Reliability and Fault-Tolerance by Choreographic Design*

Ian Cassar†
Reykjavik University
ianc@ru.is

Adrian Francalanza
University of Malta
adrian.francalanza@um.edu.mt

Claudio Antares Mezzina
IMT School for Advanced Studies Lucca, Italy
claudio.mezzina@imtlucca.it

Emilio Tuosto
University of Leicester, UK
emilio@le.ac.uk

Distributed programs are hard to get right because they are required to be open, scalable, long-running, and tolerant to faults. In particular, the recent approaches to distributed software based on (micro-)services where different services are developed independently by disparate teams exacerbate the problem. In fact, services are meant to be composed together and run in open context 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 in the program logic, which 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 industry. Our research agenda consists of (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.

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 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-related restrictions: 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.

*Partially supported by EU COST IC1405 (Reversible Computation - Extending Horizons of Computing).
†The research work disclosed in this publication is partially funded by the ENDEAVOUR Scholarships Scheme. "The scholarship may be part-financed by the European Union — European Social Fund"

Adrian Francalanza, Gordon J. Pace (Eds.): Second International Workshop
on Pre- and Post-Deployment Verification Techniques (PrePost 2017)
EPTCS 254, 2017, pp. 69–80, doi:10.4204/EPTCS.254.6

- 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 [17].

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. The typical mechanisms used to guarantee fault-tolerance are redundancy (typically to tackle hardware failures) and exception handling for software reliability. It has been observed (see e.g., [30]) 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 fault tolerance.

In this position paper, we advocate the use of choreographies to specify, analyse, and implement reliable strategies for fault-tolerance 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) and tries to give some hint of the advantages within our choreographic framework (Section 3). We draw some conclusions in Section 5.

2 Motivation

We are interested in *message-passing* frameworks, *i.e.,* models, systems, and languages where independently executing (distributed) components coordinate by exchanging messages. One archetypal model of the message-passing paradigm is the *actor model* [1] popularised by industry-strength language implementations such as those found in Akka (for both Scala and Java) [32], Elixir [31], and Erlang [9]. 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 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) [26, 10]. 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 mechanisms are 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 to be understood in isolation.

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 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 *w.r.t.* 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 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 [28]. We note that the approach followed in [28] 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 (possibly used at different levels of abstractions) 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 form 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 run-time monitors to be derived from their abstract models, to ease the development process and allow developer to focus on the application logic (such as in [6, 7]).

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 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 [12], where various pre-orders for monitor

descriptions are developed, and [13] where intrinsic monitor correctness criteria such as consistent detections are studied.

3 Choreographic models and failures

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

The starting point of our work will rely on two existing bodies of work. Our investigations will, on the one hand, be grounded on the Erlang monitoring framework developed and implemented in [6, 7], (surveyed in Section 3.1) and, on the other hand, be driven by the design of a choreographic model for distributed computation with global views and local projections of [24] (reviewed in Section 3.2).

3.1 A Runtime Adaptation Framework for Erlang Recovery Strategies

In [6, 7], the authors establish local recovery policies via runtime adaptation techniques. Runtime adaptation [22, 20] is an adaptive monitoring technique prevalent to long-running, highly available software systems, whereby system characteristics (*eg.,* its structure, locality *etc.*) are altered *dynamically* in response to runtime events (*eg.,* detected hardware faults or software bugs, changes in system loads), while causing *limited disruption* to the execution of the system. Numerous examples can be found in service-oriented architectures [29, 19] (*eg.,* cloud-services, web-services, *etc.*) for self-configuring, self-optimising and self-healing purposes; the inherent component-based, decoupled organisation of such systems facilitates the implementation of adaptive actions *affecting a subset* of the system while allowing other parts to continue executing normally.

$c,d \in \texttt{SPEC} ::=$	$\texttt{flag}$	(detect)	$\mid \texttt{end}$	(terminate)
	$\mid c \,\&\, d$	(conjunction)	$\mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ d$	(branch)
	$\mid \texttt{rec}\,\texttt{X}.c$	(recursion)	$\mid \texttt{X}$	(recursive call)
	$\mid [p]\ \texttt{rel}\ \vec{v}.\ c$	(guard)	$\mid *[p]\ \texttt{rel}\ \vec{v}.\ c$	(blocking guard)
	$\mid A(x)\ \texttt{rel}\ \vec{v}.\ c$	(asyn. adaptation)	$\mid S(x)\ \texttt{rel}\ \vec{v}.\ c$	(sync. adaptation)

Figure 1: Monitor Specification Syntax

The Logic In [6, 7], Cassar *et. al* developed the monitoring tool adaptEr[1] for synthesising adaptation monitors for actor systems developed in Erlang. Specifications in adaptEr are defined using the logic in Figure 1, consisting in 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.

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

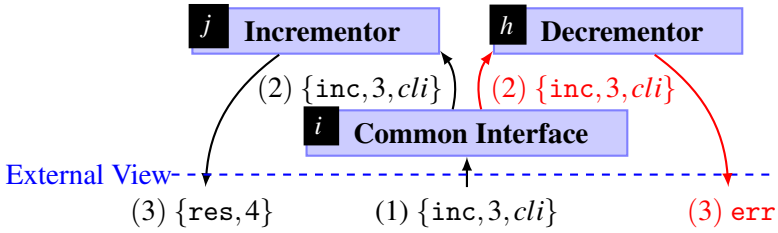


Figure 2: A server actor implementation offering integer increment and decrement services

The logic is defined over streams of visible events, α, generated by the monitored system made up of *actors* — independently-executing processes that are uniquely-identifiable by a process identifier, have their own local memory, and can either spawn other actors or interact with other actors in the system through asynchronous messaging; we use $i, j, h \in \text{PID}$ to denote the unique identifiers. Monitored events include the sending of messages, $i\,\texttt{>}\,j\,\texttt{!}\,v$, (containing the value v from actor with identifier i to actor j), the receipt of messages, $i\,\texttt{?}\,v$, (containing the value v received by actor i), function calls, $\texttt{call}(i,\{m,f,l\})$, (at actor i for function f in module m with argument list l) and function returns, $\texttt{ret}(i,\{m,f,a,v\})$ (at actor i for function f in module m with argument arity a and return value v). Event patterns, $p, q \in \text{PAT}$ follow a similar structure to that of events, but may contain term variables $x, y, z \in \text{VARS}$ that are bind to concrete values $v, u \in \text{VAL}$ (where $\text{PID} \subseteq \text{VAL}$), at runtime through pattern matching (we use $\vec{v}$ to denote lists of values).

The syntax in Figure 1 includes termination constructs `flag` and `end` respectively refer to a violation detection and an inconclusive verdict, along with *two* guarding constructs, $\texttt{[}p\texttt{]}\ \texttt{rel}\ \vec{i}.\ c$ and $\texttt{*[}p\texttt{]}\ \texttt{rel}\ \vec{i}.\ c$, instructing the respective monitor to observe system events that match event pattern p, and progressing as c if the match is successful. These constructs encompass directives for *blocking* and *releasing* actor executions, depending on the events observed. The guarding construct $\texttt{*[}p\texttt{]}\ \texttt{rel}\ \vec{i}.\ c$ is *blocking*, meaning that it *suspends* the execution the actor whose identifier is the *subject* of the event matched by the pattern (*eg.,* actor i is the subject in the events $i\,\texttt{>}\,j\,\texttt{!}\,v$, $i\,\texttt{<}\,j\,\texttt{?}\,v$, $\texttt{call}(i,\{m,f,l\})$ and $\texttt{ret}(i,\{m,f,a,r\})$). By contrast, the guarding construct $\texttt{[}p\texttt{]}\ \texttt{rel}\ \vec{i}.\ c$ does not block any actor when its pattern is matched. However, for both constructs $\texttt{[}p\texttt{]}\ \texttt{rel}\ \vec{i}.\ c$ and $\texttt{*[}p\texttt{]}\ \texttt{rel}\ \vec{i}.\ c$, pattern *mismatch* terminates monitoring, and *also* releases all the blocked actors in the list of identifiers $\vec{i}$. In addition to term variables, the abstract syntax in Figure 1 also assumes a distinct denumerable set of *formula variables* $\texttt{X}, \texttt{Y}, \ldots \in \text{VARS}$, used to define recursive specifications. It is also parametrised by a set of *decidable* boolean expressions, $b, b' \in \text{BOOL}$, and the aforementioned set of event patterns. Monitor specifications include commands for flagging violations, `flag`, and terminating (silently), `end`, conjunctions, $c_1\,\texttt{\&}\,c_2$, recursion, $\texttt{rec}\,\texttt{X}.c$, and conditionals to reason about data, $\texttt{if}\,b\,\texttt{then}\,c_1\,\texttt{else}\,c_2$ — we encode $\texttt{if}\,b\,\texttt{then}\,c_1$ as $\texttt{if}\,b\,\texttt{then}\,c_1\,\texttt{else}\,\texttt{end}$.

The syntax in Figure 1 also specifies two adaptation constructs, $A\texttt{(}j\texttt{)}\ \texttt{rel}\ \vec{i}.\ c$ and $S\texttt{(}j\texttt{)}\ \texttt{rel}\ \vec{i}.\ c$. Both constructs instruct the monitor to administer an adaptation action (A and S) on actor j, releasing the (blocked) actors in the list $\vec{i}$ afterwards, then progressing as c. The only difference between these two constructs is that the adaptation in $S\texttt{(}j\texttt{)}\ \texttt{rel}\ \vec{i}.\ c$, namely S, expects the target actor j to be *blocked* (*ie.,* synchronised with the monitor) when the adaptation is administered, and must therefore be blocked by some preceding guarding construct. Synchronous adaptations include, amongst others (see [7, 6]), $\texttt{restart(x)}\ \texttt{rel}\ \vec{v}.\ \texttt{c}$ and $\texttt{flush(x)}\ \texttt{rel}\ \vec{v}.\ \texttt{c}$ for restarting and emptying the mailboxes of misbehaving actors.

Example. Figure 2 depicts a server consisting of a front-end *Common Interface* actor with identifier i receiving client requests, a back-end *Incrementor* actor with identifier j, handling integer increment requests, and a back-end *Decrementor* actor h, handing decrement requests. A client sends service requests to actor i of the form $\{tag, arg, ret\}$ where *tag* selects the type of service, *arg* carries the service arguments and *ret* specifies the return address for the result (typically the client actor ID). The interface actor forwards the request to one of its back-end servers (depending on the *tag*) whereas the back-end servers process the requests, sending results (or error messages) to *ret*. The tool adaptEr allows us to specify safety properties such as (1), explained below:

$$\texttt{rec Y.*[}i\texttt{>\_!}\{\texttt{inc},x,y\}\texttt{] rel }\varepsilon\texttt{.}\left(\begin{array}{l}(\texttt{ [}j\texttt{>}y\texttt{!}\{\texttt{res},x+1\}\texttt{] rel }\varepsilon\texttt{.Y) \&}\\(\texttt{*[}z\texttt{>}y\texttt{!err] rel }\varepsilon\texttt{. if }z\in\{j,h\}\texttt{ then}\\\qquad\texttt{restart(i) rel }\varepsilon\texttt{. flush(z) rel [i,z] . Y)}\end{array}\right)\qquad(1)$$

It is a (recursive, `rec Y....`) property requiring that, from an *external* viewpoint, *every* increment request sent by i to either j or h, action $i\texttt{>\_!}\{\texttt{inc},x,y\}$, is followed by an answer from j to the address y carrying $x+1$, action $j\texttt{>}y\texttt{!}\{\texttt{res},x+1\}$ (recursing through variable `Y`). However, increment requests followed by an error message sent from *any* actor back to y, action $\texttt{\_<}y\texttt{?err}$, represent a violation which requires mitigation. To enable mitigation through *synchronous adaptations* it is crucial to block the respective actors beforehand, in case they contribute to a property violation. We therefore specify blocking guard $\texttt{*[}i\texttt{>\_!}\{\texttt{inc},x,y\}\texttt{] rel }\varepsilon\texttt{.}$ to block actor i upon forwarding the event to any one of the backend actors (*ie.,* either j or h), along with $\texttt{*[}z\texttt{>}y\texttt{!err] rel }\varepsilon\texttt{.}$ to block any z actor that generates an error. Before mitigating this error, the monitor inspects the value bound to data variable z by using an if-statement; this ensures that the adaptations are being performed on the back-end actors.

To mitigate the violation we then specify that the monitor should restart actor i with synchronous adaptation `restart(i) rel` ε`.` , and empty the mailbox of the back-end server—which may contain more erroneously forwarded messages—through adaptation `flush(z) rel [i,z].` (the actor to be purged is determined at runtime, where z is bound to identifier h from the previous action $\texttt{[}z\texttt{<}y\texttt{?err] rel }\varepsilon\texttt{.}$). Importantly, note that in the above execution (where h is the actor sending the error message), actor j is *not affected* by any adaptation action taken. The last adaptation also contains pids i and z in its release list; this ensures that whenever the received input number x is correctly incremented, these blocked actors are released before the monitor recurs

3.2 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 of choreographies [3]; a widely accepted informal description is the one suggested by W3C's [21]:

> [...] 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** [...]

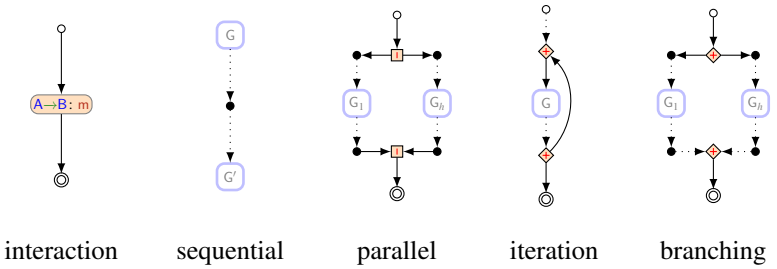
According to this description, a **global** and a **local** views are related as in the left-most diagram in Fig. 4 which evokes the following software engineering approach. First, an architect designs the global specification; then the architect uses the global specification to derive, via a 'projection' operation, a local specification for the distributed components the local ones; finally, programmers can 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. [24]). Those aspects are not in our scope here.

We choose two specific formalisms for global and local specifications. More precisely, we adapt to our needs the *global graphs* of [24] for global specifications and *communicating finite-state machines* (CFSMs) [5] to formalise local views of choreographies.

Global Specifications Global graphs can be seen as a graphical model of distributed work-flows and communicating machines (a well established model for communication protocol design) fix the distribution and communication model. We describe global graphs using their graphical notation reported in Fig. 3 featuring simple *interactions* (e.g., party A sending a message m to party B), *sequential* or *parallel* compositions of global graphs, *iteration* of global graphs, or *choice* between alternative global graphs.



A global graph G is a rooted graph with a single "enter" and a single "exit" nodes, called *source* and *sink* respectively and depicted as ○ and ◎. Other nodes are drawn as • while dotted edges have a "meta-meaning" to single out source and sink nodes of sub-graphs. More precisely, a dotted edge from a node • to a boxed G means that • is the source of G; similarly, a dotted edge from a boxed G to • means that • is the sink of G. For instance, the top-most dotted edge for the sequential composition identifies the sink node of G while the other dotted edge identifies the source node of G′; in other words, the sequential composition of G and G′ is obtained by coalescing the sink of G with the source of G′.

Figure 3: A graphical notation for choreographies

The semantics of a global graph G is given in terms of partial-orders on the communication *events* of G. For space limitations, we can only give an intuitive idea of such semantics (see [18] for details). The semantics of an interaction is straightforward: the output event of the sender must precede the input event of the receiver. Likewise, the semantics of parallel composition is obvious: the events of a thread G_i do not have any order relation with those of a thread G_j with $i \neq j$. The other cases are more delicate. For iteration, we take the unfolding of the loop and propagate the order accordingly (this is similar to what done with processes of Petri nets). The semantics of G;G′ is defined provided that (i) the semantics of G and G′ are defined, (ii) each input event in G′ follows each output event in G in the order built by taking the reflexo-transitive closure of union of semantics of G and of G′ after adding the dependencies between the events in G and those in G′ with the same subject. The semantics of a choice is defined provided that the semantics of each branch G_i is defined and that the choice is *well-branched*, namely that there is only one party deciding which branch to take and that all the other parties involved in the choice can discriminate using their input events.

Local Specifications We adopt systems of CFSMs [5] 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 [18].

4 The Proposed Approach

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 4. The left-most part of the diagram illustrates the top-down approach of choreographies of the application logic described in Section 3.2. We propose to develop a similar approach for the recovery logic as depicted in the right-most part of Fig. 4, 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 [7, 2] (unlike the choreographies for the application logic, where each local view typically yields one component).

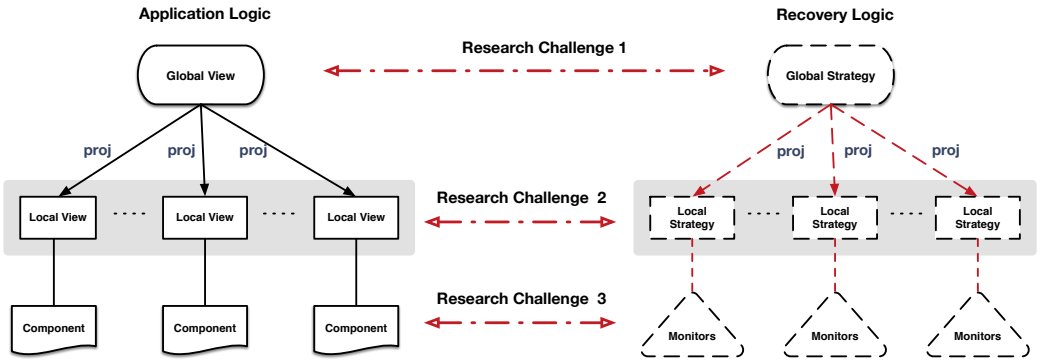


Figure 4: A Global-Local approach to Adaptation Strategies

Models to express Global and Local Strategies. Choreographic models have to 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 principle of monitors programming by looking at state-of-the-art techniques. For example,

the (global) recovery logic could allow us to specify *recovery* 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 [27] who extend the global graphs reviewed in Section 3.2 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 and then define projection operations to automatically obtain adaptEr monitors.

For example, consider the following scenario in which a party A is required to plan a trip where he may choose to take a flight or to travel via train. Depending on this choice his behaviour differs given that if he books a flight, he must also book a taxi for the date, while if he takes the train he can manage to schedule a meeting for the very same day since the station is close-by. Moreover, if he chooses to take the flight but is then unable to book the flight, he must then revert back to the train option. Following the approach of [27] this can be expressed by the following (global) choreography:

$$\mathsf{A}\to\mathsf{B}\colon \mathsf{FlightAndTaxi};\mathsf{G}_1 \ \texttt{unless}\ \phi_1 + \mathsf{A}\to\mathsf{C}\colon \mathsf{Train};\mathsf{G}_2 \ \texttt{unless}\ \mathit{ff}$$

where G_1 and G_2 describe the global behaviour for the system, followed by the choice guard ϕ_1 which expresses the fact that the taxi is unavailable; ff represents the falsehood condition. Starting from this global choreography, the appropriate local strategies must be derived accordingly. For instance, one way to go about this is by creating an encoding function $\mathsf{enc}(\phi_1)$ that allows for redefining choice guards (such as ϕ_1) in terms of adaptEr's logic. In this way the designated monitor can then verify the entire execution branch *wrt.* the encoded version of G, *ie.,* $\mathsf{enc}(\phi_1)$.Alternatively, we can also extend adaptEr's logic to allow for defining choreography guards as patterns within adaptEr's guards in the lines of $\texttt{*[}\phi\texttt{]}\ \texttt{rel}\ \vec{v}\texttt{.}\ c$.

Furthermore, support for additional adaptation constructs such as `roll(i) rel` $\vec{v}$`.` must be added in order to augment reversibility features within adaptEr. Such an adaptation should permit the monitor to reverse specific parts of the computation of an offending actor i along with the computation of other actors which had interacted with i. For instance, a local strategy for A, while executing the left branch of the choice, can be defined in terms of adaptEr's logic as follows:

$$\texttt{rec Y.*[act(A)] rel}\ \varepsilon\texttt{.}\ \begin{pmatrix} \texttt{(enc(}\neg\phi_1\texttt{)roll(A) rel A. Y)} \\ \texttt{\&(enc(}\phi_1\texttt{)Y)} \end{pmatrix}$$

The above script states that every time A performs some action $\mathsf{act}(A)$ (regardless of the type of action), this is blocked such that whenever the encoded version of ϕ_1 (*ie.,* $\mathsf{enc}(\phi_1)$) evaluates to false, a reversing adaptation is applied to A via `roll(A) rel A.` which then releases A; otherwise, the computation of A is left intact since the monitor releases A immediately. Naturally the reversing adaptation has to revert all the actions of A and to notify the parties involved in these actions.

Properties of Recovery Logic. We have to 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 when errors occur. For example, the recovery strategy could guarantee *causal consistency*, namely that a safe state is one that the execution reached previously. 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 Section 3.1 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 need to 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 studies (*eg.,* [8, 4]).

Another potential avenue worth considering is the work on monitorability [14] that relates the behaviour of the monitor to that specified by the correctness property of interest; the work in [16] 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 to adaptability and enforceability with respect to the local strategy derived from the global specification.

Once we identify and formalise our notions of compliance, we then need to 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).

Seamless Integration. A key driving principle of our proposed programme is that the recovery logic has to be orthogonal to the application logic. This separation of concerns would allow 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 will have to be integrated via appropriate code instrumentation mechanisms to provide a unified system that offers fault-tolerance capabilities. 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 to fault-tolerance 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.

5 Conclusions

We have proposed an approach for the structured development of recovery strategies for distributed applications, realising the choreographed monitor arrangements proposed in earlier work such as [11, 15, **?**]. The proposal build on existing work dealing with runtime adaptations for asynchronous systems on the one hand, and on choreographies projected from global specifications on the other. An initial step toward the proposed approach, has been explored in [27] where the recovery strategy is directly integrated into the global graph model by specifying some roll-back conditions on choice branches. In this way, during a coordinated choice if the condition does not any more then all the parties involved into the choice

are reverted to a state previous the choice, and another branch can be taken. Starting from the global specification where branches of a choice can be enriched with guards (triggering the possible roll-back) *controlled reversibility* [23] is enforced into local views. Local views are represented by *reversible* CFSMs keeping track of the entire computation history, and local strategies are just a set of constraints to be met in order to revert a certain choice. The framework of [27] has to be extended in order to allow compliance with monitors implementing a coordinated (and distributed) roll-back. Moreover, it could be used as test-bed for different recovery strategies by exploiting the underlying reversible substrate. Naturally the framework has to be refined in a way to allow for recovery strategies well-formedness and decidability.

References

[1] Gul A. Agha (1990): *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence, MIT Press, doi:10.1137/1030027.

[2] Duncan Paul Attard & Adrian Francalanza (2016): *A Monitoring Tool for a Branching-Time Logic*. In: *RV*, *LNCS* 10012, Springer, pp. 473–481, doi:10.1007/978-3-319-46982-9_31.

[3] Davide Basile, Pierpaolo Degano, Gian-Luigi Ferrari & Emilio Tuosto (2016): *Relating two automata-based models of orchestration and choreography*. *JLAMP* 85(3), pp. 425 – 446, doi:10.1016/j.jlamp.2015.09.011.

[4] G. Bernardi & M. Hennessy: *Mutually Testing Processes*. *LMCS* 11, doi:10.2168/LMCS-11(2:1)2015.

[5] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. *Journal of the ACM* 30(2), pp. 323–342, doi:10.1145/322374.322380.

[6] Ian Cassar & Adrian Francalanza (2015): *Runtime Adaptation for Actor Systems*. In Ezio Bartocci & Rupak Majumdar, editors: *RV2015*, *LNCS* 9333, Springer, pp. 38–54, doi:10.1007/978-3-319-23820-3_3.

[7] Ian Cassar & Adrian Francalanza (2016): *On Implementing a Monitor-Oriented Programming Framework for Actor Systems*. In: *IFM 2016*, *LNCS* 9681, Springer, pp. 176–192, doi:10.1007/978-3-319-33693-0_12.

[8] G. Castagna, N. Gesbert & L. Padovani (2009): *A theory of contracts for Web services*. *ACM Trans. Program. Lang. Syst.* 31(5), doi:10.1145/1538917.1538920.

[9] Francesco Cesarini & Simon J. Thompson (2009): *Erlang Behaviours: Programming with Process Design Patterns*. In: *CEFP 2009, Budapest, Hungary*, pp. 19–41, doi:10.1007/978-3-642-17685-2_2.

[10] Feng Chen, Dongyun Jin, Patrick Meredith & Grigore Roşu (2009): *Monitoring Oriented Programming - A Project Overview*. In: *ICICIS'09*, ACM, pp. 72–77.

[11] Mario Coppo, Mariangiola Dezani-Ciancaglini & Betti Venneri (2014): *Self-Adaptive Monitors for Multiparty Sessions*. In: *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, IEEE Computer Society, pp. 688–696, doi:10.1109/PDP.2014.18.

[12] Adrian Francalanza (2016): *A Theory of Monitors - (Extended Abstract)*. In: *FoSSaCS*, *LNCS* 9634, Springer, pp. 145–161, doi:10.1007/978-3-662-49630-5_9.

[13] Adrian Francalanza (2017): *Consistently-Detecting Monitors*. In: *CONCUR*, LNCS, Springer. (to appear).

[14] Adrian Francalanza, Luca Aceto & Anna Ingolfsdottir (2017): *Monitorability for the Hennessy–Milner logic with recursion*. *Formal Methods in System Design*, pp. 1–30, doi:10.1007/s10703-017-0273-z.

[15] Adrian Francalanza, Andrew Gauci & Gordon J. Pace (2013): *Distributed System Contract Monitoring*. *The Journal of Logic and Algebraic Programming (JLAP)* 82(5-7), pp. 186–215, doi:10.1016/j.jlap.2013.04.001.

[16] Adrian Francalanza & Aldrin Seychell (2015): *Synthesising Correct Concurrent Runtime Monitors*. *Formal Methods in System Design (FMSD)* 46(3), pp. 226–261, doi:10.1007/s10703-014-0217-9.

[17] Jim Gray (1986): *Why Do Computers Stop and What Can Be Done About It?* In: *SRDS*, IEEE, doi:10.1109/MC.1983.1654340.

[18] Roberto Guanciale & Emilio Tuosto (2016): *An Abstract Semantics of the Global View of Choreographies*. In: *ICE 2016, Heraklion, Greece*, pp. 67–82, doi:10.4204/EPTCS.223.5.

[19] Florian Irmert, Thomas Fischer & Klaus Meyer-Wegener (2008): *Runtime Adaptation in a Service-oriented Component Model*. SEAMS '08, ACM, pp. 97–104, doi:10.1145/1370018.1370036.

[20] Mehdi Amoui Kalareh (2012): *Evolving Software Systems for Self-Adaptation*. Ph.D. thesis, University of Waterloo, Ontario, Canada.

[21] Nickolas Kavantzas, Davide Burdett, Gregory Ritzinger, Tony Fletcher & Yves Lafon (2004): *Web Services Choreography Description Language Version 1.0*. `http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217`.

[22] Stephen Kell (2008): *A Survey of Practical Software Adaptation Techniques*. 14(13), pp. 2110–2157. doi:10.3217/jucs-014-13-2110.

[23] Ivan Lanese, Claudio Antares Mezzina, Alan Schmitt & Jean-Bernard Stefani (2011): *Controlling Reversibility in Higher-Order Pi*. In: *CONCUR*, pp. 297–311, doi:10.1007/978-3-642-23217-6_20.

[24] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2015): *From Communicating Machines to Graphical Choreographies*. In: *POPL15*, pp. 221–232, doi:10.1145/2676726.2676964.

[25] Julien Lange, Emilio Tuosto & Nobuko Yoshida (2017): *A tool for choreography-based analysis of message-passing software*. ACM. To appear. Available at `http://www.cs.le.ac.uk/~et52/chorgram_betty_ch.pdf`.

[26] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen & Grigore Roşu (2011): *An Overview of the MOP Runtime Verification Framework*. *International Journal on Software Techniques for Technology Transfer*, pp. 249–289, doi:10.1007/s10009-011-0198-6.

[27] Claudio Antares Mezzina & Emilio Tuosto (2017): *Choreographies for Automatic Recovery*. *CoRR* abs/1705.09525.

[28] Rumyana Neykova & Nobuko Yoshida (2017): *Let it recover: multiparty protocol-induced recovery*. In: *CC2017*, pp. 98–108, doi:10.1145/3033019.3033031.

[29] Peyman Oreizy, Nenad Medvidovic & Richard N. Taylor (2008): *Runtime Software Adaptation: Framework, Approaches, and Styles*. ICSE Companion '08, ACM, New York, NY, USA, pp. 899–910, doi:10.1145/1370175.1370181.

[30] Paul Rook (1990): *Software Reliability Handbook*. Elsevier Science Inc., New York, NY, USA.

[31] Dave Thomas (2014): *Programming Elixir: Functional , Concurrent , Pragmatic , Fun*, 1st edition. Pragmatic Bookshelf.

[32] Derek Wyatt (2013): *Akka Concurrency*. Artima Incorporation, USA.