions that at run-time will enable the coordinated undoing of a certain branch. Another novelty of our approach is the use of monitors to enact reversibility. We leave as future work the measurement of the overhead of our approach on the normal forward semantics of the actors, in terms of messages and memory consumption. Another research direction is to integrate our recovery logic with existing monitoring frameworks for Erlang. In [10, 11], Cassar *et al.* developed the monitoring tool adaptEr[3] for synthesising adaptation monitors for actor systems developed in Erlang. Specifications in adaptEr are defined using a version of Safe Hennessy Milner Logic with recursion (sHML) that is extended with data binding, if statements for inspecting data, adaptations and synchronisation actions. We will investigate the idea of extending this logic with reversibility capabilities, and then to synthesise monitors directly from this logic formulae.

[3] The tool adaptEr is open-source and downloadable from https://bitbucket.org/casian/adapter.

Several works have shown that reversible debuggers can be built on top of reversible semantics [17, 32, 26]. In line with these works, our ultimate goal would also be to build a (reversible) debugger for Erlang systems. One idea could be to integrate our automatic synthesis of reversible code with commercial systems which are able to monitor and aggregate several information (events) of a message passing system. One of such candidate is WombatAOM[4]. Such an integration will allow our reversion guards to predicate on real runtime information. On a different topic, REGs could also be used to enhance *Continuous Integrations* [36] scenarios, by proposing a formalism to express workflows imbued with reversible behaviour to support automatic tests generation and flakiness detection.

References

1. Erlang run-time system application, reference manual version 9.2.
2. L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. Adventures in monitorability: From branching to linear time and back again. *Proceedings of the ACM on Programming Languages*, 3(POPL):52:1–52:29, 2019.
3. L. Aceto, A. Achilleos, A. Francalanza, A. Ingólfsdóttir, and K. Lehtinen. Testing equivalence vs. runtime monitoring. 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 28–44. Springer, 2019.
4. L. Aceto, I. Cassar, A. Francalanza, and A. Ingólfsdóttir. On runtime enforcement via suppressions. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
5. G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.
6. D. P. Attard and A. Francalanza. A monitoring tool for a branching-time logic. In *RV*, volume 10012 of *LNCS*, pages 473–481. Springer, 2016.
7. D. Basile, P. Degano, G.-L. Ferrari, and E. Tuosto. Relating two automata-based models of orchestration and choreography. *JLAMP*, 85(3):425 – 446, 2016.
8. G. Bernardi and M. Hennessy. Mutually testing processes. *LMCS*, 11(2), 2015.
9. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
10. I. Cassar and A. Francalanza. Runtime adaptation for actor systems. In E. Bartocci and R. Majumdar, editors, *RV2015*, volume 9333 of *LNCS*, pages 38–54. Springer, 2015.
11. I. Cassar and A. Francalanza. On implementing a monitor-oriented programming framework for actor systems. In *IFM 2016*, volume 9681 of *LNCS*, pages 176–192. Springer, 2016.
12. I. Cassar, A. Francalanza, D. P. Attard, L. Aceto, and A. Ingólfsdóttir. A suite of monitoring tools for Erlang. In G. Reger and K. Havelund, editors, *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools*, volume 3 of *Kalpa Publications in Computing*, pages 41–47. EasyChair, 2017.

[4] https://www.erlang-solutions.com/products/wombatoam.html

13. I. Cassar, A. Francalanza, C. A. Mezzina, and E. Tuosto. Reliability and fault-tolerance by choreographic design. In *PrePost@iFM*, volume 254 of *EPTCS*, 2017.
14. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. *ACM Trans. Program. Lang. Syst.*, 31(5), 2009.
15. F. Cesarini and S. J. Thompson. Erlang behaviours: Programming with process design patterns. In *CEFP 2009, Budapest, Hungary*, volume 6299 of *LNCS*, pages 19–41. Springer, 2009.
16. F. Chen, D. Jin, P. Meredith, and G. Roşu. Monitoring oriented programming - a project overview. In *Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS'09)*, pages 72–77. ACM, 2009.
17. F. de Vries and J. A. Pérez. Reversible session-based concurrency in Haskell. In M. H. Palka and M. O. Myreen, editors, *Trends in Functional Programming - 19th International Symposium, TFP 2018, Gothenburg, Sweden, June 11-13, 2018, Revised Selected Papers*, volume 11457 of *LNCS*, pages 20–45. Springer, 2019.
18. P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP 2012*, 2012.
19. J. Field and C. A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL 2005*. ACM, 2005.
20. A. Francalanza. A Theory of Monitors - (Extended Abstract). In *FoSSaCS*, volume 9634 of *LNCS*, pages 145–161. Springer, 2016.
21. A. Francalanza. Consistently-Detecting Monitors. In *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8*, volume 85 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.
22. A. Francalanza, L. Aceto, and A. Ingolfsdottir. Monitorability for the Hennessy–Milner logic with recursion. *Formal Methods in System Design*, pages 1–30, 2017.
23. A. Francalanza, C. A. Mezzina, and E. Tuosto. Reversible choreographies via monitoring in Erlang. In S. Bonomi and E. Rivière, editors, *Distributed Applications and Interoperable Systems DAIS 2018*, volume 10853 of *LNCS*, pages 75–92. Springer, 2018.
24. A. Francalanza, J. A. Pérez, and C. Sánchez. Runtime verification for decentralised and distributed systems. In *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 176–210. Springer, 2018.
25. A. Francalanza and A. Seychell. Synthesising Correct concurrent Runtime Monitors. *Formal Methods in System Design (FMSD)*, 46(3):226–261, 2015.
26. E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In S. Gnesi and A. Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014*, volume 8411 of *LNCS*, pages 370–384. Springer, 2014.
27. J. Gray. Why do computers stop and what can be done about it? In *SRDS*. IEEE, 1986.
28. R. Guanciale and E. Tuosto. An abstract semantics of the global view of choreographies. In *ICE 2016, Heraklion, Greece*, pages 67–82, 2016.
29. C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*. Morgan Kaufmann Publishers Inc., 1973.
30. N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217, 2004.
31. I. Lanese, C. A. Mezzina, and J.-B. Stefani. Controlled reversibility and compensations. In *RC 2012. Revised Papers*, volume 7581 of *LNCS*. Springer, 2012.

32. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. Cauder: A causal-consistent reversible debugger for Erlang. In J. P. Gallagher and M. Sulzmann, editors, *Functional and Logic Programming - 14th International Symposium, FLOPS*, volume 10818 of *LNCS*, pages 247–263. Springer, 2018.
33. I. Lanese, N. Nishida, A. Palacios, and G. Vidal. A theory of reversibility for Erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018.
34. J. Lange, E. Tuosto, and N. Yoshida. From Communicating Machines to Graphical Choreographies. In *POPL*, pages 221–232, 2015.
35. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.
36. M. Meyer. Continuous integration and its tools. *IEEE Software*, 31(3):14–16, 2014.
37. C. A. Mezzina and J. A. Pérez. Causally consistent reversible choreographies: a monitors-as-memories approach. In *PPDP*, 2017.
38. C. A. Mezzina and J. A. Pérez. Reversibility in session-based concurrency: A fresh look. *J. Log. Algebr. Meth. Program.*, 90:2–30, 2017.
39. C. A. Mezzina and E. Tuosto. Choreographies for automatic recovery. *CoRR*, abs/1705.09525, 2017.
40. R. Neykova and N. Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*. ACM, 2017.
41. I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *Reversible Computation, 4th International Workshop, RC 2012, Revised Papers*, 2012.
42. P. Rook. *Software Reliability Handbook*. Elsevier Science Inc., New York, NY, USA, 1990.
43. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
44. D. Thomas. *Programming Elixir: Functional, Concurrent, Pragmatic, Fun*. Pragmatic Bookshelf, 1st edition, 2014.
45. E. Tuosto and R. Guanciale. Semantics of global view of choreographies. *J. Log. Algebr. Meth. Program.*, 95:17 – 40, 2018.
46. D. Wyatt. *Akka Concurrency*. Artima Incorporation, USA, 2013.