Reversible Choreographies via Monitoring in Erlang

Adrian Francalanza, Claudio Antares Mezzina, Emilio Tuosto

Reversible Choreographies via Monitoring in Erlang*

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

[1] University of Malta, Malta adrian.francalanza@um.edu.mt
[2] IMT Advanced Studies Lucca, Italy claudio.mezzina@imtlucca.it
[3] University of Leicester, UK emilio@le.ac.uk

Abstract. We render a model advocating an extension of choreographies to describe *reverse computation* via *monitoring*. More precisely, our extension imbues the communication behaviour of multi-party protocols with *minimal decorations* specifying the conditions triggering monitor adaptations. We show how, from these extended global descriptions, one can (i) synthesise *actors* implementing the normal local behaviour of the system prescribed by the global graph, but also (ii) synthesise *monitors* that are able to coordinate a distributed rollback when certain conditions (denoting abnormal behaviour) are met.

1 Introduction

Runtime Monitoring [17, 18] (or Monitor Oriented Programming [10, 25, 7]) is a code structuring principle whereby ancillary system functionality (dealing with aspects such as security and reliability) is separated from the core functionality of a system and compartmentalised into separate code units called *monitors*. These monitors are occasionally assigned their own thread of control and operate by observing the execution of the core system and reacting to it: typical monitor functionality includes aggregating system information, comparing the execution against some correctness specification, or attempting to modify the execution of the observed systems via filtering, adaptation or enforcement procedures. Monitoring complements traditional verification techniques such as model checking and testing [3, 4, 6, 13, 23] because it allows verification checks to be offloaded to a post-deployment phase: these checks are typically either too expensive to perform statically or else intrinsically dependent on (missing) run-time information. Experience has also shown that often, computation misbehaviour still arises even after the software has undergone rigorous scrutiny prior to deployment. In such cases, monitors provide a natural mechanism to mitigate this misbehaviour.

The goal of this work is to show that monitoring can be used to attain *reversible computation* in models using asynchronous message-passing, such as those found in distributed computing and actor-based languages. Reversible computing [30] has been show to be a suitable abstraction for a variety of application domains from software debugging, to transactions, to fault tolerant schemes [29, 19, 14]. Mechanisms for reversible computing can also be useful to describe and execute recovery strategies in distributed setting, where it is hard to anticipate all the conditions under which parts of computation is carried out: the ability to reverse computation would allow a system to

* Research partly supported by the EU COST Action IC1405.

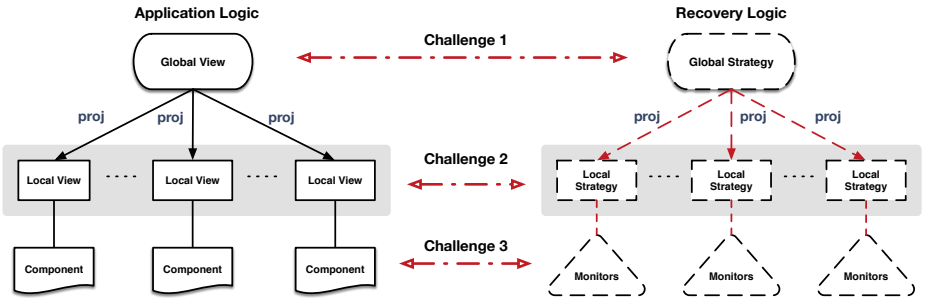


Fig. 1. A model for specifying monitors (reproduced from [8])

backtrack certain execution steps that have been invalidated by the violation of certain conditions, and subsequently execute alternative commands instead.

In spite of its utility, the programming of recovery strategies for asynchronous communicating software can be hard and error-prone. To this end, mechanisms for dealing with reversed execution in asynchronously communicating computation have been introduced. For instance, in [16] an approach based on checkpoints has been developed to cope with transactional behaviour in actor-based systems. But despite providing these convenient abstractions for reversibility, this approach still leaves the programmer with the burden of specifying checkpoints at judicious points within her program. In another work [29], a reversible Erlang dialect has been proposed, maintaining the fundamental features of Erlang while automating the mechanism associated with reversibility. In spite of its advantages, the framework requires the development of a new run-time support (and the adoption of a special-purpose Erlang VM) to handle reversibility.

In this paper, we explore a third approach to tackle this problem. We combine "correctness-by-design" of choreographies, checkpoint-based mechanisms, and runtime monitoring to attain disciplined interweaving of forward and reversed execution. Unlike existing approaches, we fully exploit the benefit of choreographies to guarantee communication soundness by construction. For instance, our checkpoints are automatically derived from global views of choreographies unlike [16]. Moreover, we use runtime monitors to handle reversed executions while avoiding modifications to the standard runtime setup of Erlang (as is done in [29]). This, in turn, facilitates the adoption and portability of our approach.

Technically, we concretise a proof-of-concept realisation of the methodology presented in [8] (see Fig. 1), for designing and implementing monitors for message-passing software. The main ingredient of this methodology is the use of *choreographies* for distributed applications and, in particular, the so-called *top-down* approach illustrated in the left part of Fig. 1. The starting point of our approach are the global graph models called *global views*, which are algorithmically "projected" on the *local views* (one for each "participant" of a choreography). Global views can, on the one hand, be checked for errors at an early stage of the design process and, on the other hand, be automatically *projected* on distributed components interacting via message-passing, to the participants in the local view in *top-down* fashion (see Fig. 1, left). The software components

implementing each participant can, in turn, be checked against their corresponding local view; this guarantees communication soundness whenever the global view satisfies some conditions (*e.g.*, see [22]). The right part of Fig. 1 mirrors the top-down approach of choreographies for the realisation of the *recovery logic*, *i.e.*, activities that a distributed system should carry out to handle undesired states of the computation reached at runtime. Concretely, starting from a global description of the recovery logic, *local strategies* can be derived and rendered as dedicated monitors for every participant. The realisation of such a scheme poses various research questions:

Q1. What global models are suitable to specify the recovery logic?
Q2. How should components and monitors smoothly interact with each other?
Q3. What are the properties should the recovery logic have to facilitate such a scheme?

The challenges illustrated in Fig. 1 correspond to the above questions (*e.g.*, Challenge 1 relates to Q1 and so forth). These challenges should also observe a separation-of-concerns principle espoused by monitor-oriented programming, namely that of *decoupling the application logic from the recovery logic* (as much as possible) [8].

Contributions. This paper describes a proof-of-concept solution for Q1 and Q2 in the setting of Erlang programs. More precisely, (*i*) we propose *reversibility-enabling global graphs* (REGs for short) as a suitable model for the global view of the recovery logic and intertwine this general specification language with Erlang's support for monitoring. Also, (*ii*) we show how to project REGs into Erlang monitors that steer the execution of the system according to some conditions. A basic feature of REGs is the possibility of specifying conditions allowing distributed components to execute distributed choices more flexibly. Specifically, the designer can specify conditions in the global views, dubbed *reversion guards* on distributed choices that are *orthogonal* to the application logic and depend on the run-time state of the computation. In this way, branches of distributed choices may be reversed when their reversion guards flag an undesired state of the computation. Alternatively, these conditions may be easily ignored during projection if desired or updated, without altering the application logic produced. We illustrate this with the following example that will help us throughout the paper.

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. 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 of these 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. ◇

A definition of global graphs enabling reversible computations that is able to handle the aforementioned case would contribute towards answering Q1. Moreover, an automated

projection of these graphs in terms of actors and monitors would start to address Q2. Question Q3 above requires some theoretical results not in scope here. However, this paper already sheds some light on possible desirable properties such as the requirement for "distributability" (over the respective participants) of the recovery logic.

2 Background

We begin by overviewing the preliminaries relating to Global Graph descriptions and the target actor model.

2.1 Global Specifications

Global graphs, originally proposed in [12] and recently generalised in [32, 20], are a convenient specification language for the 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 [32].

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 *message* (ranged over by m, x, etc.). To exchange messages and coordinate with each other, participants use asynchronous point-to-point communication via *channels*. Basically, we adopt the *actor model* [21, 1]. 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

$$\mathsf{G} \;::=\; \mathsf{A}\to\mathsf{B}\colon \mathsf{m} \;\;\big|\;\; \mathsf{G};\mathsf{G}' \;\;\big|\;\; \mathsf{G}\,|\,\mathsf{G}' \;\;\big|\;\; \mathsf{G}+\mathsf{G}' \;\;\big|\;\; \mathsf{repeat}\,\{\mathsf{G}\}$$

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 $\mathsf{repeat}\,\{\mathsf{G}\}$ of G.

An example of global graph is given below.

Example 2. The example discussed in Example 1 of Section 1 can be modelled with the graph $\mathsf{G} = \mathsf{repeat}\,\{(\mathsf{G}_1\,|\,\mathsf{G}'_1);\mathsf{G}_2;\mathsf{G}_3\}$ where

$$\begin{aligned}
\mathsf{G}_1 &= \mathsf{C}\to\mathsf{L}\colon \mathsf{newReq} & \mathsf{G}'_1 &= \mathsf{A}\to\mathsf{B}\colon \mathsf{req}_1;\mathsf{B}\to\mathsf{A}\colon \mathsf{res}_1\\
\mathsf{G}_2 &= \mathsf{L}\to\mathsf{C}\colon \mathsf{ack}\,|\,\mathsf{B}\to\mathsf{A}\colon \mathsf{rep} & &\quad +\\
\mathsf{G}_3 &= \mathsf{A}\to\mathsf{L}\colon \mathsf{log} & &\mathsf{A}\to\mathsf{B}\colon \mathsf{req}_2;\mathsf{B}\to\mathsf{A}\colon \mathsf{res}_2
\end{aligned}$$

The decision to leave or repeat the loop is non-deterministically taken by one of the participants (which one is immaterial) which then communicates to all the others what to do. This will become clearer in Section 4. ◇

The syntax captures the 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 ∘ and ◎. Other

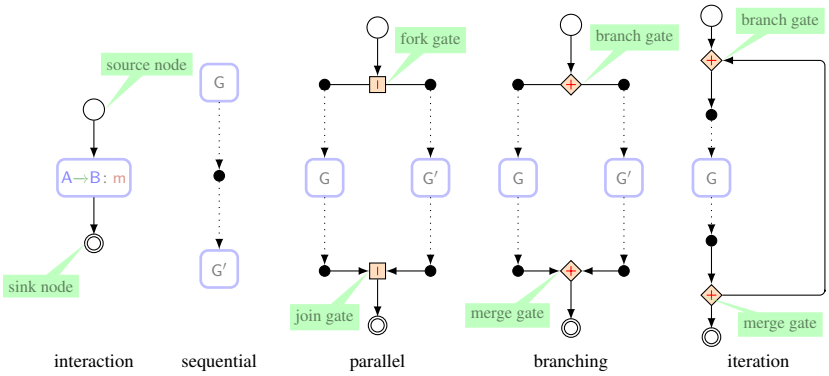


Fig. 2. A visual notation for global graphs

nodes are drawn as • and a dotted edge from/to a •-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 sink node of G and the other edge identifies the source node of G′; intuitively, • 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 ◈ and ▯ nodes; also, to each branch/fork nodes corresponds a "closing" gate merge/join gate.

The (forward) semantics of global graphs can be defined in terms of partial orders of communication events [32, 20]. We do not present this semantics here (the reader is referred to [32, 20]) for space limitations; instead, we give only a brief and informal account based on an example through a "token game" similar to the one of Petri nets.

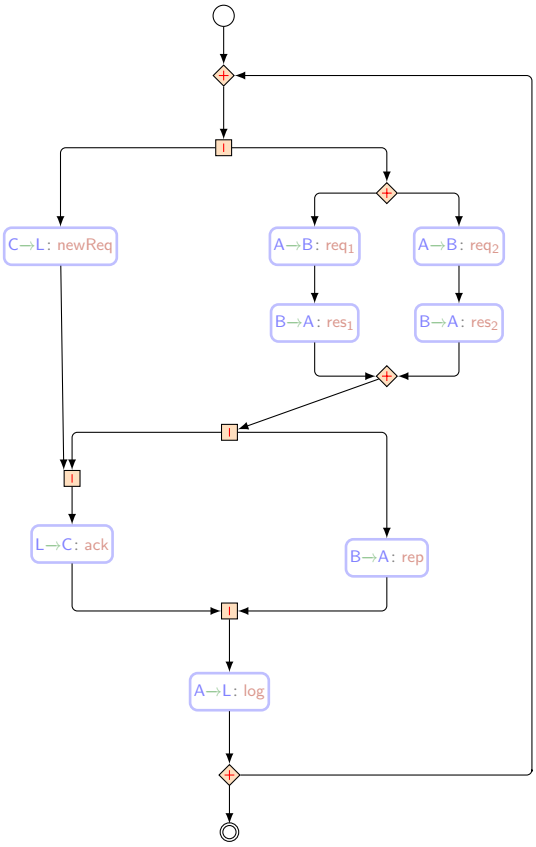
Example 3. The diagram in Fig. 3 is the visual counterpart of G in Example 2. The token game semantics in the example of Fig. 3 would start from the source node and flow down along the edges in the diagram as described by the test in Fig. 3. ◇

For the semantics of global graphs to be defined, *well-branchedness* [32, 20] is a key requirement. This is a condition guaranteeing that *all* the participants involved in a distributed choice follow a same branch. Well-branchedness is quite simple and 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 4. In the branch of Example 2, 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. ◇

2.2 Erlang Model

Erlang [2, 9] is a general-purpose, industry-strength concurrent programming language. Actors—implemented as lightweight processes—constitute its concurrent units of decomposition: (in principle) actors do not share any mutable memory but rather interact



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 of the loop 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

with one another via asynchronous messages, changing their internal state in response to the messages received. Every actors is uniquely identified via a process ID (PID); it owns a message queue, called a mailbox, to which messages are sent in a non-blocking fashion. Messages may be sent to an actor's mailbox only if its PID is known, and, once received, these messages can be selectively (and exclusively) consumed by the recipient actor using pattern matching. Actors may spawn other actors dynamically (at run-time): the PID of a newly spawned actor is originally known only by the spawning actor, but this can then be communicated to other actors via messaging.

Concurrent Erlang actors are typically organised as supervision tiers. Using the process-linking and exit-trapping mechanisms [2, 9], an actor (referred to as a supervisor) may be notified via a message that a linked actor has terminated abnormally (*i.e.*, crashed), which allows it to take remedial action (*e.g.*, avoid waiting indefinitely for a message, or spawn a replacement actor). Erlang supervision hierarchies admit a form of monitor-oriented programming [7], whereby the recovery logic is teased apart from

the application logic, so as to keep the latter as clear as possible; the recovery logic can instead be encapsulated within the supervision structure encasing the application.

Finally, our mapping from REGs to Erlang programs heavily uses *atoms*, that is literal constants which do not carry any value but can be used as a value; Erlang atoms corresponds to values of some unit types in typed languages.

3 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 described in Section 1. The syntax of REGs uses *control points*[4] to univocally identify positions where choices have to be made on how to continue the protocol. Syntactically, control points are written as $i \cdot \mathsf{A}$, where i is a strictly positive integers and $\mathsf{A} \in \mathcal{P}$ is the participant responsible for taking the decision.

Definition 1 (Reversibility-enabling global graphs). *The set $\mathcal{G}$ of* reversibility-enabling global graphs (REGs) *consists of the terms* G *derived by the grammar obtained by replacing the last two productions of the grammar in Section 2.1 with*

$$\mathsf{G} ::= \ldots \mid \mathit{sel}\ i \cdot \mathsf{A}\left\{\mathsf{G}_1\ \mathit{unless}\ \phi_1 + \mathsf{G}_2\ \mathit{unless}\ \phi_2\right\} \quad (1)$$

$$\mid \mathit{repeat}\ i \cdot \mathsf{A}\{\mathsf{G}\} \quad (2)$$

that satisfy the following conditions:

- *in $i \cdot \mathsf{A}\ \mathsf{G}$, A is the active participant of G and*
- *for any two control points $i \cdot \mathsf{A}$ and $j \cdot \mathsf{B}$ 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 (1) non-deterministically decides which branch to follow; in (2) it 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 $\mathsf{G}\ \mathsf{unless}\ \phi + \mathsf{G}'\ \mathsf{unless}\ \phi'$ instead of $\mathsf{sel}\ i \cdot \mathsf{A}\left\{\mathsf{G}\ \mathsf{unless}\ \phi + \mathsf{G}'\ \mathsf{unless}\ \phi'\right\}$.

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 4 which realises the following intended behaviour. To execute $\mathsf{sel}\ i \cdot \mathsf{A}\left\{\mathsf{G}_1\ \mathsf{unless}\ \phi_1 + \mathsf{G}_2\ \mathsf{unless}\ \phi_2\right\}$ we first non-deterministically choose branch $h \in \{1,2\}$ and execute the REG G_h. If the guard ϕ_h is false once the execution of G_h terminates then the execution stops (*i.e.*, it executes as normal); otherwise, if the other branch has not been tried yet, the execution of G_h is reversed and the other branch

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

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 $\mathsf{sel}\ \mathsf{i} \cdot \mathsf{A}\left\{\mathsf{G}_1\ \mathsf{unless}\ \phi_1 + \cdots + \mathsf{G}_h\ \mathsf{unless}\ \phi_h\right\}$ 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 and resolve how reversion guards are to be 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 [15] 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 5. Consider the following concrete examples of revision guards:

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

```
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 so is the global graph obtained by removing reversion guards from G. This guarantees communication soundness in presence of reverse executions.

4 From REGs to Erlang

This section shows how we map REGs into Erlang programs. This mapping corresponds to the definition of *projecting* the global view provided by REGs into Erlang implementations of their local view. Our encoding embraces the principles advocated in [8] and reviewed in Section 1: 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 3. In the following, we first describe the architecture of our solution. We then show how forward and reversed executions are rendered in it. We discuss our design choices and a few possible alternative solutions in Section 5.

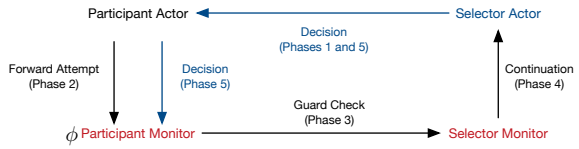


Fig. 4. Our architecture

4.1 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 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. This is emphasised by the arrows in Fig. 4, which are 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:

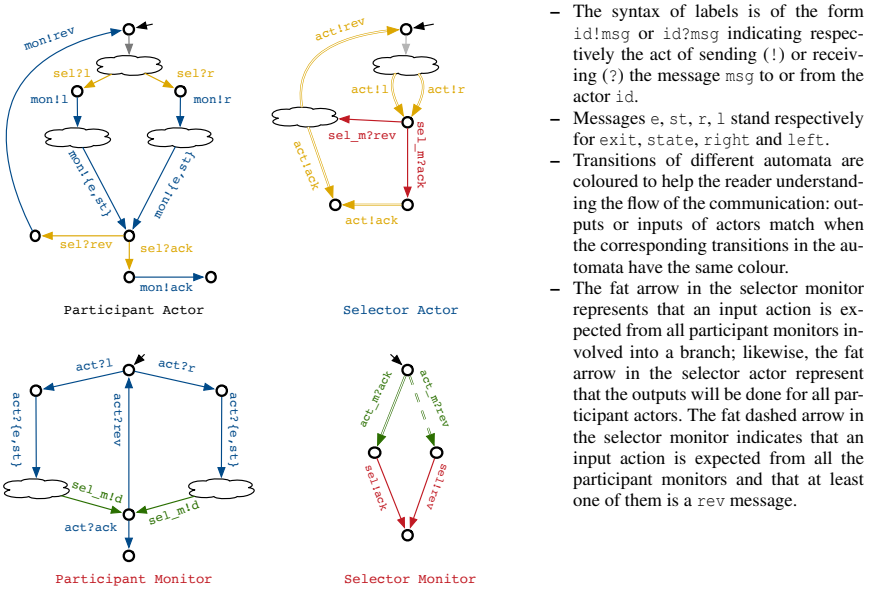
1. **Inception**: The selector actor (started at a branching point) decides which branch to execute and communicates its decision to the participants involved.
2. **Forward attempt**: Participant actors execute the selected branch accordingly and report their local state at the end of the branch to their 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 Fig. 4.

4.2 Branching actors & monitors

We now describe the behaviour of actors and monitors in a choice, with the help of their automata-like representation in Fig. 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 syntax of labels is of the form `id!msg` or `id?msg` indicating respectively the act of sending (`!`) or receiving (`?`) the message `msg` to or from the actor `id`.
- Messages `e`, `st`, `r`, `l` stand respectively for `exit`, `state`, `right` and `left`.
- Transitions of different automata are coloured to help the reader understanding the flow of the communication: outputs or inputs of actors match when the corresponding transitions in the automata have the same colour.
- The fat arrow in the selector monitor represents that an input action is expected from all participant monitors involved into a branch; likewise, the fat arrow in the selector actor represent that the outputs will be done for all participant actors. The fat dashed arrow in the selector monitor indicates that an input action is expected from all the participant monitors and that at least one of them is a `rev` message.

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

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—the yellow arrows in the automaton of Fig. 5. Upon the receipt of such a message, participant actors first forward this message to their monitor and then enter the second phase executing the branch—represented by the cloud in the automaton. The third phase starts (if the chosen branch does not diverge) 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 done by the `exit` message which also carries the local state of execution (described in Section 3). 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 gray 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, the participant monitor sends the selector

monitor either a request to *reverse* the branch (message `rev`). Otherwise the monitor sends a message to commit the choice (message `exit`). In Fig. 5 this is represented by the label `sel_m!d`, where `d` stands for decision. 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 Fig. 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 4.1 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 tried yet then the last executed branch is marked as "tried", a branch 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 otherwise, 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 selector is quite straightforward: 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.

4.3 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[5].

[5] These can however be found at `http://staff.um.edu.mt/afra1/rgg`.

```
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, spawn(sel_mon, [CP, self()])),
48   Sel =
49     case Attempt of
50       [] -> getBranch();
51       [left] -> right;
52       [right] -> left;
53       _ -> throw("panic....")
54     end,
55     P = participants(CP),
56     foreach(fun(X) -> X!{CP, Sel} end, P),
57     receive {CP,Outcome} ->
58       Decision =
59         case {Outcome,Attempt} of
60           {ack,_} -> ack;
61           {rev,[]} -> rev;
62           {_,_} -> ack
63         end
64     end,
65    foreach(fun(X) -> X!{CP, Decision} end, P),
66     case Decision of
67       rev -> sel_act(Attempt ++ [Sel], CP);
68       _   -> end_branch
69     end.

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

The code for the participant actor (lines 1-21) is parametrised with respect to `cp`, the value of the control point[6] 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 44-69. 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 48-54) and communicates its decision to the participants of the branch (lines 55-56, 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

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

phase of Section 4.1, 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 it 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 45-47) where `cp` is the value passed through the second parameters `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 72-78 and sent to the selector on line 79. The selector monitor awaits a message from all the participant monitors involved in the branch (lines 72-73), 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`.

5 Design Choices & Alternatives

We now discuss our design choices and some potential alternatives. As remarked earlier, the architecture and the coordination proposed here strives for a high degree of decoupling between the run-time support of the application and revision logics. In light of this, we tried to limit the overhead required to manage the reversible semantics proposed in Section 3.

Not surprisingly, the design choices we had to make mainly concerned the implementation of branches and the corresponding reversible behaviour. A first decision we had to take related to the realisation of the application logic part of the branches. Our solution introduces selector actors to implement the policy for selecting a branch. A plausible alternative could have been to let the actor corresponding to the active participant to manage the choice. We argue that such alternative has two main drawbacks. Firstly, it makes the projection of active and non-active participants less uniform, negatively impacting on the cohesion of the architecture. This lack of symmetry would also impact on the corresponding monitors, which would invariably become more complex—ideally, the monitor logic is kept as simple as possible, since this is conducive to correct, efficient code. Also, Erlang does not allow multi-threaded actors, hence this alternative would have introduced unnecessary dependencies between actors and monitors.

We remark that the decision process would typically be specified in the application logic of the active participant. For instance, one could specify priorities on branches or allow the same branch to be tried more than once and reversed only after a certain number of attempts have failed. Our current solution abstracts away from this, and adopts a non-deterministic policy (using an Erlang randomisation function) for simplicity.

Instead of dynamically spawning selector actors, we could statically generate them. This solution would simplify the projection operation trading on efficiency since, at run-time, *not* all branches are typically executed (*e.g.*, in the case of nested branching).

Local guards are designed to attain locally checkable conditions (cf. Section 3). A less ad-hoc mechanism could possibly be considered following the approach taken in [5] where "global" logical formulae (dubbed *global assertions*) are projected into local ones. This is a more complex approach that nevertheless could be worth exploring as it could lead to more expressive frameworks. For instance, it would allow the definition of hyperproperties [11] such as as those that compare the size of message queues at different actors over time. Note that this could require non-trivial interactions among monitors for exchanging local information (or more complex aggregation at the branch monitor). An intriguing research direction would be to explore to which extent these non-trivial interactions could be automatically derived by the projection of global conditions. Another crucial decision we had to contend with concerned the execution points at which the reversion guards should be checked. Here the range of possibilities is fairly broad and we opted for an "optimistic" policy, leaving the realisation of other alternative policies for future work. For instance, another alternative would be one where each monitor would continuously check the guard and trigger the reverse execution as soon as it is breached (instead of waiting for the completion of the branch). This option is interesting because it avoids the wasteful execution of the entire branch before trying to then reverse it. However, such a "preemptive" approach would also make actors and monitors more complex (*e.g.*, participant and selector actors would need to "poll" for message arrival) and would increase monitoring overheads as well.

6 Final remarks

We have presented a minimally-intrusive extension to global graph choreographies [20] 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 [29, 28, 16]. In [29] a reversible semantics for a subset of Erlang is given. The goal of [29] is a debugger based on a fully reversible semantics. To achieve this, the Erlang virtual machine is modified in order to keep track of computational history. Our goal is different since we focus on *controlled reversibility* [24]. 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 [28] 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 participant 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 [28] so as to generalise our reversible semantics to harness fault tolerance. This is not a trivial task, because the fault-tolerance mechanism of [28] needs to follow a specific protocol, making it unclear whether participants

can be automatically derived. In [16] 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 [31, 26, 27] have investigated the use of monitors to steer reversibility in concurrent systems. In [31] 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 [26, 27] 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 exploit this information to reconstruct previous states. As in our approach, in [27] participants and their monitors are derived from a global specification as well. We diverge from [26, 27] 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.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.
2. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
3. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. R. Lowry, C. S. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *TCS*, 336(2-3), 2005.
4. L. Bocchi, T.-C. Chen, R. Demangeon, K. Honda, and N. Yoshida. Monitoring networks through multiparty session types. In *FMOODS/FORTE 2013*, volume 7892 of *LNCS*, 2013.
5. L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, volume 6269, 2010.
6. I. Cassar and A. Francalanza. Runtime Adaptation for Actor Systems. In *RV*, volume 9333 of *LNCS*. Springer, 2015.
7. I. Cassar and A. Francalanza. On Implementing a Monitor-Oriented Programming Framework for Actor Systems. In *iFM*, 2016.

8. 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.
9. F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly, 2009.
10. F. Chen and G. Rosu. Towards Monitoring-Oriented Programming: A paradigm combining specification and implementation. *Electr. Notes Theor. Comput. Sci.*, 89(2):108–127, 2003.
11. M. R. Clarkson and F. B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, Sept. 2010.
12. P. Deniélou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP 2012*, 2012.
13. A. Desai, T. Dreossi, and S. A. Seshia. Combining model checking and runtime verification for safe robotics. In *RV*, LNCS. Springer, 2017.
14. E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3), 2002.
15. Erlang Run-Time System Application, Reference Manual Version 9.2. Available at http://erlang.org/doc/man/erlang.html.
16. J. Field and C. A. Varela. Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In *POPL 2005*. ACM, 2005.
17. A. Francalanza. A Theory of Monitors. In *FoSSaCS*, volume 9634 of *LNCS*. Springer, 2016.
18. A. Francalanza, L. Aceto, A. Achilleos, D. P. Attard, I. Cassar, D. D. Monica, and A. Ingólfsdóttir. A Foundation for Runtime Monitoring. In *Runtime Verification*, volume 10548 of *LNCS*. Springer, 2017.
19. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
20. R. Guanciale and E. Tuosto. An abstract semantics of the global view of choreographies. In *ICE*, 2016.
21. C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*. Morgan Kaufmann Publishers Inc., 1973.
22. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1), 2016. Extended version of a paper presented at POPL08.
23. K. Kejstová, P. Ročkai, and J. Barnat. From Model Checking to Runtime Verification and Back. In *RV*. Springer, 2017.
24. I. Lanese, C. A. Mezzina, and J.-B. Stefani. Controlled reversibility and compensations. In *RC 2012. Revised Papers*, volume 7581 of *LNCS*. Springer, 2012.
25. P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP runtime verification framework. *STTT*, 14(3):249–289, 2012.
26. C. A. Mezzina and J. A. Pérez. Causally consistent reversible choreographies: a monitors-as-memories approach. In *PPDP*, 2017.
27. C. A. Mezzina and J. A. Pérez. Reversibility in session-based concurrency: A fresh look. *J. Log. Algebr. Meth. Program.*, 90:2–30, 2017.
28. R. Neykova and N. Yoshida. Let it recover: multiparty protocol-induced recovery. In *CC*. ACM, 2017.
29. N. Nishida, A. Palacios, and G. Vidal. A reversible semantics for erlang. In *LOPSTR*, volume 10184 of *LNCS*. Springer, 2016.
30. K. Perumalla. *Introduction to Reversible Computing*. Chapman & Hall/CRC Computational Science. Taylor & Francis, 2013.
31. 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.
32. E. Tuosto and R. Guanciale. Semantics of global view of choreographies. *J. Log. Algebr. Meth. Program.*, 95:17 – 40, 2018.