Towards Choreographic-Based Monitoring

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

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

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

1 Introduction

Distributed applications are notoriously complex and guaranteeing their correctness, robustness, and resilience is particularly challenging. These reliability

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

I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 128–150, 2020.
https://doi.org/10.1007/978-3-030-47361-7_6

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

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

We advocate the use of choreographies to specify, analyse, and implement reliable strategies for recovery and monitoring of distributed message-passing applications. We strive towards a setup that teases apart the main program logic from the coordination of error detection, correction and recovery. The rest of the paper motivates our approach: Sect. 2 further introduces our motivations,

Sect. 3 presents our (abstract) model by posing some research challenges, while Sects. 4 to 6 provide and instance of such model. We draw some conclusions in Sect. 7.

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

2 Motivation

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

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

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

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

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

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

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

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

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

such an investigation are [20], where various pre-orders for monitor descriptions are developed, and [21] where intrinsic monitor correctness criteria such as consistent detections are studied.

3 The Model

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

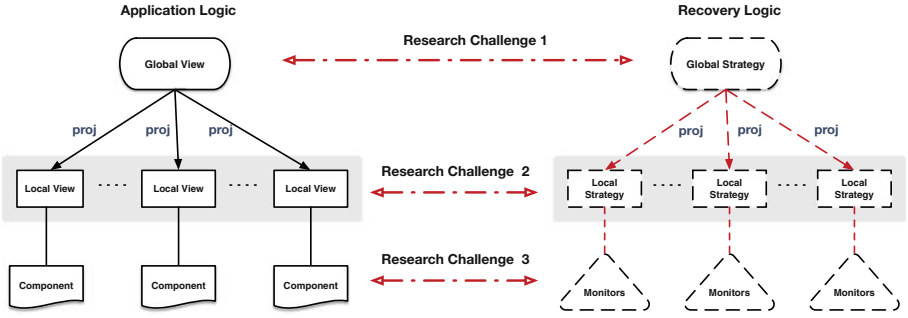


Fig. 1. A global-local approach to adaptation strategies incorporating the three research challenges identified in Sect. 2

Models to Express Global and Local Strategies. Choreographic models should be equipped with features allowing us to design and analyse the recovery logic of systems. This requires, on the one hand, the identification of suitable linguistic mechanisms for expressing global/local strategies and, on the other hand, to define principles of monitors programming by looking at state-of-the-art techniques. For example, the (global) recovery logic should allow us to specify *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 [39] who extend the global graphs reviewed in Sect. 4.1 with *reversibility guards* to recover the system when it reaches undesired configurations. A promising research direction in this respect is to extend the language of reversibility guards with the patterns featured by adaptEr [10–12] and then define projection operations to automatically obtain adaptEr monitors.

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

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

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

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

Once we identify and formalise our notions of compliance, we should study their decidability properties, and investigate approaches to check compliance

such as type-checking or behavioural equivalence checking (*e.g.,* via testing preorders or bisimulations [3,20]).

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

4 An Instance

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

Local Runtime Adaptation + Static Choreography Specifications = Choreographed MOP

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

4.1 Global and Local Specifications

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

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

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

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

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

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

Hereafter we fix two disjoint sets $\mathcal{P}$ and $\mathcal{M}$; the former is a finite set of *participants* (ranged over by A, B, etc.) and $\mathcal{M}$ is the set of *messages* (ranged over by m, x, etc.). To exchange messages and coordinate with each other, participants use asynchronous point-to-point communication via *channels* following the *actor model* [5,29]. 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

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

A global graph can be a simple interaction $\mathsf{A}{\to}\mathsf{B}\colon \mathsf{m}$ (for which we require $\mathsf{A} \neq \mathsf{B}$), the sequential composition $\mathsf{G};\mathsf{G}'$ of G and G', the parallel composition (for which the participants of G and of G' are disjoint), a nondeterministic choice $\mathsf{G}{+}\mathsf{G}'$ between G and G', or the iteration $*\mathsf{G}@\mathsf{A}$ of G. The syntax captures the structure of a visual language of distributed workflows illustrated in Fig. 2. Each global graphs G can be represented as a rooted diagram with a single source node and a single sink node respectively represented as $\circ$ and $\circledcirc$. Other nodes are drawn as $\bullet$ and a dotted edge from/to a $\bullet$-node singles out the source/sink nodes the edge connects to. For instance, in the diagram for the sequential composition, the top-most edge identifies the sink node of G and the other edge identifies the

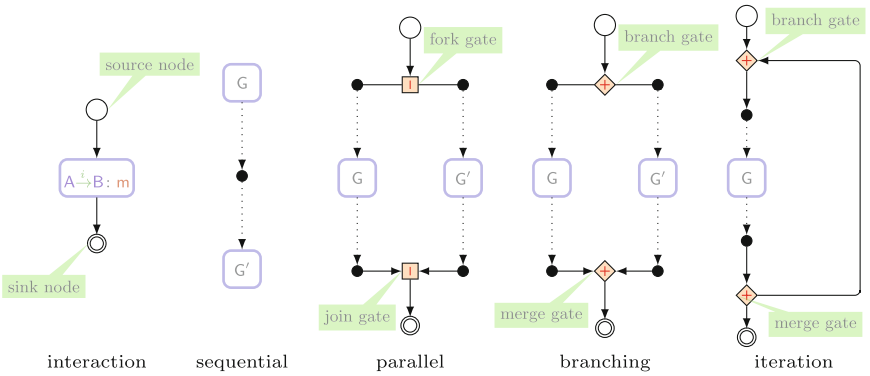


Fig. 2. A visual notation for global graphs

source node of G'; intuitively, $\bullet$ is the node of the sequential composition of G and G' obtained by "coalescing" the sink of G with the source of G'. In our diagrams, branches and forks are marked respectively by ◈ and ◫ nodes; also, to each branch/fork nodes corresponds a "closing" gate merge/join gate.

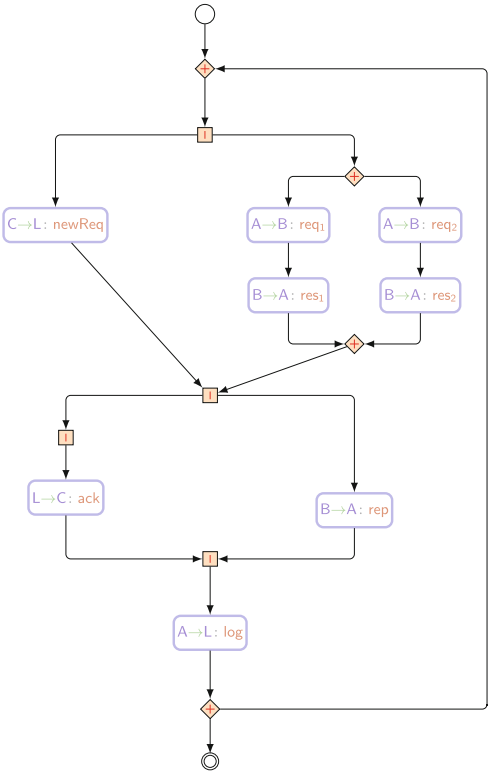
Example 1. Consider a protocol where iteratively participant C sends a newReq message to a logging service L. In parallel, a C's partner, A, makes either requests of either type req_1 or type req_2 to a service B, which, in turn, replies via two different types of responses, namely res_1 and res_2. Once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

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

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

The (forward) semantics of global graphs can be defined in terms of partial orders of communication events [28,45]. We do not present this semantics here (the reader is referred to [28,45]) for space limitations; instead, we give only a brief and informal account through a "token game" similar to the one of Petri nets based on Fig. 3. The token game would start from the source node and flow down along the edges in the diagram as described by the test in Fig. 3.

For the semantics of global graphs to be defined, *well-branchedness* [28,45] is a key requirement. This is a simple condition guaranteeing that *all* the participants involved in a distributed choice follow a same branch. Well-branchedness requires that each branch in a global graph (i) has a unique *active* participant (that is a



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

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

unique participant taking the decision on which branch to follow) and (*ii*) that any other participant is *passive*, namely that it is either able to ascertain which branch was selected from the messages it receives or it does not play any role in the branching.

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

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

of message m to B, then m is put in the buffer from A to B and A moves to state q'. Similarly, when B is in a state q with a transition to a state q' whose label is an input of m from B and the m is on the top of the buffer from A to B then B pops m from the buffer and moves to state q'.

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

5 Global Graphs for Reversibility

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

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

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

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

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

$$ {}_i.\big(*\mathsf{G}@\mathsf{A}\big) \qquad (2)$$

that satisfy the following conditions:

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

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

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

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

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

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

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

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

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

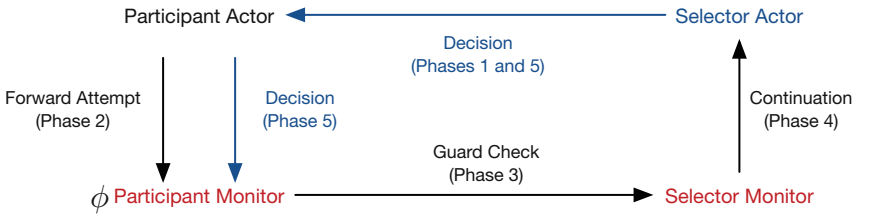


Fig. 4. The instrumentation architecture connecting participant actors, coordinating (selector) actors and their respective monitor actors

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

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

6 From REGs to Erlang

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

6.1 Architecture

The abstract architecture of our proposal is given in Fig. 4. Each participant of a REG is mapped to a *pair* of Erlang actors, the *participant actor* and the *participant monitor* which liaise with one another in order to realise reversible distributed choices. The execution of a distributed choice is supported by another pair of (dynamically generated) actors, the *selector actor* which liaises with its corresponding *selector monitor*. The basic idea is that participant and selector actors are in charge of executing the forward logic part of the choice while their respective monitors deal with the reversibility logic.

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

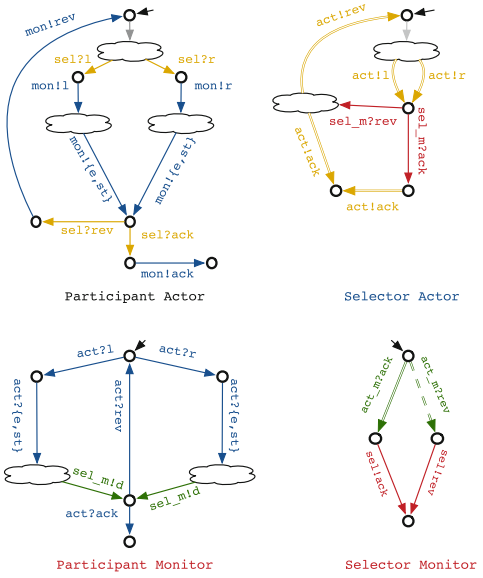
1. **Inception**: The selector actor (started at a branching point) decides which branch to execute and communicates its decision to the participants involved.
2. **Forward attempt**: Participant actors execute the selected branch accordingly and report their local state at the end of the branch to their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

These phases roughly correspond to the arrows in Fig. 4.

6.2 Branching Actors and Monitors

We now describe the behaviour of actors and monitors in a choice, with the help of their automata-like representation in 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 choice; all other participant actors just wait for the selector's decision. The act of spawning the selector arrow by the active participant is represented in Fig. 5 via the gray arrow and the cloud in the automaton of the participant actor. Subsequently, all the actor participants involved in a branch will wait from the selector to instruct them with the branch (either left or right) to take—these are the yellow arrows in the automaton of 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. Unless the chosen branch diverges, the third phase starts when participant actors finish the branch (possibly at different times) and they signal to their monitor that they are ready to exit the choice. This is signalled by the `exit` message which also carries the local state of execution (described in Sect. 5). At this point, participant actors take part only 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`, and `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 last phase: they receive from the selector either an `ack` message (confirming that the choice has been resolved) or a `rev` message to reverse the execution. In either case, they propagate the message to their monitor and either "commit" the branch or return to the state that waits for the message dictating the next branch to take. Participant actors behave uniformly but for the active one, which has the additional task of spawning the selector at the very beginning (for non-active participants the grey transition is an internal step not affecting communications).

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

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

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

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

6.3 Compiling to Erlang

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

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

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

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

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

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

The Erlang process spawned by a participant actor implementing the selector actor executes the function on lines 44–70. This function takes two parameters: the `Attempt` representing the branches chosen so far and the control point `CP` identifying the choice. The former parameter is a list of atoms `left` and `right`; note that the empty list is passed initially when the process is spawned and that (in our case) the size of this list should never exceed 1. As discussed above, the selector chooses a branch (lines 49–55) and communicates its decision to the participants of the branch (lines 56–57, where `participants` is computed at compile time, from the global graph script, and returns the participants of a branch given its control point). Finally, the selector enters the fourth phase of Sect. 6.1, waiting for the message from its monitor, and decides accordingly how to continue the execution of the choreographed choice.

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

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

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

7 Conclusions

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

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

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

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

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

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

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

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

References

1. Erlang run-time system application, reference manual version 9.2 (2017)
2. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. Proc. ACM Program. Lang. **3**(POPL), 52:1–52:29 (2019)
3. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Testing equivalence vs. runtime monitoring. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming. LNCS, vol. 11665, pp. 28–44. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21485-2_4
4. Aceto, L., Cassar, I., Francalanza, A., Ingólfsdóttir, A.: On runtime enforcement via suppressions. In: 29th International Conference on Concurrency Theory, CONCUR 2018, Beijing, China, 4–7 September 2018. LIPIcs, vol. 118, pp. 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
5. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press Series in Artificial Intelligence. MIT Press, Cambridge (1990)
6. Attard, D.P., Francalanza, A.: A monitoring tool for a branching-time logic. In: Falcone, Y., Sánchez, C. (eds.) RV 2016. LNCS, vol. 10012, pp. 473–481. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46982-9_31
7. Basile, D., Degano, P., Ferrari, G.-L., Tuosto, E.: Relating two automata-based models of orchestration and choreography. JLAMP **85**(3), 425–446 (2016)
8. Bernardi, G., Hennessy, M.: Mutually testing processes. LMCS **11**(2), 1–23 (2015)
9. Brand, D., Zafiropulo, P.: On communicating finite-state machines. J. ACM **30**(2), 323–342 (1983)
10. Cassar, I., Francalanza, A.: Runtime adaptation for actor systems. In: Bartocci, E., Majumdar, R. (eds.) RV 2015. LNCS, vol. 9333, pp. 38–54. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23820-3_3
11. Cassar, I., Francalanza, A.: On implementing a monitor-oriented programming framework for actor systems. In: Ábrahám, E., Huisman, M. (eds.) IFM 2016. LNCS, vol. 9681, pp. 176–192. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33693-0_12
12. Cassar, I., Francalanza, A., Attard, D.P., Aceto, L., Ingólfsdóttir, A.: A suite of monitoring tools for Erlang. In: Reger, G., Havelund, K. (eds.) RV-CuBES 2017. An

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

International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools. Kalpa Publications in Computing, vol. 3, pp. 41–47. EasyChair (2017)
13. Cassar, I., Francalanza, A., Mezzina, C.A., Tuosto, E.: Reliability and fault-tolerance by choreographic design. In: PrePost@iFM. EPTCS, vol. 254 (2017)
14. Castagna, G., Gesbert, N., Padovani, L.: A theory of contracts for web services. ACM Trans. Program. Lang. Syst. **31**(5), 1–61 (2009)
15. Cesarini, F., Thompson, S.: Erlang behaviours: programming with process design patterns. In: Horváth, Z., Plasmeijer, R., Zsók, V. (eds.) CEFP 2009. LNCS, vol. 6299, pp. 19–41. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17685-2_2
16. Chen, F., Jin, D., Meredith, P., Roşu, G.: Monitoring oriented programming - a project overview. In: Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS 2009), pp. 72–77. ACM (2009)
17. de Vries, F., Pérez, J.A.: Reversible session-based concurrency in Haskell. In: Pałka, M., Myreen, M. (eds.) TFP 2018. LNCS, vol. 11457, pp. 20–45. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18506-0_2
18. Deniélou, P.-M., Yoshida, N.: Multiparty session types meet communicating automata. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 194–213. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28869-2_10
19. Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In: POPL 2005. ACM (2005)
20. Francalanza, A.: A theory of monitors - (extended abstract). In: Jacobs, B., Löding, C. (eds.) FoSSaCS 2016. LNCS, vol. 9634, pp. 145–161. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49630-5_9
21. Francalanza, A.: Consistently-detecting monitors. In: 28th International Conference on Concurrency Theory, CONCUR 2017, 5–8 September 2017. LIPIcs, vol. 85, pp. 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
22. Francalanza, A., Aceto, L., Ingolfsdottir, A.: Monitorability for the Hennessy-Milner logic with recursion. Formal Methods Syst. Des. **51**, 1–30 (2017). https://doi.org/10.1007/s10703-017-0273-z
23. Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible choreographies via monitoring in Erlang. In: Bonomi, S., Rivière, E. (eds.) DAIS 2018. LNCS, vol. 10853, pp. 75–92. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93767-0_6
24. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
25. Francalanza, A., Seychell, A.: Synthesising correct concurrent runtime monitors. Formal Methods Syst. Des. (FMSD) **46**(3), 226–261 (2015). https://doi.org/10.1007/s10703-014-0217-9
26. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 370–384. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_26
27. Gray, J.: Why do computers stop and what can be done about it? In: SRDS. IEEE (1986)
28. Guanciale, R., Tuosto, E.: An abstract semantics of the global view of choreographies. In: ICE 2016, Heraklion, Greece, pp. 67–82 (2016)

29. Hewitt, C., Bishop, P., Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In: IJCAI. Morgan Kaufmann Publishers Inc. (1973)
30. Kavantzas, N., Burdett, D., Ritzinger, G., Fletcher, T., Lafon, Y.: Web services choreography description language version 1.0 (2004). http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217
31. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Controlled reversibility and compensations. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 233–240. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_19
32. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
33. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebraic Methods Program. **100**, 71–97 (2018)
34. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. In: POPL, pp. 221–232 (2015)
35. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. Int. J. Softw. Tech. Technol. Transf. **14**, 249–289 (2011)
36. Meyer, M.: Continuous integration and its tools. IEEE Softw. **31**(3), 14–16 (2014)
37. Mezzina, C.A., Pérez, J.A.: Causally consistent reversible choreographies: a monitors-as-memories approach. In: PPDP (2017)
38. Mezzina, C.A., Pérez, J.A.: Reversibility in session-based concurrency: a fresh look. J. Log. Algebr. Meth. Program. **90**, 2–30 (2017)
39. Mezzina, C.A., Tuosto, E.: Choreographies for automatic recovery. CoRR, abs/1705.09525 (2017)
40. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: CC. ACM (2017)
41. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_18
42. Rook, P.: Software Reliability Handbook. Elsevier Science Inc., New York (1990)
43. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000)
44. Thomas, D.: Programming Elixir: Functional, Concurrent, Pragmatic, Fun, 1st edn. Pragmatic Bookshelf (2014)
45. Tuosto, E., Guanciale, R.: Semantics of global view of choreographies. J. Log. Algebr. Meth. Program. **95**, 17–40 (2018)
46. Wyatt, D.: Akka Concurrency. Artima Incorporation, USA (2013)

Reversibility in Chemical Reactions

Stefan Kuhn[1(✉)], Bogdan Aman[2,3], Gabriel Ciobanu[2,3], Anna Philippou[4], Kyriaki Psara[4], and Irek Ulidowski[5]

[1] School of Computer Science and Informatics, De Montfort University, Leicester, UK
stefan.kuhn@dmu.ac.uk
[2] Romanian Academy, Institute of Computer Science, Iaşi, Romania
[3] Faculty of Computer Science, A.I. Cuza University, Iaşi, Romania
{bogdan.aman,gabriel}@info.uaic.ro
[4] Department of Computer Science, University of Cyprus, Nicosia, Cyprus
{annap,kpsara01}@cs.ucy.ac.cy
[5] School of Informatics, University of Leicester, Leicester, UK
irek.ulidowski@leicester.ac.uk

Abstract. In this chapter we give an overview of techniques for the modelling and reasoning about reversibility of systems, including out-of-causal-order reversibility, as it appears in chemical reactions. We consider the autoprotolysis of water reaction, and model it with the Calculus of Covalent Bonding, the Bonding Calculus, and Reversing Petri Nets. This exercise demonstrates that the formalisms, developed for expressing advanced forms of reversibility, are able to model autoprotolysis of water very accurately. Characteristics and expressiveness of the three formalisms are discussed and illustrated.

Keywords: Reversible computation · Reaction modelling · Calculus of Covalent Bonding · Bonding Calculus · Reversing Petri Nets

1 Introduction

Biological reactions, pathways, and reaction networks have been extensively studied in the literature using various techniques, including process calculi and Petri nets. Initial research was mainly focused on reaction rates by the modelling and simulating networks of reactions, in order to analyse or even predict the common paths through the network. Reversibility was not considered explicitly. Later on reversibility started to be taken into account, since it plays a crucial rôle in many processes, typically by going back to a previous state in the system. Two common types of reversibility are backtracking and causally-consistent reversibility [8,19,25]. Backtracking executes exactly the inverse order of the forward execution, and causally-consistent reversibility allows undoing effects before causes, but not necessarily in the exact inverse order. Beyond backtracking and

The authors acknowledge partial support of COST Action IC1405 on Reversible Computation - Extending Horizons of Computing.

I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 151–176, 2020.
https://doi.org/10.1007/978-3-030-47361-7_7

causally-consistent reversibility, there is a more general form of reversibility, known as out-of-causal-order reversibility [28], which makes it possible to get to states which cannot be reached by forward reactions alone. Such sequences of forward and reverse reaction steps are important as they lead to new chemical structures and new reactions, which would not be possible without out-of-causal-order reversibility [28]. A typical example is a catalytic reaction: a catalyst C enables compounds A and B to combine, a combination that would not normally happen or be very unlikely without the presence of C. Initially, catalyst C binds with B resulting in a compound BC. Then A combines with BC creating ABC. Finally, with its job done, C breaks away from ABC, leaving A and B bonded. This sequence of reactions can be written as follows:

$$A + B + C \rightarrow A + BC \rightarrow ABC \rightarrow AB + C$$

This is a typical example of out-of-causal order reversibility since the bond between B and C is undone before its effect, namely the bond from A to B (which is not undone at all). The modelling of such reactions is the focus of this chapter. For further motivation, formal definitions and more illustrating examples of the various types of reversibility we refer the reader to [8,19,25,28].

1.1 Contribution

This chapter presents and compares three formalisms, the Calculus of Covalent Bonding (CCB) [15,16], the Bonding Calculus [1], and Reversing Petri Nets [23], that have been developed during COST Action IC1405. These models are variations of existing formalisms and set out to study reversible computation by allowing systems to reverse at any time leading to previously visited states or even new states without the need of additional forward actions. The contribution of this chapter is a comparative overview of the three formalisms, a discussion of their expressiveness, and a demonstration of their use on a common case study, namely the autoprotolysis of water reaction.

Our case study was selected to be non-trivial, of manageable size, and to allow us to exhibit the crucial features of the formalisms. It is a chemical reaction that involves small molecules, so it is different from biological reactions that involve proteins and other macromolecules. New modelling techniques may be needed in order to capture fully reversible behaviour of biological systems, however, in this chapter we concentrate on chemical reactions, a domain that offers interesting examples of out-of-causal-order reversibility.

The discussed formalisms enable us to model the intermediate steps of chemical reactions where some bonds are only "helping" to achieve the overall aim of the reaction: specifically, they are only formed to be broken before the end of the reactions. Thus, the allowed level of detail makes a more accurate depiction of the reversibility possible, and allows a more thorough understanding of the underlying reaction mechanisms compared to higher-level models.

1.2 Related Work

Process calculi, originally designed for the modelling of sequential and concurrent computation, have been applied to biochemical and biological systems. The main instances are the π-calculus [34], BioAmbients [33], the stochastic π-calculus [30], beta binders [31] and bioPEPA [6]. Another way to model biochemical reactions is with rule-based formalisms such as BIOCHAM [10], the κ-calculus [7], and the BioNetGen Language (BNGL) [9]. The formalisms κ and BNGL can be used to model interactions between proteins, while this is not possible in BIOCHAM. BNGL allows the use of molecule sites having the same name, which is not allowed in the κ-calculus.

Most of the formalisms mentioned above do not explicitly represent reversibility. If an action is the reverse of another action performed before, there is no explicit knowledge of that in the model. Reversibility was added explicitly to process calculi in RCCS [8], CCSK [25], and reversible π [17,18]. CCSK and RCCS are based on the Calculus of Communicating Systems (CCS) [21]. They extend CCS by keeping track of past actions and enabling an undo of those. So a reverse action is the reverse execution of a forward action. These calculi support backtracking and causally-consistent reversibility. Out-of-causal-order reversibility was first addressed in CCSK extended with controller processes [28], and in the context of reversible event structures [26,27,37]. CCB [16] allows all types of reversibility in the context of chemical reactions and in other settings.

Petri nets (PNs) [35] are another formalism that has been widely used to model and reason about a wide range of applications featuring concurrency and distribution. They are a graphical language associated with a rich mathematical theory and supported by a variety of tools. Their use in systems biology dates back to [12,32]. Since then, they have been employed for the modelling, analysis, and simulation of biochemical reactions in metabolic pathways, gene expression, signal transduction, and neural processes [2,4,5]. Indeed, PNs seem to be a natural framework for representing biochemical systems as they constitute a set of interdependent transitions/reactions which consume and produce resources, and are represented graphically in a similar fashion to the systems in question. Several specialised Petri net classes, such as qualitative, stochastic, continuous, or hybrid Petri nets and their coloured counterparts, have been used to describe different biochemical systems [13,20,22,29,38].

Even though classical PNs and their extensions have been extensively used to model biochemical systems, they cannot directly model reversibility. Specifically, when modelling reversible reactions in these formalisms it is required to employ mechanisms involving two distinct transitions, one for the forward and one for the reverse version of a reaction. This may result in expanded models and less natural and/or less accurate models of reversible behaviour. It is also in contrast to the notion of reversible computation, where the intention is not to return to a state via arbitrary execution but to reverse the effect of already executed transitions. For this reason, the formalism of reversing Petri nets [23] has been proposed to allow systems to reverse already executed transitions leading to previously visited states or even new ones without the need of additional forward actions.

Reversing Petri nets have also been extended with a mechanism for controlling transition reversal by associating transitions with conditions [24].

1.3 Paper Organisation

In the next section, we introduce the autoprotolysis of water reaction, which will be modelled using our three formalisms. This is followed by a section introducing the formalisms, their syntax and, informally, their operational semantics. We also give three models of the autoprotolysis of water using the formalisms. In Sect. 4, we compare the formalisms and the models of our example reaction, and we also briefly discuss software support for the three formalisms. Finally, Sect. 5 concludes the paper.

2 Autoprotolysis of Water

We consider a chemical reaction that transfers a hydrogen atom between two water molecules. This reaction is known as the *autoprotolysis of water* and is shown in Fig. 1. There, O indicates an oxygen atom and H a hydrogen atom. The lines indicate bonds. Positive and negative charges on atoms are shown by $\oplus$ and $\ominus$ respectively. The meaning of the curved arrows and the dots will be explained in the next paragraphs. The reaction is reversible and it takes place at a relatively low rate, making pure water slightly conductive. We have chosen this reaction as our example reaction, since it is non-trivial but manageable, and has some interesting aspects to be represented.

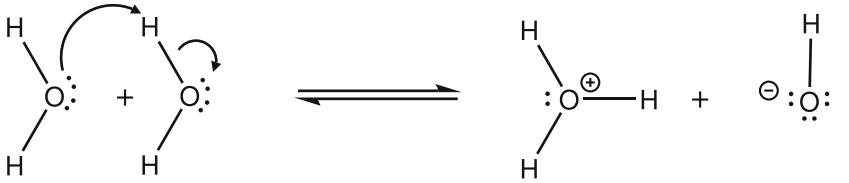


Fig. 1. Autoprotolysis of water.

To model the reaction we need to understand why it takes place and what causes it. The main reason is that the oxygen in the water molecule is *nucleophilic*, meaning it has the tendency to bond to another atomic nucleus, which would serve as an *electrophile*. This is because oxygen has a high electronegativity, therefore it attracts electrons and has an abundance of electrons around it. The electrons around the atomic nucleus are arranged on electron shells, where only those in the outer shell participate in bonding. Oxygen has four electrons in its outer shell, which are not involved in the initial bonding with hydrogen atoms. These electrons form two *lone pairs* of two electrons each, which can form new bonds (lone pairs are shown in Fig. 1 by pairs of dots). All

this makes oxygen nucleophilic: it tends to connect to other atomic nuclei by forming bonds from its lone pairs. Since oxygen attracts electrons, the hydrogen atoms in water have a positive partial charge and oxygen has a negative partial charge.

The reaction starts when an oxygen in one water molecule is attracted by a hydrogen in another water molecule due to their opposite charges. This results in a *hydrogen bond*. This bond is formed out of the electrons of one of the lone pairs of the oxygen. The large curved arrow in Fig. 1 indicates the movements of the electrons. Since a hydrogen atom cannot have more than one bond, the creation of a new bond is compensated by breaking the existing hydrogen-oxygen bond (indicated by the small curved arrow). When this happens, the two electrons, which formed the original hydrogen-oxygen bond, remain with the oxygen. Since a hydrogen contains one electron and one proton, it is only the proton that is transferred, so the process can be called a proton transfer as well as a hydrogen transfer. The forming of the new bond and the breaking of the old bond are *concerted*, meaning that they happen together without a stable intermediate configuration. As a result we have reached the state where one oxygen atom has three bonds to hydrogen atoms and is positively charged, represented on the right side of the reaction in Fig. 1. This molecule is called *hydronium* and is written as H_3O^+. The other oxygen atom bonds to only one hydrogen and is negatively charged, having an electron in surplus. This molecule is called a *hydroxide* and is written as OH^-.

Note that the reaction is reversible: the oxygen that lost a hydrogen can pull back one of the hydrogens from the other molecule, the H_3O^+ molecule. This is the case since the negatively charged oxygen is a strong nucleophile and the hydrogens in the H_3O^+ molecule are all positively charged. Thus, any of the hydrogens can be removed, making both oxygens formally uncharged, and restoring the two water molecules. In Fig. 1 the curved arrows are given for the reaction going from left to right. Since the reaction is reversible (indicated by the double arrow) there are corresponding electron movements when going from right to left. These are not given in line with usual conventions, but can be inferred.

In this simple reaction, the forward and the reverse step consist of two steps each. The breaking of the old and the forming of the new bond occur simultaneously. This means that there is no strict causality of actions, since none of them can be called the cause of the overall reaction. Furthermore, the reverse step can be done with a different atom to the one used during the forward step because each of the molecules are in a sense identical and in practice there does not exist a single "reverse" path corresponding to a forward one.

It should be noted that there are two types of bonding modelled here. Firstly, we have the initial bonds where two atoms contribute an electron each. Secondly, the *dative* or *coordinate bonds* are formed where both electrons come from one atom (an oxygen in this case). Both are *covalent bonds*, and once formed they cannot be distinguished. Specifically, in the oxygen with three bonds all bonds are the same and no distinction can be made. If one of the bonds is broken by

a deprotonation (as in the autoprotolysis of water) the two electrons are left behind and they form a lone pair. If the broken bond was not previously formed as a dative bond, the electrons changed their "rôle". This explains why any proton can be transferred in the reverse reaction and not just the one that was involved in the forward path.

3 Formalisms for Reversible Chemical Reactions

3.1 Calculus of Covalent Bonding

In this subsection we introduce the Calculus of Covalent Bonding (CCB) [16], concentrating on the new general prefixing operator $(s; b).P$ which, together with a generalised composition operator, produces pairs of *concerted* actions. Then we present a CCB model of the autoprotolysis of water.

Definition of CCB. We recall the definition of CCB, presenting only the main ideas. More details can be found in [15,16]. First, we introduce some preliminary notions and notations.

Let $\mathcal{A}$ be the set of (forward) action labels, ranged over by a, b, c, d, e, f. We partition $\mathcal{A}$ into the set of *strong actions*, written as $\mathcal{SA}$, and the set of *weak actions*, written as $\mathcal{WA}$. Reverse (or past) action labels are members of $\underline{\mathcal{A}}$, with typical members $\underline{a}, \underline{b}, \underline{c}, \underline{d}, \underline{e}, \underline{f}$, and represent undoing of actions. The set $\mathcal{P}(\mathcal{A} \cup \underline{\mathcal{A}})$ is ranged over by L.

Let $\mathcal{K}$ be an infinite set of *communication keys* (or *keys* for short) [25], ranged over by k, l, m, n. The Cartesian product $\mathcal{A} \times \mathcal{K}$, denoted by $\mathcal{AK}$, represents past actions, which are written as $a[k]$ for $a \in \mathcal{A}$ and $k \in \mathcal{K}$. Correspondingly, we have the set $\underline{\mathcal{AK}}$ that represents undoing of past actions. We use α, β to identify actions which are either from $\mathcal{A}$ or $\mathcal{AK}$. It would be useful to consider sequences of actions or past actions, namely the elements of $(\mathcal{A} \cup \mathcal{AK})^*$, which are ranged over by s, s' and sequences of purely past actions, namely the elements of $\mathcal{AK}^*$, which are ranged over by t, t'. The empty sequence is denoted by ϵ. We use the notation "α, s" and "s, s'" to denote a concatenation of elements, which can be strings or single actions.

We shall also use two sets of auxiliary action labels, namely the set $(\mathcal{A}) = \{(a) \mid a \in \mathcal{A}\}$, and its product with the set of keys, namely $(\mathcal{A})\mathcal{K}$. These labels will be used in the auxiliary rules when defining the semantics of CCB. They denote the execution of a weak action, which makes it possible in the SOS rules to force breaking of a bond for those actions only.

The syntax of CCB is given below where P is a process term:

$$P ::= S \mid S \stackrel{def}{=} P \mid (s; b).P \mid P|Q \mid P \backslash L$$

The set of process identifiers (constants) $\mathcal{PI}$ contains typical elements S and T. Each process identifier S has a defining equation $S \stackrel{def}{=} P$ where P contains

only forward actions (and no past actions). There is also a special identifier $\mathbf{0}$, denoting the deadlocked process, which has no defining equation. For restrictions $L \subseteq \mathcal{A}$ holds.

We have a general prefixing operator $(s; b).P$, where s is a non-empty sequence of actions or past actions. This operator extends the prefixing operator in [28]. The action b is a weak action and it can be omitted, in which case the prefixing is written as $(s).P$ and is called the *simple prefix*. The simple prefix (which is still a sequence) is the prefixing operator in [28]. Exactly one of the actions in s in $(s).P$ may be a weak action from $\mathcal{WA}$. A weak action in s is only allowed for the simple prefix, in the $(s; b)$ operator b is the only allowed weak action. If s is a sequence that contains a single action, then the action is a strong action and the operator is the prefixing operator of CCS [21]. We omit trailing $\mathbf{0}$s so, for example, $(s).\mathbf{0}$ is written as (s). The new feature of the operator $(s; b).P$ is the execution of the weak action b, which can happen only after all the actions in s have taken place. Performing b then forces undoing one of the past actions in s (by the concert rule in Fig. 4). If a $(s; b)$ operator is followed by another sequence of actions, where all actions in s have already taken place, then there is a non-deterministic choice of either doing b or progressing to the next sequence of actions (see act1 and act2).

$P \mid Q$ represents two systems P and Q which can perform actions or reverse actions on their own, or which can interact with each other according to a communication function γ. As in the calculus ACP [11], the communication function is a partial function $\gamma : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ which is commutative and associative. The function γ is used in the operational semantics to define when two processes can interact. Processes P and Q in $P \mid Q$ can also perform a pair of concerted actions, which is the new feature of our calculus. We also have the ACP-like restriction operator $\backslash L$, where L is a set of labels. It prevents actions from taking place and, due to the synchronisation algebra used, it also blocks communication. If $\gamma(a, b) = c$ then $a.P$ and $b.Q$ cannot communicate in $(a.P \mid b.Q) \setminus c$.

The set Proc of *process terms* is ranged over by P, Q and R. In the setting of CCB these terms are simply called *processes*. We define the semantics of our calculus using SOS rules (Figs. 2, 3, 4) and rewrite rules (Fig. 5).

We use some predicates and functions, which are formally defined in [16]. Informally, a process P is standard, written $\mathsf{std}(P)$, if it contains no past actions (hence no keys). A key n is fresh in Q, written $\mathsf{fsh}[n](Q)$, if Q contains no past action with the key n. Function k returns the keys in a sequence of actions, whereas keys returns the keys in a process, and fn gives the actions of a process which could be executed.

The forward and reverse SOS rules for CCB are given in Figs. 2 and 3. Figure 4 contains the SOS rules that define the new concerted actions transitions. The rule concert defines when a pair of concerted actions takes place. This enables the linking of forming and breaking of bonds, and therefore a degree of control over the reversing of actions. The modelling in the next section will give examples of the application. Note that the concert rule uses *lookahead* [36]. Lookahead is a property of SOS rules, where a variable appears both on the right hand side and

S. Kuhn et al.

$$\textsf{act1}\ \frac{\textsf{std}(P) \quad \textsf{fsh}[k](s,s')}{(s,a,s';b).P \xrightarrow{a[k]} (s,a[k],s';b).P} \qquad \textsf{act2}\ \frac{P \xrightarrow{a[k]} P' \quad \textsf{fsh}[k](t)}{(t;b).P \xrightarrow{a[k]} (t;b).P'}$$

$$\textsf{par}\ \frac{P \xrightarrow{a[k]} P' \quad \textsf{fsh}[k](Q)}{P \mid Q \xrightarrow{a[k]} P' \mid Q} \qquad \textsf{com}\ \frac{P \xrightarrow{a[k]} P' \quad Q \xrightarrow{d[k]} Q'}{P \mid Q \xrightarrow{c[k]} P' \mid Q'}\ (*)$$

$$\textsf{res}\ \frac{P \xrightarrow{a[k]} P'}{P\backslash L \xrightarrow{a[k]} P'\backslash L}\ a \notin L \qquad \textsf{con}\ \frac{P \xrightarrow{a[k]} P'}{S \xrightarrow{a[k]} P'}\ S \stackrel{def}{=} P$$

Fig. 2. Forward SOS rules for CCB. The condition (*) is $\gamma(a,d) = c$, and $b \in \mathcal{WA}$. Recall that s is a sequence of actions and past actions and t is a sequence of purely past actions.

$$\textsf{rev act1}\ \frac{\textsf{std}(P)}{(s,a[k],s';b).P \xrightarrow{\underline{a}[k]} (s,a,s';b).P} \qquad \textsf{rev act2}\ \frac{P \xrightarrow{\underline{a}[k]} P'}{(t;b).P \xrightarrow{\underline{a}[k]} (t;b).P'}$$

$$\textsf{rev par}\ \frac{P \xrightarrow{\underline{a}[k]} P' \quad \textsf{fsh}[k](Q)}{P \mid Q \xrightarrow{\underline{a}[k]} P' \mid Q} \qquad \textsf{rev com}\ \frac{P \xrightarrow{\underline{a}[k]} P' \quad Q \xrightarrow{\underline{d}[k]} Q'}{P \mid Q \xrightarrow{\underline{c}[k]} P' \mid Q'}\ (*)$$

$$\textsf{rev res}\ \frac{P \xrightarrow{\underline{a}[k]} P'}{P\backslash L \xrightarrow{\underline{a}[k]} P'\backslash L}\ \underline{a} \notin L \qquad \textsf{rev con}\ \frac{P \xrightarrow{\underline{a}[k]} P'}{P \xrightarrow{\underline{a}[k]} S}\ S \stackrel{def}{=} P'$$

Fig. 3. Reverse SOS rules for CCB. The condition (*) is $\gamma(a,d) = c$, and $b \in \mathcal{WA}$.

on the left hand side of a transition in the premises. for example P' and Q' in concert. The rule concert par requires that k is fresh in Q, correspondingly as in par. Moreover, we need to ensure that when we reverse h with the key l in P we do not leave out any actions with the key l in Q which make up a multiaction communication with the key l. Hence, we also include the premise $\textsf{fsh}[l](Q)$ in concert par. The rule concert act requires, correspondingly as act, that k is fresh in t. Our operational semantics guarantees that if a standard process evolves to $(t;b).P$, for some P, and P reverses an action with the key l, then l is fresh in t. Hence, we do not include $\textsf{fsh}[l](t)$ in the premises of concert act. Overall, the transitions in Figs. 2, 3 and 4 are labelled with $a[k] \in \mathcal{AK}$, or with $\underline{c}[l] \in \underline{\mathcal{A}}\mathcal{K}$, or with concerted actions $(a[k], \underline{c}[l])$.

Next, we recall the main new rewrite rules for a reduction relation for CCB in Fig. 5. All the rules can be found in [15,16] but here we only give rules for *promotion* of actions. These are prom, move-r, and move-l which promote weak bonds (here b) to strong bonds (here a). The rule prom applies to the full version of our prefix operator (with the; construct), and move-r and move-l apply only to the simple prefix. These three rules are here to model what happens in chemical systems: a bond on a weak action is temporary and as soon as there is a strong action that can accommodate that bond (as the result of concerted

$$\textsf{aux1}\ \frac{\textsf{std}(P) \quad \textsf{fsh}[k](t)}{(t;b).P \xrightarrow{(b)[k]} (t;b[k]).P} \qquad \textsf{aux2}\ \frac{P \xrightarrow{(b)[k]} P' \quad \textsf{fsh}[k](t)}{(t;b').P \xrightarrow{(b)[k]} (t;b').P'}$$

$$\textsf{concert}\ \frac{P \xrightarrow{(b)[k]} P' \quad P' \xrightarrow{\underline{a}[l]} P'' \qquad Q \xrightarrow{\alpha[k]} Q' \quad Q' \xrightarrow{\underline{d}[l]} Q''}{P \mid Q \xrightarrow{\{e[k],\underline{f}[l]\}} P'' \mid Q''}(*)$$

$$\textsf{concert act}\ \frac{P \xrightarrow{\{a[k],\underline{h}[l]\}} P' \quad \textsf{fsh}[k](t)}{(t;b).P \xrightarrow{\{a[k],\underline{h}[l]\}} (t;b).P'}$$

$$\textsf{concert par}\ \frac{P \xrightarrow{\{a[k],\underline{h}[l]\}} P' \quad \textsf{fsh}[k](Q) \quad \textsf{fsh}[l](Q)}{P \mid Q \xrightarrow{\{a[k],\underline{h}[l]\}} P' \mid Q}$$

$$\textsf{concert res}\ \frac{P \xrightarrow{\{a[k],\underline{h}[l]\}} P'}{P\backslash L \xrightarrow{\{a[k],\underline{h}[l]\}} P'\backslash L}(**)$$

Fig. 4. SOS rules for concerted actions in CCB. The condition (*) is 1. $\alpha = c \vee \alpha = (c)$ and $\exists c \in \mathcal{A} | \gamma(b,c) = e$, and 2. $\gamma(a,d) = f$. The condition (**) is $a, \underline{h} \notin L \cup (L)$. Recall that $t \in \mathcal{AK}^*$, and $b \in \mathcal{WA}$.

$$\begin{aligned}
&\textsf{prom}: && (s,a,s';b[k]).P \Rightarrow (s,a[k],s';b).P && \text{if } a \in \mathcal{SA}, b \in \mathcal{WA}\\
&\textsf{move-r}: && (s,a,s',b[k],s'').P \Rightarrow (s,a[k],s',b,s'').P && \text{if } a \in \mathcal{SA}, b \in \mathcal{WA}\\
&\textsf{move-l}: && (s,b[k],s',a,s'').P \Rightarrow (s,b,s',a[k],s'').P && \text{if } a \in \mathcal{SA}, b \in \mathcal{WA}
\end{aligned}$$

Fig. 5. New reduction rules for CCB. Sequences s, s', s'' are members of $(\mathcal{A} \cup \mathcal{AK})^*$.

actions) the bond establishes itself on the strong action thus releasing the weak action. In order to align the use of these three rules to what happens in chemical reactions, we insist that they are used as soon as they becomes applicable, a formal definition is given in [15,16].

We shall call henceforth the transitions derived by the forward SOS rules the *forward transitions* and, the transitions derived by the reverse SOS rules the *reverse transitions*. Correspondingly, there are the *concerted (action)* transitions.

The Autoprotolysis of Water in CCB. When modelling the autoprotolysis of water in CCB, we shall model the hydrogen and oxygen atoms as processes H and O as follows, where h, o are actions representing the bonding capabilities of the atoms and n, p representing negative and positive charges, respectively. H' and O' are process constants, and p and n are weak actions.

$$H \stackrel{def}{=} (h;p).H' \quad O \stackrel{def}{=} (o,o,n).O'$$

The synchronisation function γ is as follows:

$$\gamma(h,o) = ho \qquad \gamma(n,p) = np \qquad \gamma(n,h) = nh$$

Each water molecule is a structure consisting of two hydrogen atoms and one oxygen atom which are bonded appropriately. We shall use subscripts to distinguish the individual copies of atoms and actions; for example H_1 is a specific copy of hydrogen defined by $(h_1; p).H_1'$, similarly for O_1 defined as $(o_1, o_2, n).O_1'$. The atoms are composed with the parallel composition operator "|" using the communication keys (which are natural numbers) to combine actions into bonds. So a water molecule is modelled by the following process, where the key 1 shows that h_1 of H_1 has bonded with o_1 of O_1 (correspondingly for key 2). The restriction $\backslash\{h_1, h_2, o_1, o_2\}$ ensures that these actions cannot happen on their own, but only together with their partners, forming a bond.

$$((h_1[1]; p).H_1' \mid (h_2[2]; p).H_2' \mid (o_1[1], o_2[2], n).O_1') \setminus \{h_1, h_2, o_1, o_2\}$$

The system of two water molecules in Fig. 1 is represented by the parallel composition of two water processes, where the restriction $\backslash\{n, p\}$ represses actions n, p from taking place separately by forcing them to combine into bonds (according to γ).

$$\begin{aligned}&(((h_1[1]; p).H_1' \mid (h_2[2]; p).H_2' \mid (o_1[1], o_2[2], n).O_1') \setminus \{h_1, h_2, o_1, o_2\} \mid \\ &((h_3[3]; p).H_3' \mid (h_4[4]; p).H_4' \mid (o_3[3], o_4[4], n).O_2') \setminus \{h_3, h_4, o_3, o_4\}) \setminus \{n, p\}\end{aligned}$$

Following a general principle in process calculi in the style of CCB we can move the restrictions to the outside. The rule used can be written as $(P \mid Q) \setminus L = P \setminus L \mid Q$ if the actions of L are not used in Q. Applying this gives us a water molecule modelled as follows:

$$\begin{aligned}&((h_1[1]; p).H_1' \mid (h_2[2]; p).H_2' \mid (o_1[1], o_2[2], n).O_1') \mid (h_3[3]; p).H_3' \mid \\ &(h_4[4]; p).H_4' \mid (o_3[3], o_4[4], n).O_2')) \setminus \{h_1, h_2, o_1, o_2\} \setminus \{h_3, h_4, o_3, o_4\} \setminus \{n, p\}\end{aligned}$$

Note the $\underline{h_i}$, $\underline{o_j}$, and $\underline{n}$ are not restricted: this allows us to break bonds via concerted actions involving these actions. We will see an example of this shortly. We now leave out the restrictions to improve readability.

Actions n in O_1 and p in H_3 combine (we use the new key 5), representing a transfer of a proton from one atom of oxygen (O_2 in our model) to another one (O_1 in our model). As a hydrogen atom consists of a proton and an electron, and the electron stays in such a transfer, it can either be called a proton transfer or the transfer of a (positively charged) hydrogen atom. We perform the transfer of H_3 from O_2 to O_1. The creation of the bond with key 5 from O_1 to H_3 forces a break of the bond with key 3 (between h_3 and o_3) due to the property of the operator $(s; b).P$ discussed earlier. These two reactions happen almost simultaneously so we represent them as a pair of *concerted actions*.

$$\begin{aligned}&(h_1[1]; p).H_1' \mid (h_2[2]; p).H_2' \mid (o_1[1], o_2[2], n).O_1' \mid (h_3[3]; p).H_3' \\ &\mid (h_4[4]; p).H_4' \mid (o_3[3], o_4[4], n).O_2') \\ &\xrightarrow{\{np[5], \underline{h_3 o_3}[3]\}} \\ &((h_1[1]; p).H_1' \mid (h_2[2]; p).H_2' \mid (o_1[1], o_2[2], n\mathbf{[5]}).O_1' \mid (\boldsymbol{h_3}; p\mathbf{[5]}).H_3' \\ &\mid (h_4[4]; p).H_4' \mid (\boldsymbol{o_3}, o_4[4], n).O_2'\end{aligned}$$

We have now arrived at the state on the right hand side in Fig. 1. There are weak bonds between n and p (denoted by key 5) and *strong* bonds between h_i and o_j for all appropriate i, j. Since H_3 is weakly bonded to O_1 and its strong capability h_3 has become available, the bond 5 gets promoted to the stronger bond, releasing the capability p of H_3. We represent this change as a rewrite and we obtain the following process:

$$
\begin{array}{l}
((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], n\mathbf{[5]}).O_1' \mid (\boldsymbol{h_3};p\mathbf{[5]}).H_3' \\
\mid (h_4[4];p).H_4' \mid (\boldsymbol{o_3}, o_4[4], n).O_2' \\
\Rightarrow \\
((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], n[5]).O_1') \mid (h_3\mathbf{[5]};p).H_3') \\
\mid (h_4[4];p).H_4' \mid (o_3, o_4[4], n).O_2'
\end{array}
$$

Note that we wrote h_3, o_3 and the key 5, the actions and keys affected by the promotion, in bold font to improve readability. We shall do correspondingly below.

Oxygen O_1 is still blocked, which represents it being fully bonded (and positively charged). Oxygen O_2 has a free n capability and can remove any of the hydrogens from O_1. As a result the process can reverse to its original state.

We show this by again transferring H_3. We then execute promotion again:

$$
\begin{array}{l}
(((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], n[5]).O_1') \mid (h_3[5];p).H_3') \\
\mid (h_4[4];p).H_4' \mid (o_3, o_4[4], n).O_2' \\
\xrightarrow{\{np[3], \underline{n}h_3[5]\}} \\
(((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], \boldsymbol{n}).O_1') \mid (\boldsymbol{h_3};\boldsymbol{p}\mathbf{[3]}).H_3') \\
\mid (h_4[4];p).H_4' \mid (o_3, o_4[4], \boldsymbol{n}\mathbf{[3]}).O_2' \\
\Rightarrow \\
(((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], \boldsymbol{n}).O_1') \mid (\boldsymbol{h_3}\mathbf{[3]};\boldsymbol{p}).H_3') \\
\mid (h_4[4];p).H_4' \mid (\boldsymbol{o_3}\mathbf{[3]}, o_4[4], \boldsymbol{n}).O_2'
\end{array}
$$

This corresponds to the original process. Putting back restrictions we obtain

$$
\begin{array}{l}
((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], n).O_1' \mid (h_3[3];p).H_3' \\
\mid (h_4[4];p).H_4' \mid (o_3[3], o_4[4], n).O_2') \setminus \{h_1, h_2, o_1, o_2\} \setminus \{h_3, h_4, o_3, o_4\} \setminus \{n, p\}
\end{array}
$$

and then if we apply the movement of restrictions in reverse we get

$$
\begin{array}{l}
(((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], n).O_1') \setminus \{h_1, h_2, o_1, o_2\} \mid \\
((h_3[3];p).H_3' \mid (h_4[4];p).H_4' \mid (o_3[3], o_4[4], n).O_2') \setminus \{h_3, h_4, o_3, o_4\}) \setminus \{n, p\}
\end{array}
$$

3.2 Bonding Calculus

In this subsection we recall briefly the Bonding Calculus [1], and illustrate its expressiveness by modelling the autoprotolysis of water.

Definition of the Bonding Calculus. The abstraction "processes as interactions" from process calculi is used in the Bonding Calculus, but processes are not able to communicate values in order to interact. Just like in the BNGL [9], the Bonding Calculus allows the use of molecule sites having the same name, while this is not possible in the κ-calculus. While the κ-calculus describes molecules as a set of sites and uses rules to manipulate these sites between two or more molecules, in the Bonding Calculus a molecule is described by the sequence of operations it can perform on its sites (including also non-deterministic choices), regardless of the form of the other molecules. This allows to use the compositionality of the process calculus.

The syntax of the Bonding Calculus syntax is presented in Fig. 6. Let us consider the set $\mathbb{N}$ of natural numbers, the set $\mathcal{N} = \{x, x^+, x^-, \dots\}$ of bond names, the set $\mathcal{M} = \{a, b, \dots\}$ of molecules and the set $\mathcal{P} = \{P, Q, \dots\}$ of processes. A multiset over $\mathcal{N}$ is defined as a partial function $N : \mathcal{N} \to \mathbb{N}$. In the Bonding Calculus each molecule has a unique name, and the bond x between two molecules a and b is denoted by $\{a -^x b\}$.

Bonds	L	$::=$	$\emptyset$	(empty)
		$\mid$	$\{a -^x b\}$	(bond)
		$\mid$	$L \uplus L$	(union)
Actions	α	$::=$	$\overline{x}(b)$	(bond)
		$\mid$	$\underline{x}(b)$	(unbond)
Processes	P	$::=$	$\mathbf{0}$	(empty)
		$\mid$	$\alpha.P$	(action prefix)
		$\mid$	$P + Q$	(choice)
		$\mid$	$P \mid Q$	(parallel)
		$\mid$	**if** $a \doteqdot^L b$ **then** P **else** Q	(testing)
		$\mid$	$A(b_1, \dots, b_n)$	(identifier)
Definition	$A(a_1, \dots, a_n)$	$::=$	P	(recursion)
System	S	$::=$	$P \mid\mid L$.	

Fig. 6. Syntax of the Bonding Calculus

A bond prefix $\overline{x}(b)$ is used to indicate the availability of a molecule with name b to create a new bond with name x, while an unbond prefix $\underline{x}(b)$ indicates the availability of b to destroy an existing bond x. Creating or breaking a bond leads to an update of the global bond memory L. As several similar bonds can exist between the same molecules, L is actually a multiset of bonds.

The process $\mathbf{0}$ denotes inactivity. The availability to perform an action α, and then to continue the execution as process P is denoted by the process $\alpha.P$. The process $P + Q$ offers a choice between the processes P and Q, while the process $P \mid Q$ allows the execution of processes P and Q in parallel, with possible interactions between them by using appropriate actions.

As we work with bonds, we use the function $\div\!\!\div$: $\mathcal{M} \times \mathbb{N}^{\mathcal{N}} \times \mathcal{M} \to Bool$ to check whether between two molecules there exist certain bonds. For example, $a \div\!\!\div^N b$ checks for the existence of all bonds in N between the molecules a and b; it returns true when such bonds exist, and false otherwise. When we consider $N = \emptyset$, then $a \div\!\!\div^\emptyset b$ checks if at least a bond exists between the two molecules. When $b = \varepsilon$, then $a \div\!\!\div^N \varepsilon$ checks if a has all of bonds from N, regardless of the molecules he has them with. The Boolean result $a \div\!\!\div^N b$ used in the testing process is defined formally as:

$$a \div\!\!\div^N b = \begin{cases} (\biguplus_{x \in N} \{a -^x b\}) \in L & N \neq \emptyset \text{ and } a \neq b \neq \varepsilon \\ (\biguplus_{x \in \mathcal{N}} \{a -^x b\}) \cap L \neq \emptyset & N = \emptyset \text{ and } a \neq b \neq \varepsilon \\ \bigwedge_{x \in N} (|L|_{a,x} = |N|_x) & N \neq \emptyset \text{ and } a \neq \varepsilon \text{ and } b = \varepsilon \\ \textit{undefined} & \textit{otherwise}, \end{cases}$$

where $|L|_{a,x}$ is the number of bonds containing the molecule a and bond name x that appear in the multiset L, while $|N|_x$ is the number of occurrences of x in N.

Depending on the truth value of $a \div\!\!\div^N b$, the process **if** $a \div\!\!\div^N b$ **then** P **else** Q executes either P or Q. An identifier $A(b_1, \ldots, b_n)$ is used to provide recursion by creating new instances of processes defined as $A(a_1, \ldots, a_n) = P$, where $a_i \neq a_j$ for all $i \neq j \in \{1, \ldots, n\}$; the new process is defined as $A(b_1, \ldots, b_n) = P\{b_1/a_1, \ldots, b_n/a_n\}$, where $\{b_i/a_i\}$ denotes the replacement of variable a_i by value b_i. A system S is given as a composition of a process P and the multiset of bonds L, written as $P \parallel L$.

The structural congruence relation $\equiv$ is the least congruence such that $(\mathcal{P}, +, \mathbf{0})$ and $(\mathcal{P}, |, \mathbf{0})$ are commutative monoids and the unfolding law $A(b_1, \ldots, b_n) \equiv P\{b_1/a_1, \ldots, b_n/a_n\}$ holds whenever $A(a_1, \ldots, a_n) = P$.

The calculus presented in [1] was intended to model the creation and breaking of covalent bonds. In order to be able to model both covalent and hydrogen bonds, we apply a minor update to the operational semantics in [1] because we need two instances of the rules used to create and to break bonds. The only difference between the two instances of the same rule is given by the names of the bonds appearing in the interacting processes, and by the fact that a bond cannot be created using the names x^+ and x^- if other bonds exist between the same molecules; more details about this restriction are given in the example below.

The operational semantics of the Bonding Calculus is given in Fig. 7. The rules (CREATE1) and (CREATE2) describe the creation of a new bond $\{a -^x b\}$, while the rules (REMOVE1) and (REMOVE2) describe the breaking of a bond $\{a -^x b\}$. If there exist two bonds $\{a -^x b\}$ in L, then any of these bonds is broken. The rule (PAR) is used to compose processes in parallel, while the rules (TRUE) and (FALSE) choose one of the branches of the testing process based on the result of the checking. The rule (IDE) describes the recursion, while the (STRUCT) rule indicates the fact that we reason up to the structural congruence.

$$\text{(CREATE1)}\ \frac{P = \overline{x}(b).P' + P'' \quad Q = \overline{x}(a).Q' + Q''}{(P \mid Q) \mid\mid L \rightarrow (P' \mid Q') \mid\mid L \uplus \{a -^{x} b\}}$$

$$\text{(CREATE2)}\ \frac{P = \overline{x^{+}}(b).P' + P'' \quad Q = \overline{x^{-}}(a).Q' + Q'' \quad a \Rightarrow^{\emptyset} b \ is\ false\ w.r.t.\ L}{(P \mid Q) \mid\mid L \rightarrow (P' \mid Q') \mid\mid L \uplus \{a -^{x} b\}}$$

$$\text{(REMOVE1)}\ \frac{P = \underline{x}(b).P' + P'' \quad Q = \underline{x}(a).Q' + Q'' \quad a \Rightarrow^{x} b \ is\ true\ w.r.t.\ L}{(P \mid Q) \mid\mid L \rightarrow (P' \mid Q') \mid\mid L \backslash \{a -^{x} b\}}$$

$$\text{(REMOVE2)}\ \frac{P = \underline{x^{+}}(b).P' + P'' \quad Q = \underline{x^{-}}(a).Q' + Q'' \quad a \Rightarrow^{N} b \ is\ true\ w.r.t.\ L}{(P \mid Q) \mid\mid L \rightarrow (P' \mid Q') \mid\mid L \backslash \{a -^{x} b\}}$$

$$\text{(PAR)}\ \frac{P \mid\mid L \rightarrow P' \mid\mid L}{(P \mid Q) \mid\mid L \rightarrow (P' \mid Q) \mid\mid L}$$

$$\text{(TRUE)}\ \frac{a \Rightarrow^{N} b \ is\ true\ w.r.t.\ L}{(\textbf{if}\ a \Rightarrow^{N} b\ \textbf{then}\ P\ \textbf{else}\ Q) \mid\mid L \rightarrow P \mid\mid L}$$

$$\text{(FALSE)}\ \frac{a \Rightarrow^{N} b \ is\ false\ w.r.t.\ L}{(\textbf{if}\ a \Rightarrow^{N} b\ \textbf{then}\ P\ \textbf{else}\ Q) \mid\mid L \rightarrow Q \mid\mid L}$$

$$\text{(IDE)}\ \frac{P\{b_1/a_1, \ldots, b_n/a_n\} \rightarrow P'}{A(b_1, \ldots, b_n) \rightarrow P'} \quad \text{if} \quad A(a_1, \ldots, a_n) = P$$

$$\text{(STRUCT)}\ \frac{S_1 \rightarrow S_1' \quad S_1 \equiv S_2 \quad S_2 \rightarrow S_2'}{S_1' \rightarrow S_2'}$$

Fig. 7. Operational Semantics of the Bonding Calculus.

The Autoprotolysis of Water in the Bonding Calculus. We use two types of bond names, namely c and h, to stand for the covalent and hydrogen bonds, respectively. Using our calculus, the system composed of two molecules of water is described by:

$$\begin{aligned} &MolOxy_2(O_1) \mid MolHy_1(H_1) \mid MolHy_1(H_2) \\ &\mid MolOxy_2(O_2) \mid MolHy_1(H_3) \mid MolHy_1(H_4) \\ &\mid\mid \{O_1 -^{c} H_1, O_1 -^{c} H_2, O_2 -^{c} H_3, O_2 -^{c} H_4\} \end{aligned}$$

where the molecules are those of hydrogen and oxygen that are described below:

$$\begin{aligned} &MolHy_0(H_i) = \overline{c}(H_i).MolHy_1(H_i) \\ &MolHy_1(H_i) = \underline{c}(H_i).MolHy_0(H_i) + \overline{h^{+}}(H_i).MolHy_2(H_i); \\ &MolHy_2(H_i) = \underline{c}(H_i).\overline{c}(H_i).\underline{h^{+}}(H_i).MolHy_1(H_i). \\ &MolOxy_0(O_i) = \overline{c}(O_i).MolOxy_1(O_i); \\ &MolOxy_1(O_i) = \underline{c}(O_i).MolOxy_0(O_i) + \overline{c}(O_i).MolOxy_2(O_i); \\ &MolOxy_2(O_i) = \underline{c}(O_i).MolOxy_1(O_i) + \overline{h^{-}}(O_i).MolOxy_3(O_i). \\ &MolOxy_3(O_i) = \underline{h^{-}}(O_i).MolOxy_2(O_i). \end{aligned}$$

Each molecule of water is a structure consisting of one molecule of oxygen and two molecules of hydrogen which are properly bonded. For example, the process $MolOxy_2(O_1) \mid MolHy_1(H_1) \mid MolHy(H_2)$ together with the bonds $\{O_1 -^c H_1, O_1 -^c H_2\}$ model one molecule of water. We use unique names for the molecules given as O_i (for oxygen) and H_i (for hydrogen), while the processes having the names $MolHy_i$ and $MolOxy_i$ identify processes modelling hydrogen and oxygen molecules with i bonds, respectively. For example, the process $MolOxy_1(O_i)$ can either create or break bonds, and this is why we use the operator + to describe such a (non-deterministic) choice.

Now we present the steps of one of the possible sequences of reactions modelling the autoprotolysis of water. The system of two molecules of water can be rewritten as follows (where we extend the definitions for the processes that will interact in the next step, and bold the actions to be executed):

$$\begin{gathered}\underline{c}(O_1).MolOxy_1(O_1) + \overline{\mathbf{h}^-}(\mathbf{O_1}).MolOxy_3(O_1) \mid MolHy_1(H_1) \mid MolHy_1(H_2) \\ \mid MolOxy_2(O_2) \mid MolHy_1(H_3) \mid \underline{c}(H_4).MolHy_0(H_4) + \overline{\mathbf{h}^+}(\mathbf{H_4}).MolHy_2(H_4) \\ \mid\mid \{O_1 -^c H_1, O_1 -^c H_2, O_2 -^c H_3, O_2 -^c H_4\}\end{gathered}$$

This leads to the next system, where we again bold the processes to be executed:

$$\begin{gathered}MolOxy_3(O_1) \mid MolHy_1(H_1) \mid MolHy_1(H_2) \\ \mid \underline{\mathbf{c}}(\mathbf{O_2}).MolOxy_1(O_2) + \overline{h^-}(O_2).MolOxy_3(O_1) \mid MolHy_1(H_3) \\ \mid \underline{\mathbf{c}}(\mathbf{H_4}).\overline{c}(H_4).\underline{h^+}(H_4).MolHy_1(H_4) \\ \mid\mid \{O_1 -^c H_1, O_1 -^c H_2, O_1 -^h H_4, O_2 -^c H_3, O_2 -^c H_4\}\end{gathered}$$

The creation of the hydrogen bond forces the break of the other bond in which the hydrogen molecule H_4 is involved. This leads to the following system containing the H_3O and HO molecules:

$$\begin{gathered}MolOxy_3(O_1) \mid MolHy_1(H_1) \mid MolHy_1(H_2) \\ \mid \underline{c}(O_2).MolOxy_0(O_2) + \overline{\mathbf{c}}(\mathbf{O_2}).MolOxy_2(O_2) \\ \mid MolHy_1(H_3) \mid \overline{\mathbf{c}}(\mathbf{H_4}).\underline{h^+}(H_4).MolHy_1(H_4) \\ \mid\mid \{O_1 -^c H_1, O_1 -^c H_2, O_1 -^h H_4, O_2 -^c H_3\}\end{gathered}$$

Since some bonds are weaker, the system is evolving to:

$$\begin{gathered}\underline{\mathbf{h}^-}(\mathbf{O_1}).MolOxy_2(O_1) \mid MolHy_1(H_1) \mid MolHy_1(H_2) \\ \mid MolOxy_2(O_2) \mid MolHy_1(H_3) \mid \underline{\mathbf{h}^+}(\mathbf{H_4}).MolHy_1(H_4) \\ \mid\mid \{O_1 -^c H_1, O_1 -^c H_2, O_1 -^h H_4, O_2 -^c H_3, O_2 -^c H_4\}\end{gathered}$$

followed by the breaking of the hydrogen bond $O_1 -^h H_4$:

$$\begin{gathered}MolOxy_2(O_1) \mid MolHy_1(H_1) \mid MolHy_1(H_2) \\ \mid MolOxy_2(O_2) \mid MolHy_1(H_3) \mid MolHy_1(H_4) \\ \mid\mid \{O_1 -^c H_1, O_1 -^c H_2, O_2 -^c H_3, O_2 -^c H_4\}\end{gathered}$$

The obtained system contains again two water molecules of water.

3.3 Reversing Petri Nets

In this subsection we present Reversing Petri Nets [23] (RPNs, pronounced as 'reversing Petri nets'), an extension of Petri nets developed for the modelling reversing computations, and we employ the formalism to model the autoprotolysis of water.

Definition of RPNs. We consider an extension of reversing Petri nets suitable for describing chemical reactions by allowing multiple tokens of the same type as well as the possibility for transitions to break bonds. Thus, a transition may simultaneously create and/or destroy bonds, and its reversal results in the opposite effect. Formally, a Reversing Petri net is defined as follows:

Definition 1. A *reversing Petri net* (RPN) is a tuple (P, T, A, A_V, B, F) where:

1. P is a finite set of *places* and T is a finite set of *transitions*.
2. A is a finite set of *base* or *token types* ranged over by $a, b, \ldots$. $\overline{A} = \{\overline{a} \mid a \in A\}$ contains a "negative" version for each token type. We assume that for any token type a there may exist a finite number of *token instances*. We write $a_1, \ldots,$ for instances of type a and A_I for the set of all token instances.
3. A_V is a finite set of *token variables*. We write $type(v)$ for the type of variable v and assume that $type(v) \in A$ for all $v \in A_V$.
4. $B \subseteq A \times A$ is a finite set of undirected *bond* types ranged over by $\beta, \gamma, \ldots$. We use the notation $a-b$ for a bond $(a, b) \in B$. $\overline{B} = \{\overline{\beta} \mid \beta \in B\}$ contains a "negative" version for each bond type. $B_I \subseteq A_I \times A_I$ is a finite set of *bond instances*, where we write β_i for elements of B.
5. $F : (P \times T \cup T \times P) \to \mathcal{P}(A_V \cup (A_V \times A_V) \cup \overline{A} \cup \overline{B})$ is a set of directed labelled *arcs*.

A reversing Petri net is built on the basis of a set of *tokens* or *bases*. These are organised in a set of token types A, where each token type is associated with a set of token instances. Token instances correspond to the basic entities that occur in a system and they may occur as stand-alone elements but as computation proceeds they may also merge together to form *bond instances*. Places and transitions have the standard meaning and are connected via directed arcs, which are labelled by a set of elements from $A_V \cup (A_V \times A_V) \cup \overline{A} \cup \overline{B}$. Intuitively, these labels express the requirements for a transition to fire when placed on arcs incoming the transition, and the effects of the transition when placed on the outgoing arcs. Graphically, a RPN is portrayed as a directed bipartite graph where token instances are indicated by $\bullet$, places by circles, transitions by boxes, and bond instances by lines between token instances.

Before we recall the semantics of RPNs we need to introduce some notation. Note that in what follows we omit the discussion of negative tokens and negative bonds as they are not relevant to our case study. We write $\circ t = \{x \in P \mid F(x,t) \neq \emptyset\}$ and $t\circ = \{x \in P \mid F(t,x) \neq \emptyset\}$ for the incoming and outgoing places of transition t, respectively. Furthermore, we write $\mathsf{pre}(t) = \bigcup_{x \in P} F(x,t)$

for the union of all labels on the incoming arcs of transition t, and $\mathsf{post}(t) = \bigcup_{x \in P} F(t,x)$ for the union of all labels on the outgoing arcs of transition t.

Definition 2. A reversing Petri net is *well-formed*, if for all $t \in T$:

1. $A_V \cap \mathsf{pre}(t) = A_V \cap \mathsf{post}(t)$,
2. $F(t,x) \cap F(t,y) \cap A_V = \emptyset$ for all $x, y \in P$, $x \neq y$.

Thus, a reversing Petri net is well-formed if (1) whenever a variable exists in the incoming arcs of a transition then it also exists on the outgoing arcs, which implies that transitions do not erase tokens, and (2) tokens/bonds cannot be cloned into more than one outgoing places.

As with standard Petri nets the association of token/bond instances to places is called a *marking* such that $M : P \to 2^{A_I \cup B_I}$, where we assume that if $(u,v) \in M(x)$ then $u, v \in M(x)$. In addition, we employ the notion of a *history*, which assigns a memory to each transition $H : T \to \mathbb{N}$. Intuitively, a history of $H(t) = 0$ for some $t \in T$ captures that the transition has not taken place, or every execution of it has been reversed, and a history of $H(t) = k, k > 0$, captures that the transition had k forward executions that have not been reversed. Note that $H(t) > 1$ may arise due to the consecutive execution of the transition with different token instances. A pair of a marking and a history, $\langle M, H \rangle$, describes a *state* of a RPN with $\langle M_0, H_0 \rangle$ the initial state, where $H_0(t) = 0$ for all $t \in T$.

Finally, we define $\mathsf{con}(a_i, C)$, where $a_i \in A_I$ and $C \subseteq 2^{A_I \cup B_I}$, to be the token instances connected to a_i as well as the bonds creating these connections according to set C.

Forward Execution. During the forward execution of a transition in a RPN, a set of tokens and bonds, as specified by the incoming arcs of the transition, are selected and moved to the outgoing places of the transition, as specified by the transition's outgoing arcs, possibly forming or destructing bonds, as necessary. Due to the presence of multiple instances of the same token type, it is possible that different token instances are selected during the transition's execution.

A transition is forward-enabled in a state $\langle M, H \rangle$ of a reversing Petri net if there exists a selection of token instances available at the incoming places of the transition matching the requirements on the transitions incoming arcs. Formally:

Definition 3. Given a RPN (P, T, A, A_V, B, F), a state $\langle M, H \rangle$, and a transition t, we say that t is *forward-enabled* in $\langle M, H \rangle$ if there exists a surjective function $U : \mathsf{pre}(t) \cap A_V \to A_I$ such that:

1. for all $v \in \mathsf{pre}(t)$, if $\mathit{type}(v) = a$ then $\mathit{type}(U(v)) = a$
2. for all $a \in F(x,t)$, then $U(a) \in M(x)$ and for all $(a,b) \in F(x,t)$, then $(U(a), U(b)) \in M(x)$,
3. for all $(a,b) \in \mathsf{post}(t) - \mathsf{pre}(t)$ then $(U(a), U(b)) \notin M(x)$ for all $x \in {\circ}t$.

Thus, t is enabled in state $\langle M, H\rangle$ if (1) there is a type-respecting assignment of token instances to the variables on the incoming edges, with (2) the token instances originating from the appropriate input places of the transition and connected with bonds as required by the variable bonds occurring on the incoming edges, and (3) if a bond occurs in the outgoing edges of the transition but not the incoming ones, then the selected instances associated with the bond's variables should not be bonded together in the incoming places of the transition (thus transitions do not recreate bonds). We refer to U as a forward enabling assignment.

To execute a transition t according to an enabling assignment U, the selected token instances, along with their connected components, are relocated to the outgoing places of the transition as specified by the outgoing arcs, with bonds created and destructed accordingly. Furthermore, the history of the executed transition is increased by one.

Definition 4. Given a RPN (P, T, A, A_V, B, F), a state $\langle M, H\rangle$, and an enabling assignment U, we write $\langle M, H\rangle \xrightarrow{t}_{\mathrm{S}} \langle M', H'\rangle$ where for all $x \in P$:

$$M'(x) = M(x) - \bigcup_{a \in f(x,t)} \mathsf{con}(U(a), M(x)) \cup \bigcup_{a \in f(t,x), U(a) \in M(y)} \mathsf{con}(U(a), S)$$

where $S = (M(y) - \{(U(a), U(b)) \mid (a, b) \in F(y, t)\}) \cup \{(U(a), U(b)) \mid (a, b) \in F(t, x)\}$

$$\text{and} \quad H'(t') = \begin{cases} H(t') + 1, & \text{if } t' = t \\ H(t'), & \text{otherwise} \end{cases}$$

Reversing Execution. We now move on to reversing transitions. A transition can be reversed in a certain state if it has been previously executed and there exist token instances in its output places that match the requirements on its outgoing arcs. Specifically, we define the notion of reverse enabledness as follows:

Definition 5. Consider a RPN (P, T, A, A_V, B, F), a state $\langle M, H\rangle$, and a transition t. We say that t is *reverse-enabled* in $\langle M, H\rangle$ if (1) $H(t) \neq 0$, and (2) there exists a surjective function $W : \mathsf{post}(t) \cap A_V \to A_I$ such that:

1. for all $v \in \mathsf{post}(t)$, if $\mathit{type}(v) = a$ then $\mathit{type}(W(v)) = a$,
2. for all $a \in F(t, x)$, then $W(a) \in M(x)$ and for all $(a, b) \in F(t, x)$, then $(W(a), W(b)) \in M(x)$,
3. for all $(a, b) \in \mathsf{pre}(t) - \mathsf{post}(t)$ then $(W(a), W(b)) \notin M(x)$ for all $x \in \circ t$.

Thus, a transition t is reverse-enabled in $\langle M, H\rangle$ if (1) the transition has been executed and (2) there exists a type-respecting assignment of token instances, from the instances in the out-places of the transition, to the variables on the outgoing edges of the transition, and where the instances are connected with bonds as required by the transition's outgoing edges. Also we do not recreate existing bonds when going backwards. We refer to W as a reversal enabling assignment. To implement the reversal of a transition t according to a reversal enabling assignment W, the selected instances are relocated from the outgoing

places of the transition to the incoming places, as specified by the incoming arcs of the transition, with bonds created and destructed accordingly.

Definition 6. Given a RPN (P, T, A, A_V, B, F), a state $\langle M, H\rangle$, and a transition t reverse-enabled in $\langle M, H\rangle$ with W a reversal enabling assignment, we write $\langle M, H\rangle \overset{t}{\rightsquigarrow} \langle M', H'\rangle$ where for all x:

$$M'(x) = M(x) - \bigcup_{a \in f(t,x)} \mathsf{con}(W(a), M(x)) \cup \bigcup_{a \in f(x,t), W(a) \in M(y)} \mathsf{con}(W(a), S)$$

where $S = (M(y) - \{(W(a), W(b)) \mid (a, b) \in F(t, y)\}) \cup \{(W(a), W(b)) \mid (a, b) \in F(x, t)\}$

$$\text{and} \quad H'(t') = \begin{cases} H(t') - 1, & \text{if } t' = t \\ H(t'), & \text{otherwise} \end{cases}$$

The Autoprotolysis of Water in RPNs. Figure 8 shows the graphical representation of the forming of a water molecule as a RPN. In this model, we assume two token types, H for hydrogen and O for oxygen. They are instantiated via four token instances of H (H_1, H_2, H_3, and H_4) and two token instances of O, (O_1 and O_2). The net consists of five places and three transitions and the edges between them are associated with token variables and bonds, where we assume that $type(o) = type(o_1) = type(o_2) = O$ and $type(h) = type(h_1) = type(h_2) = type(h_3) = type(h_4) = H$. Looking at the transitions, transition t_1 models the formation of a bond between a hydrogen token and an oxygen token. Precisely, the transition stipulates a selection of two such molecules with the use of variables o and h on the incoming arcs of the transition which are bonded together, as described in the outgoing arc of the transition. Subsequently, transition t_2 completes the formation of a water molecule by selecting an oxygen token from place x and a hydrogen token from place v and forming a bond between them, placing the resulting component at place y. Note that the selected oxygen instance in this transition will be connected to a hydrogen token via a bond created by transition t_1; this bond is preserved and the component resulting from the creation of the new $o - h$ bond will be transferred to place y. Finally, transition t_3 models the autoprotolysis reaction: assuming the existence of two distinct oxygen instances, as required by the variables o_1 and o_2 on the incoming arc of the transition, connected with hydrogen instances as specified in $F(y, t_3)$, the transition breaks the bond $o_2 - h_3$ and forms the bond $o_1 - h_3$. As such, assuming the existence of two water molecules at place y, the transition will form a hydronium (H_3^+O) and a hydroxin (OH^-) molecule in place z of the net. The reversibility semantics of RPNs ensures that reversing the transition t_3 will result in the re-creation of two water molecules placed at y, while the use of variables allows the formation of water molecules consisting of different bonds between the hydrogen and oxygen instances.

The first net in Fig. 9 shows the system after the execution of transition t_1 with enabling assignment $U(h) = H_1$, $U(o) = O_1$. Note that the term [1] written over transition t_1 captures that at this point $H(t_1) = 1$ since the

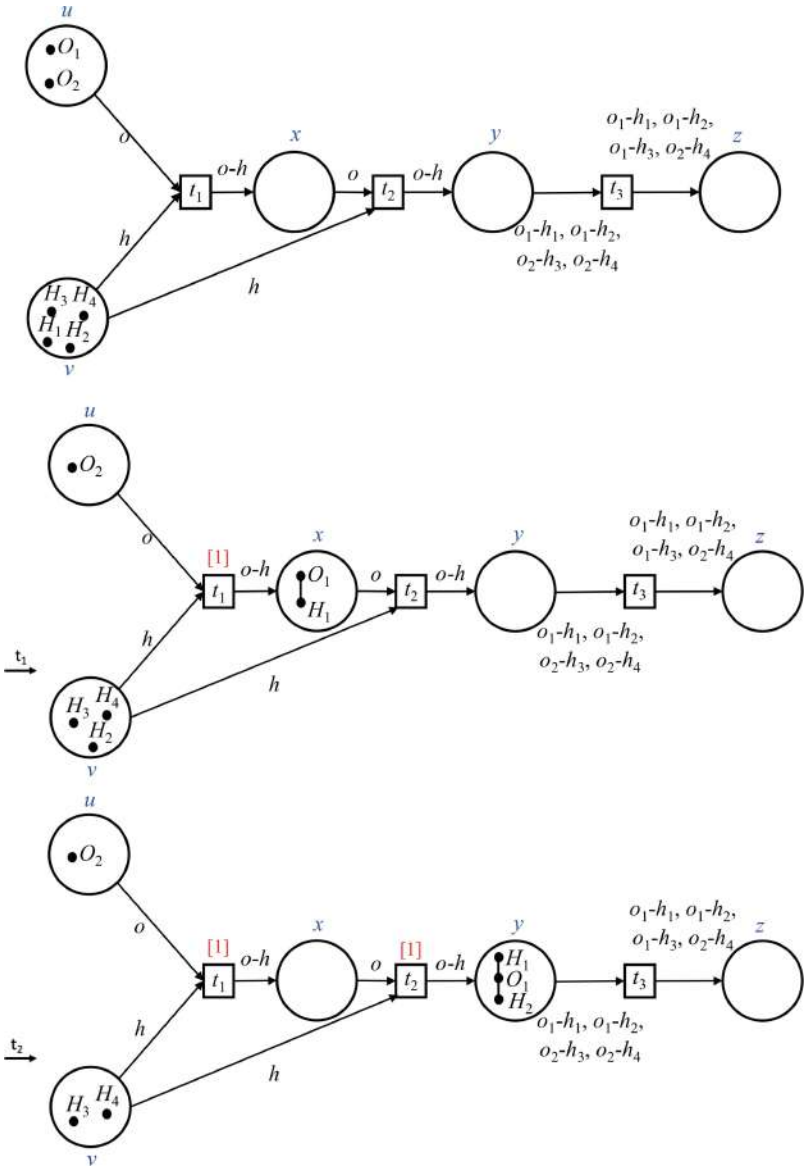


Fig. 8. RPN model of the formation of a water molecule.

transition has been executed once. This notation is generally used for histories in the graphical representation with occasional missing histories corresponding to histories equal to 0. Subsequently, we have the model after execution of transition t_2 with enabling assignment $U(h) = H_2$, $U(o) = O_1$, creating the bond $O_1 - H_2$, thus forming the first water molecule. A second

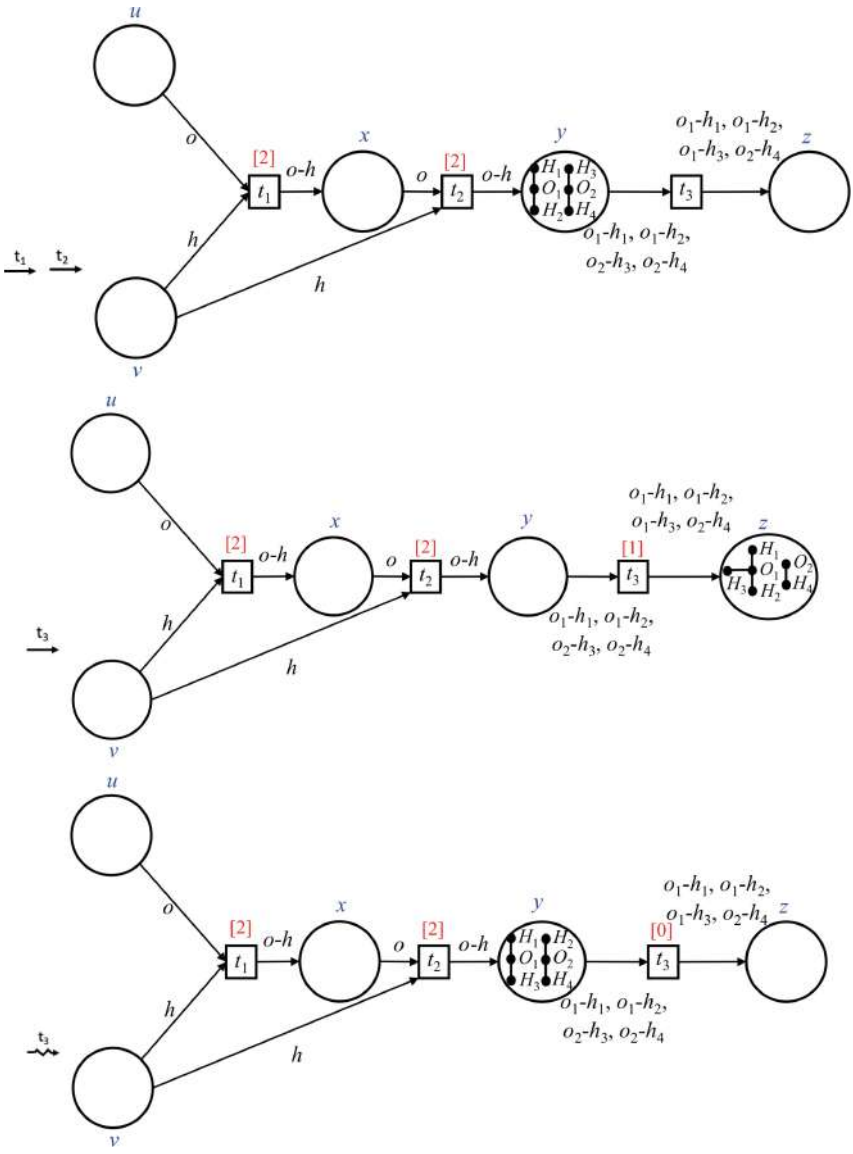


Fig. 9. RPN model of the execution of the autoprotolysis of water.

execution of transitions t_1 and t_2 results in the second molecule of water in the system, placed again at place y, as shown in the third net in the figure. At this state, transition t_3 is forward-enabled and, with enabling assignment $U(o_1) = O_1, U(o_2) = O_2, U(h_1) = H_1, U(h_2) = H_2, U(h_3) = H_3, U(h_4) = H_4$, we have the creation of the hydronium and hydroxide depicted at place z in the fourth net of the figure. At this stage, transition t_3 is now reverse-enabled and

the last net in the figure illustrates the state resulting after reversing t_3 with reversal enabling assignment $W(o_1) = O_1, W(o_2) = O_2, W(h_1) = H_1, W(h_2) = H_3, W(h_3) = H_2, W(h_4) = H_4$.

4 Evaluation

We have presented three formalisms which can be used to model chemical reactions. CCB is a reversible version of ACP that employs communication keys to record executed actions. Its main feature is a mechanism to link forming and breaking of bonds, which gives rise to a type of explicit reversibility we call "locally controlled reversibility". We have modelled a simple covalent chemical reaction in CCB. A similar modelling approach can be used to model more complex atoms and reactions, for example, involving carbon atoms [16]. Finally, CCB can also be used to model reactions beyond simple chemical reactions [14]. In CCB, we can actually distinguish different instances of the same atom or molecule, and of identical actions in a process via the use of subscripts. As mentioned above, the reverse reaction in the autoprotolysis of water can work by transferring any of the hydrogens of the hydronium. When reversing the reaction in CCB, instead of the transition in Sect. 3.1, we could also have done this (writing the transition and the rewrite together):

$$
\begin{aligned}
&(((h_1[1];p).H_1' \mid (h_2[2];p).H_2' \mid (o_1[1], o_2[2], n[5]).O_1') \mid (h_3[5];p).H_3') \\
&\mid (h_4[4];p).H_4' \mid (o_3, o_4[4], n).O_2' \\
&\xrightarrow{\{np[3], \underline{nh_1}[1]\}} \Rightarrow \\
&(((\boldsymbol{h_1}[\mathbf{3}];p).H_1' \mid (h_2[2];p).H_2' \mid (\boldsymbol{o_1}[\mathbf{5}], o_2[2], \boldsymbol{n}).O_1') \mid (h_3[5];p).H_3') \\
&\mid (h_4[4];p).H_4' \mid (\boldsymbol{o_3}[\mathbf{3}], o_4[4], \boldsymbol{n}).O_2'
\end{aligned}
$$

The result is different from that in Sect. 3.1, but identical from a chemical point of view, since the hydrogens are all identical. On the other hand a technique called isotopic labelling can be used to trace atoms by using different isotopes of, in this case hydrogen, confirming that the different options happen in reality. In CCB, we can trace the atoms as well as show which results are identical from a chemical point of view (see Section 6.5 of [16]).

The Bonding Calculus is suitable for modelling in a natural way the autoprotolysis of water by using only bond and unbond actions. Simulations by using a software platform can describe the dynamics of the bonding systems, and so it is possible to test the validity of some underlying assumptions. Also, we can verify various properties of the bonding compounds described by using the calculus.

Reversing Petri Nets are Petri net structures that assume tokens to be distinct and persistent. During the execution of transitions individual tokens can be bonded/unbonded with each other, and the creation/destruction of these bonds is considered to be the effect of a transition, whereas their destruction/creation is the effect of the transition's reversal. Reversing Petri Nets are a natural choice to model and analyse biochemical reaction systems, such as the autoprotolysis

of water, which by nature has multi-party interactions, is inherently concurrent, and features reversible behaviour. In particular, the feature of token multiplicity and the use of variables allows to non-deterministically select different combinations of atoms of a particular element when creating molecules. Also the ability of transitions to break bonds allows to model concerted actions where, for example, a transition simultaneously destroys a water molecule and creates a hydronium whose reversal results in the opposite effect. Moreover, the collective token interpretation adopted in the framework, treating all tokens of the same type as equivalent, allows the reaction to reverse into two different water molecules than the original ones, i.e. using different instances of the atoms (as is possible in CCB). Note that the presented model abstracts away the positive/negative charge of the atoms and captures the existence of electrons by the enabledness of transitions. A model at a lower level of abstraction would be possible by introducing tokens to represent the electrons bonded to the associated atom tokens to illustrate the relevant charges.

The three formalisms presented can model our example fairly well but, as expected, there are some differences. In order to evaluate each formalism, we consider as first criterion if all chemically valid interactions between the compounds of the reaction can be represented well in our formalisms. CCB shows the linked forming and breaking of bonds. RPNs can also express these concerted actions, since a transition enables the simultaneous creation and destruction of bonds. In the Bonding Calculus, this link is not expressed. Each of the formalisms can perform the forward reaction using any of the hydrogens involved. CCB and RPNs can perform the reverse reaction by transferring arbitrary hydrogens, whereas the Bonding Calculus in the reverse reaction permits only the transfer of exactly those hydrogens that were used in the forward reaction. All models presented use subscripts and enable the tracking of atoms.

The other criterion for assessing the suitability of our formalisms for the modelling of chemical reactions is to ask if they enable in the produced model any transitions that actually do not occur in reality. Each formalism does not permit a H_3O^+ molecule to be formed directly. CCB allows one reaction which is not realistic: If there are many water molecules and therefore several hydroxide and water molecules at the same time, it is possible that the remaining hydrogen is transferred from the hydroxide to a water. In reality, this is not possible since the hydroxide is strongly negatively charged and no hydrogen bond can form. Due to the nondeterministic behaviour of processes written with the '+' operator, such as those for hydrogen and oxygen in Subsect. 3.2, the Bonding Calculus also presents the same problem. However, this is not the case for RPNs since, on the one hand, a transition's conditions make restrictions on the types of molecules that will participate in a transition firing or its reversal and, on the other hand, places impose a form of locality for molecules. For instance, in the autoprotolysis example, each place is the location of specific types of molecules, e.g., transition t_3 modelling the autoprotolysis reaction is only applied on water molecules and its reversal only on pairs of a hydronium and a hydroxide molecule, as required.

There are a number of software tools that can aid simulation and analysis for our formalisms. Regarding the Bonding Calculus, we can simulate various bonding descriptions by using an existing software platform called UPPAAL (as shown in [1]). For CCB, there is a simulation tool presented in [14]. It allows a much closer form of representation of chemical notation than that possible with a typical programming language. Reversing Petri nets have been shown to be closely related to Coloured Petri Nets, as a subset of the former model has been encoded into the latter [3]. Thus, an algorithmic translation can be implemented that transforms RPNs to CPNs in an automated manner using the transformation techniques discussed in [3]. This allows RPNs to exploit tools such as CPNTools that support traditional models of Petri nets.

5 Conclusion

We have presented the Calculus of Covalent Bonding, the Bonding Calculus, and Reversing Petri Nets as models of chemical reactions and reversible processes in general. We have shown that they can all model the out-of-causal-order reversibility present in such reactions. We have also noted that the two process calculi allow few reactions which do not happen in reality. This is due to the modelling that abstracts away from some chemical properties of atoms and molecules such as, for example, spacial arrangement and distance between molecules. In future work, we plan to develop these formalisms further and apply them to the modelling and reasoning about reversible biochemical reactions and processes.

References

1. Aman, B., Ciobanu, G.: Bonding calculus. Nat. Comput. **17**(4), 823–832 (2018). https://doi.org/10.1007/s11047-018-9709-7
2. Baldan, P., Cocco, N., Marin, A., Simeoni, M.: Petri nets for modelling metabolic pathways: a survey. Nat. Comput. **9**(4), 955–989 (2010)
3. Barylska, K., Gogolińska, A., Mikulski, Ł., Philippou, A., Piątkowski, M., Psara, K.: Reversing computations modelled by coloured Petri nets. In: Proceedings of ATAED 2018. CEUR Workshop Proceedings, vol. 2115, pp. 91–111 (2018)
4. Blätke, M.A., Heiner, M., Marwan, W.: Petri nets in systems biology. Technical report, Otto-von-Guericke University Magdeburg (2011)
5. Chaouiya, C.: Petri net modelling of biological networks. Brief. Bioinform. **8**(4), 210–219 (2007)
6. Ciocchetta, F., Hillston, J.: Bio-PEPA: a framework for the modelling and analysis of biological systems. Theoret. Comput. Sci. **410**(33–34), 3065–3084 (2009)
7. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-based modelling of cellular signalling. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 17–41. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74407-8_3
8. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19

9. Faeder, J.R., Blinov, M.L., Hlavacek, W.S.: Rule-based modeling of biochemical systems with BioNetGen. Methods Mol. Biol. **500**, 113–167 (2009)
10. Fages, F., Soliman, S., Chabrier-Rivier, N.: Modelling and querying interaction networks in the biochemical abstract machine BIOCHAM. J. Biol. Phys. Chem. **4**, 64–73 (2004)
11. Fokkink, W.: Introduction to Process Algebra. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-662-04293-9
12. Hofestädt, R.: A Petri net application of metabolic processes. J. Syst. Anal. Model. Simul. **16**, 113–122 (1994)
13. Hofestädt, R., Thelen, S.: Quantitative modeling of biochemical networks. Silico Biol. **1**(1), 39–53 (1998)
14. Kuhn, S.: Simulation of base excision repair in the calculus of covalent bonding. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 123–129. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_8
15. Kuhn, S., Ulidowski, I.: A calculus for local reversibility. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 20–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_2
16. Kuhn, S., Ulidowski, I.: Local reversibility in a calculus of covalent bonding. Sci. Comput. Program. **151**(Supplement C), 18–47 (2018)
17. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order Pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_20
18. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Controlled reversibility and compensations. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 233–240. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_19
19. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Reversing higher-order Pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 478–493. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_33
20. Matsuno, H., Nagasaki, M., Miyano, S.: Hybrid Petri net based modeling for biological pathway simulation. Nat. Comput. **10**(3), 1099–1120 (2011)
21. Milner, R. (ed.): A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
22. Peleg, M., Rubin, D.L., Altman, R.B.: Using Petri net tools to study properties and dynamics of biological systems. J. Am. Med. Inform. Assoc. **12**(2), 181–199 (2005)
23. Philippou, A., Psara, K.: Reversible computation in Petri nets. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 84–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_6
24. Philippou, A., Psara, K., Siljak, H.: Controlling reversibility in reversing Petri nets with application to wireless communications. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 238–245. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_15
25. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Logic Algebraic Program. **73**(1–2), 70–96 (2007)
26. Phillips, I., Ulidowski, I.: Reversibility and asymmetric conflict in event structures. In: D'Argenio, P.R., Melgratti, H. (eds.) CONCUR 2013. LNCS, vol. 8052, pp. 303–318. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40184-8_22
27. Phillips, I., Ulidowski, I., Yuen, S.: Modelling of bonding with processes and events. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 141–154. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38986-3_12

28. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_18
29. Popova-Zeugmann, L., Heiner, M., Koch, I.: Time Petri nets for modelling and analysis of biochemical networks. Fundam. Informaticae **67**(1–3), 149–162 (2005)
30. Priami, C.: Stochastic π-calculus. Comput. J. **38**(7), 578–589 (1995)
31. Priami, C., Quaglia, P.: Beta binders for biological interactions. In: Danos, V., Schachter, V. (eds.) CMSB 2004. LNCS, vol. 3082, pp. 20–33. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-25974-9_3
32. Reddy, V.N., Mavrovouniotis, M.L., Liebman, M.N.: Petri net representations in metabolic pathways. In: Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology, pp. 328–336. AAAI (1993)
33. Regev, A., Panina, E.M., Silverman, W., Cardelli, L., Shapiro, E.: BioAmbients: an abstraction for biological compartments. Theoret. Comput. Sci. **325**(1), 141–167 (2004)
34. Regev, A., Shapiro, E.: The π-calculus as an abstraction for biomolecular systems. In: Ciobanu, G., Rozenberg, G. (eds.) Modelling in Molecular Biology, pp. 219–266. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-642-18734-6_11
35. Reisig, W.: Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33278-4
36. Ulidowski, I.: Equivalences on observable processes. In: Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science, pp. 148–159. IEEE (1992)
37. Ulidowski, I., Phillips, I., Yuen, S.: Reversing event structures. New Gener. Comput. **36**(3), 281–306 (2018)
38. Voss, K., Heiner, M., Koch, I.: Steady state analysis of metabolic pathways using Petri nets. Silico Biol. **3**(3), 367–387 (2003)

Reversible Control of Robots

Ulrik Pagh Schultz(✉)

SDU UAS, MMMI, University of Southern Denmark, Odense, Denmark
ups@mmmi.sdu.dk

Abstract. Programming industrial robots is challenging due to the difficulty of precisely specifying general yet robust operations. As the complexity of these operations increases, so does the likelihood of errors. Certain classes of errors during industrial robot operations can however be addressed using reverse execution, allowing the robot to temporarily back out of an erroneous situation, after which the operation can be automatically retried. Moreover reverse execution permits automatically deriving programs that physically reverse the operations of an industrial robot. This can be useful in industrial assembly, where a disassembly program can be automatically derived from the assembly program.

In this case study we investigate robotic assembly from the point of view of reversibility, investigating to what extent program inversion of a robotic assembly sequence for a given product can be considered to derive a robotic disassembly sequence for this same product, and investigating to what extent changing the execution direction at runtime (i.e., backtracking and retrying) using program inversion can be used as an automatic error handling procedure. The programming model used to reversibly control industrial robots is based on an abstract semantics-based model, extended with various features required for reversible control of industrial robots in real-world scenarios, and implemented as a domain-specific programming language.

1 Introduction

Robots normally have one or more degrees of freedom controlled by a computational process; using reversible computing to control the robot potentially gives rise to new reverse behaviours. For example, major industrial robot manufacturers such as ABB and KUKA offer limited forms of ad-hoc reverse execution for interactive programming and debugging, but due to limitations in the underlying execution models, their programming models are incapable of reversing complex actions such as steps of an industrial assembly process [5,6]. We attribute the ad-hoc limitations to the lack of an underlying reversible model. The first investigation of fully reversible robot behaviours was for self-reconfigurable robots [10]. The useful application of reversibility to this type of robot is however only observed for self-reconfiguration operations, significantly limiting the notion of

The author acknowledges partial support of COST Action IC1405 on Reversible Computation - Extending Horizons of Computing.

I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 177–186, 2020.
https://doi.org/10.1007/978-3-030-47361-7_8

reversibility and real-world interaction that can be studied using this type of robot. To better understand the underlying relation between reversible computation and physical reversibility, we in this case study investigate reversible control of industrial robots.

Programming industrial robots is challenging due to the difficulty of precisely specifying general yet robust operations. As the complexity of these operations increases, so does the likelihood of errors. Certain classes of errors during industrial robot operations can however be addressed using reverse execution, allowing the robot to temporarily back out of an erroneous situation, after which the operation can be automatically retried. Specifically, this approach has been shown to be useful for automatic error recovery for small-sized batch production of assembly operations [11]. Moreover, reversibility can in this case be used to automatically derive a disassembly sequence from a given assembly sequence, or vice versa. These results were demonstrated using an initial design and implementation of a reversible domain-specific language (DSL) for specifying such assembly sequences [5,11]. The area however remains largely unexplored, both from a theoretical and practical point of view. There is for example a large design space for different programming language approaches, both in terms of the generality of the language and the means by which reversibility is achieved. At a more fundamental level, the notion of reversible control of a reversible physical system remains largely unexplored. From a practical point of view, only the specific case of assembly operations has been investigated, and only using a specific set of industrial use cases. There has been no attempt at integration into an existing robotics platform, although we observe that many existing platforms offer limited notions of reversibility for using during programming and debugging.

The result of this case study is significant progress in the area of reversibility for industrial robots [4]. Key developments include an improved understanding of the interaction between reversible computing and real-world systems that only are partially reversible, as well as a substantial experimental evaluation of the use of reversible languages to control industrial robots performing assembly and disassembly in the context of small-batch production. Overall this work experimentally demonstrates the use of reversible computing to improve system reliability.

2 Related Work

Reversibility has previously been investigated for self-reconfigurable robots. Self-reconfigurable, modular robots are distributed robotic devices that can autonomously change their physical shape [13]. Self-reconfiguration from one shape to another is typically achieved through a specific sequence of actuation operations distributed across the modules of the robot. Automatically reversing the sequence of operations can bring the robot back to its initial shape, as has been experimentally demonstrated using the DynaRole reversible language [10]. DynaRole however only allows simple sequences of operations to be reversed, which is suitable for reversing self-reconfiguration sequences, but lacks

the generality needed to implement more complex behaviours. Initial ideas on generalising the DynaRole language to support a wider range of modular robot control scenarios retain the possibility of reversing distributed sequences [8,9], but have neither been formalised nor experimentally demonstrated.

Large-scale modular robotic systems can be considered as intensive parallel systems [7]. Reversibility for intensive parallel systems was studied by Agrigoroaiei and Ciobanu [1]. Here, the process of reversing is presented as a form of duality (a notion from category theory). A related approach presenting reversibility for the bio-inspired formalism of membrane systems is given by the same authors [2].

Partial reversibility has been studied for reversible programming languages [12] using logging of program state to handle irreversible operations. This approach would in our case correspond to recording the motions of the robot and replaying them in reverse, which is applicable to any operation but does not normally serve to reverse actions in the real world. Rather, our approach relies on the programmer explicitly writing reverse code that, through a different sequence of operations, brings the system back to a previous state. This approach can be compared to causal-consistent reversibility [3] in the sense that the observable events (i.e., the state of the system the robot is working on) is reversed in a consistent way; unlike causal-consistent reversibility we however require the programmer to manually implement the basic reverse operations using the notion of indirect reversibility.

3 Reversible Assembly Tasks

We investigate robotic assembly tasks from the point of view of reversibility, investigating to what extent program inversion of a robotic assembly sequence for a given product can be considered to derive a robotic disassembly sequence for this same product, and investigating to what extent changing the execution direction at runtime (i.e., backtracking and retrying) using program inversion can be used as an automatic error handling procedure [4].

3.1 Robotics, Assembly, and Reversibility

Robotic assembly and disassembly is done in terms of sequences of operations such as precise placement of objects, insertions with tight fits, screwing operations and so forth. All are challenged by uncertainties from sensors, robot kinematics and part tolerances; not all are reversible, some are not even repeatable. Our approach has been tested with a standard robotic platform based on a Universal Robots UR5, shown in Fig. 1 together with the two industrial assembly cases used to evaluate the approach [4].

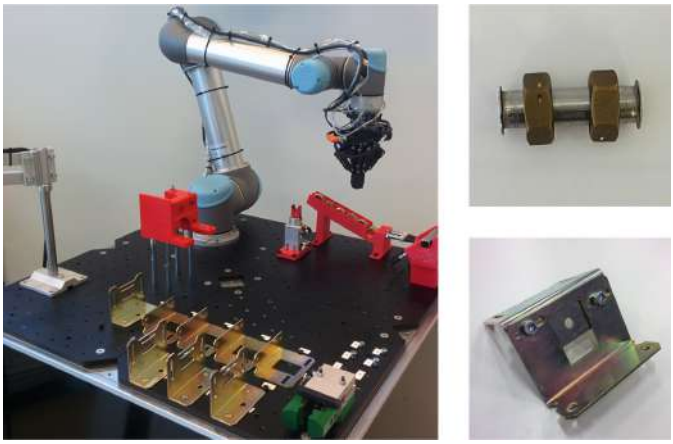


Fig. 1. The experimental platform and two assembly test cases (from [4]).

3.2 Reversibility

Many physical phenomena and actions are in principle reversible, although this reversibility may depend on the abstraction level at which they are observed. For example, an industrial robot that pushes an object to a new position could easily move this object back to its original position, but cannot simply do this by reversing its movements as pulling requires gripping the object first. Moreover, some operations, such as cutting, should in general be considered nonreversible. A study of 13 real-world industrial cases showed roughly 76% of the operations to be reversible [4], but many of the operations require the robot to perform different physical actions to reverse a given action. Based on this observation, we can divide the reversible operations into two categories: *directly reversible* and *indirectly reversible* operations. Operations which can be reversed through program inversion are considered directly reversible. Indirectly reversible operations on the other hand can be reversed, but require a different sequence of instructions.

3.3 Repeatability

Unlike Janus-style reversible computing, where programs can be said to be time-invertible [14], with robotics physical changes made to the environment from the execution influences the repeatability of operations. Operations that can be done again and again can be referred to as *fully repeatable*. Other actions can only be done a limited number of times, e.g., due to wear and tear, and are said to be *partially-repeatable*. Last, *nonrepeatable* operations are those that cannot be retried.

3.4 Reversibility and Repeatability

Considering reversibility and repeatability together leads to a classification of robotic assembly operations [4]. Operations that are fully repeatable and directly reversible can be automatically managed using a program inversion approach, whereas indirectly reversible operations require explicit reverse code to be provided by a programmer, partially repeatable operations limit how many times a program can be reversed, and certain operations are fundamentally irreversible and thus mark points across which the program cannot be reversed.

4 Programming Model

The programming model developed in our case study is based on an abstract semantics-based model [11] extended with various features required for reversible control of industrial robots in real-world scenarios [4].

4.1 Basic Model

A robot assembly task is programmed as a sequential flow of operations. It is sequential since in practice assembly tasks tend to be a simple sequence of operations (except for error handling, but we aim to automatically handle errors using reverse execution). Reversibility is relevant due to the presence of random behaviour of the physical operations: reversing and re-executing an operation may produce a different results. Each operation represents high-level assembly case logic and is a sequence of instructions. *Instructions* are either reversible, providing a two-way reversible forward/backward mapping of hardware instructions, or non-reversible, providing a single-directional mapping. Instructions are implemented using traditional nonreversible programming. Taking inspiration from Janus [14], it is possible to both call and uncall operations, the latter causing the operation to be interpreted in reverse.

The programming model used to represent robot assembly tasks is built on the following principles. (1) Instructions always map the robot system from a known state to a known state, but may have different semantics for forward and reverse. (2) Indirect reversibility is achieved by modelling instruction sequences that are different for forwards and reverse execution using the principle of overridden reverse flow, where users can write different code for forwards and backwards execution. (3) Instructions can be marked as nonreversible. A directed graph is used to model the underlying reversible assembly sequence. In this graph each node corresponds to a primitive instruction which is executable on the physical platform. Furthermore, each node contains pointers to the next forward instruction and the next reverse instruction (if any). Overall the graph is evaluated through forward/backwards interpretation and each instruction is evaluated using instruction inversion in the sense that different semantics are applied for forward and backwards execution.

```
operation attach_nut_bolt {
  state begin_nut_bolt (...tool pos...) bolt:(...pos...) nut:(...pos...)
  moveto (...pos above table...)
  pickup (nut, fixed_gripper, (...pos of nut...))
  moveto (...)
  ...
}
operation apply_and_turn_nut { ...commands... }
reverse { ...commands that undo apply_and_turn_nut...  }
```

Fig. 2. Sample RASQ program, vector constants are omitted for clarity (adapted from [11]).

4.2 Implementation

The basic model provides the foundation for programming realistic assembly cases [4]. The principle of indirect reversibility is in practice instantiated in many different ways, such as movement or error detection instructions that only activate in one execution direction. Error handling is implemented in the interpreter: when an error is detected during forwards execution the direction is immediately reversed for a number of steps, after which forwards execution is again resumed. The same model is applied for execution in reverse, and even applies recursively, i.e., if an error is detected during reverse execution triggered due to an error. Each instruction carries specific information describing how to handle switching of execution direction, specifically whether the instruction should be repeated in reverse or not when switching direction due to the instruction failing. A simple error handling strategy that changes execution direction for a random number of steps and that ensures termination by bounding the total number of steps was observed to work well in practice.

4.3 Language

The idea of reversible control of industrial robots was initially presented using a high-level programming language [11]. An example is shown in Fig. 2. The program declares two operations, `attach_nut_bolt` and `apply_and_turn_nut`. The operation `attach_nut_bolt` only specifies a single (forwards) body for both forwards and reverse execution, so reverse execution will inversely evaluate the forwards body in reverse order. The first statement is a state assertion, named `begin_nut_bolt`, specifying the spatial positioning of the tool and the respective positions of the bolt and nut objects. The next statement of the program is a move, which moves the robot to the given position (again, the position is given as a constant, not shown). After the move follows a pick up instruction that causes the pickup operation associated with the name `fixed_gripper` and the object `nut` to be evaluated. Last follows the declaration of the second operation `apply_and_turn_nut`, which is not shown in detail, but has both a forwards and a reverse body, so forwards execution evaluates the forwards body in forwards

```
operation("screwdriver_activate").
  io(screwdriver, Switch::on).
  wait(0.3).
  wait(screwingFinished).
  reverseWith("screwing_finished_backwards");
  io(screwdriver, Switch::off).
  io(screwdriverBackwards, Switch::off);
```

Fig. 3. Sample SCP-RASQ program (adapted from [4]).

order, and reverse execution evaluates the reverse body in forwards order (i.e., in the order written in the program).

In practice it turned out to be more useful to rely on an internal DSL implemented in C++, using a model-driven approach that serialises the program to an XML structure that can subsequently be instantiated as the graph structure used by the reversible interpreter. This internal DSL, named SCP-RASQ for "Simple C++ RASQ", is exemplified in Fig. 3. This program declares an operation that performs IO operations to communicate with the screwdriver, and shows how indirect reversibility can be programmed in-place using the `reverseWith` declaration.

5 Results

This section will give an overview of the experimental results demonstrated in earlier work on several industrial use cases [4].

5.1 Methodology

Error recovery using reverse execution was tested using two industrial assembly tasks use-cases; the physical robot platform and the assembled products are shown in Fig. 1. An SCP-RASQ program was created for each of the use cases. Both cases include a final step where the finished product is discarded into a box. This step was not performed when running the programs backwards, as it is a nonreversible task since our current setup cannot bin-pick the part out again.

5.2 Experiment 1: Reversing the Programs

Both use-cases were used to test the principle of reversible assembly. Forward execution performs assembly while reverse execution performs disassembly. For each case the program is executed forward to assemble an object. Afterwards the finished objects is then manually placed back into the system, and the program is then executed backwards to disassemble the object. This was done a total of three times for each case, with no errors.

In our test programs directly reversible operations made up 45% of all operations. Moreover, directly reversible operations such as the "pick screwdriver"

were used in both their forward and backwards form in the same program using the call and uncall functionality. Both use-cases could be made almost entirely reversible using either directly or indirectly reversible operations through the execution model and the programming language. We believe that if the reversibility concept was to be integrated more deeply into the design of assembly processes and external equipment such as feeders, an even greater degree of directly reversible instructions could be achieved.

5.3 Experiment 2: Assembling 100 Objects

By assembling a large number of objects the use of reverse execution as an effective error correction tool was demonstrated. The workcell was set to assemble 100 objects of each type consecutively and without pause. During these 200 assemblies a total of 22 errors occurred, of which 18, corresponding to 82%, were automatically resolved and corrected using reverse execution. Errors that were automatically corrected include failed peg-in-hole operations (fixed by backtracking and trying again), dropping a tube (fixed by reversing until a new tube was picked from the feeder), failed to grasp a screw, and screwing failing due to misalignment. Errors that could not be automatically corrected include air-tubing from the gripper getting stuck on the platform, causing the gripper to misalign, and a screw being inserted at a skewed angle causing a bracket to misalign, which could not be corrected as the system had no means of detecting the bracket misalignment.

This experiment shows that reverse execution is capable of solving a wide variety of errors and that the exact method for solving each kind of error need not always be the same, as backtracking was done randomly at different lengths and sometimes resulted in different solutions to the same problem. Moreover we see that the backtracking system is promising in handling errors related to small uncertainties in the assembly tasks, but that errors resulting in larger and mechanical failures still need to be addressed either in the design phase or by some other error handling mechanism. Last, the experiments also show that while reverse execution can be used for solving a wide variety of errors, it also places strong demands on the error detection system.

6 Conclusion

From a society point of view, industrial robots are key to maintaining production in Europe, and reversible computation has the potential to increase robustness for specific kinds of operations such as small-batch assembly, and moreover facilitate the programming of such operations. In this case study we have introduced a programming model which enables robot assembly programs to be executed in reverse. We have experimentally demonstrated that temporarily switching the direction of program execution can be an efficient error recovery mechanism. Moreover, we have shown that additional benefits arise from supporting

reversibility in our robotic assembly language, namely increased code reuse and automatically derived disassembly sequences.

This case study has resulted in an improved understanding of the interaction between reversible computing and real-world systems that only are partially reversible, as well as a substantial experimental evaluation of the use of reversible programming languages to control industrial robots performing assembly and disassembly in the context of small-batch production. Overall this case study has experimentally demonstrated the use of reversible computing to improve system reliability.

Acknowledgements. Thanks to Gabriel Ciobanu for help in describing the related work on reversibility of massively parallel systems.

References

1. Agrigoroaiei, O., Ciobanu, G.: Dual P systems. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 95–107. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-95885-7_7
2. Agrigoroaiei, O., Ciobanu, G.: Reversing computation in membrane systems. J. Logic Algebraic Program. **79**(3), 278–288 (2010)
3. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. Bull. EATCS **114** (2014)
4. Laursen, J., Ellekilde, L., Schultz, U.: Modelling reversible execution of robotic assembly. Robotica **36**(5), 625–654 (2018)
5. Laursen, J.S., Schultz, U.P., Ellekilde, L.P.: Automatic error recovery in robot assembly operations using reverse execution. In: Evers, C., Sheaffer, J., Tourbabin, V., Naylor, P.A., Romanoni, A., Matteucci, M. (eds.) International Conference on Intelligent Robots and Systems (IROS 2015). IEEE/RSJ (2015)
6. Mühe, H., Angerer, A., Hoffmann, A., Reif, W.: On reverse-engineering the KUKA robot language. In: Schultz, U.P., Stinckwich, S., Ziane, M. (eds.) Proceedings of the First International Workshop on Domain-Specific Languages for Robotic Systems (DSLRob 2010) (2010). arXiv:1009.5004 [cs.RO]
7. Păun, G.: Membrane Computing. An Introduction. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-642-56196-2
8. Schultz, U.P.: Using scheme to control simulated modular robots. In: Danvy, O. (ed.) Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, pp. 90–95. ACM (2012)
9. Schultz, U.P.: Towards a general-purpose, reversible language for controlling self-reconfigurable robots. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 97–111. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_8
10. Schultz, U., Bordignon, M., Støy, K.: Robust and reversible execution of self-reconfiguration sequences. Robotica **29**, 35–57 (2011)
11. Schultz, U.P., Laursen, J.S., Ellekilde, L.-P., Axelsen, H.B.: Towards a domain-specific language for reversible assembly sequences. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 111–126. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_7

12. Tyagi, N., Lynch, J., Demaine, E.D.: Toward an energy efficient language and compiler for (partially) reversible algorithms. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 121–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_8
13. Yim, M., et al.: Modular self-reconfigurable robot systems [grand challenges of robotics]. IEEE Robot. Autom. Mag. **14**(1), 43–52 (2007)
14. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Proceedings of the 5th Conference on Computing Frontiers (CF 2008), pp. 43–54. ACM (2008)

Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event Simulation

Markus Schordan[1(✉)], Tomas Oppelstrup[1], Michael Kirkedal Thomsen[2], and Robert Glück[2]

[1] Lawrence Livermore National Laboratory, Livermore, USA
{schordan1,oppelstrup2}@llnl.gov
[2] University of Copenhagen, Copenhagen, Denmark
m.kirkedal@di.ku.dk, glueck@acm.org

Abstract. Optimistic parallel discrete event simulation (PDES) requires to do a distributed rollback if conflicts are detected during a simulation due to the massively parallel optimistic execution approach. When a rollback of a simulation is performed each node that is determined to be in a wrong state must be restored to one of its previous states. This can be achieved through reverse computation or by restoring a previous checkpoint. In this paper we investigate and compare both approaches, reverse computation and a variant of checkpointing, incremental state saving (also called incremental checkpointing), to restore a previous program state as part of an optimistic parallel discrete event simulation. We present a benchmark model that is specifically designed for evaluating the performance of approaches to reversibility in PDES. Our benchmarking model has mathematical properties that allow to tune the amount of arithmetic operations relative to the amount of memory operations. These tuning opportunities are the basis for our systematic performance evaluation.

1 Introduction

Discrete event simulation (DES) is a simulation paradigm suitable for systems whose states are modeled as changing *discontinuously* and *irregularly* at discrete moments of simulation time. State changes occur at simulation times that are calculated dynamically rather than determined statically as typical in time-stepped simulations. Most irregular systems whose behavior is not describable by continuous equations and do not happen to be suitable for simple time-stepped models are candidates for DES. Efficient *parallel* discrete event simulation (PDES) is much more complicated than the sequential version. There are two broad approaches to resolving the PDES synchronization issue, called *conservative* and *optimistic* [1]. Recently Omelchenko and Karimabadi have developed an asynchronous flux-conserving DES technique for physical simulations [2]. Their preemptive event processing approach to parallel synchronization complements

I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 187–207, 2020.
https://doi.org/10.1007/978-3-030-47361-7_9

standard optimistic and conservative strategies for PDES. In this paper we will discuss optimistic PDES, which requires reversibility, in more detail.

In particular, we will focus on PDES using the Time Warp optimistic synchronization method [3]. The optimistic classification of Time Warp implies that it employs speculative execution to enable parallelism. In order to allow roll-backs needed to resolve incorrect speculation, the original formulation of Time Warp utilized checkpointing of the entire system state. This can be very wasteful, so in recent years reverse computation has become a key concept in optimistic parallel discrete event simulation [4,5], as it allows one to reduce the overhead in the forward execution in comparison to checkpointing and, thus, improve the performance. Fundamentally, there are two ways to achieve reversibility: (1) incremental state saving and (2) reverse execution. *Incremental state saving* (also called incremental checkpointing in [5]) is a well-established approach, which has the advantage that only a few language constructs need to be augmented to establish reversibility of an arbitrary piece of code. However, it (often) results in a high runtime overhead as any checkpointing is a memory-heavy method. *Reverse execution* is based on the idea that for many programs there exists an inverse program that can uncompute all results of the (forward) computed program. The inverse program can be achieved either through implementation of reverse code from a given forward code, or by implementing the program in a reversible programming language that offers the capability to automatically generate the inverse program: the imperative reversible language Janus [6] has such functionality.[1]

In this paper we systematically evaluate the generation of forward and reverse C++ code from Janus code (Sect. 4) as well as automatically generated code based on incremental state saving (Sect. 5). We also discuss the differences in methodology, whether a model code is written in a "destructive" language such as C/C++ or in the reversible language Janus, and its applications when implementing (and debugging) a model for PDES.

For this purpose and in order to validate the simulator and also check correctness of generated code, we have developed a new discrete event benchmark model that can be scaled in various dimensions. For execution of our model codes we use the ROSS general purpose discrete event simulator. Our new discrete event benchmark model is similar to the classic PHOLD benchmark model, but includes some extra state variables and computations that aid in detecting simulation errors. In our new model each event involves non-commutative matrix algebra, and the matrix that results from the simulation of the model serves as a checksum or hash of the simulation, and is sensitive to the order of events. The size of this matrix can be controlled by the user, as can the number of bits in its elements. This new benchmark is particularly useful for debugging simulations that are computed with the Time Warp Algorithm as its mathematical properties allow for checking of various assertions.

In our new model we can also tune the amount of arithmetic operations relative to the amount of memory modifying operations. This enables a systematic

[1] Online Janus interpreter at https://topps.diku.dk/pirc/?id=janus.

comparison of hand-written reverse code with multiple approaches of automatically generated reverse code and code instrumented for incremental state saving.

In our performance evaluation we use several different versions of the model code: (1) the original forward code with hand written reverse code, (2) Backstroke instrumented code to perform incremental state saving [7], and (3) Janus generated code for forward/reverse functions.

The forward/reverse code generated from Janus is particularly interesting because it allows to get forward code with no memory overhead and in some cases no runtime overhead, whereas for instrumented code one can only try to reduce the runtime and memory overhead in the forward code.

To the best of our knowledge this is the very first runtime comparison of the two approaches to reversible computation: generating reverse code and incremental state saving. In the optimistic PDES setting incremental state saving is suitable because optimistic PDES follows the Forward-Reverse-Commit (FRC) paradigm. In that paradigm, after an event has been executed in the forward direction, it can either be reversed (e.g. in the case it was incorrect to run it forward in the first place), or committed (when it has been proved that it was a correct event). When an event is committed its associated data is no longer needed, which allows to dispose recorded traces with every commit. In this paper we also investigate whether the combination of both the reversible language and incremental checkpointing approaches can be beneficial.

After giving a brief overview of PDES in Sect. 2, we describe our benchmark model and its properties in Sect. 3. In Sect. 4 we describe the reversible language Janus and how we generated forward/reverse function from Janus code. In Sect. 5 we briefly describe what source code transformations are applied to code to support incremental state saving with the Forward-Reverse-Commit paradigm. In Sect. 6 we describe the discrete event simulator that we use for optimistic parallel discrete event simulation and some adaptations that we implemented to better support the Forward-Reverse-Commit paradigm. The performance evaluation results are presented in Sect. 7. In Sect. 8 we discuss previous work that is related to our evaluated approaches and in Sect. 9 we discuss conclusions from the observed performance results.

2 Optimistic Parallel Discrete Event Simulation (PDES)

In this section we give a brief overview of PDES. A more detailed overview can be found in our previous work [7]. The general approach is to divide the simulation and its state into semi-independent units called LPs (logical processes) that can execute concurrently and communicate asynchronously, each maintaining its own state. A simulated event generally triggers a state change in one LP and affects only that LP's state. Any event may schedule other events to happen in the future of the current LP's simulation time. Events scheduled for other LPs must be transmitted to them as event messages with a timestamp indicating the simulation time when the event happens. Arriving event messages get enqueued in the event queues of the receiving LPs in increasing time stamp order. The LP has to allocate enough memory to store these queues.

Every LP must execute all of its events in strictly non-decreasing timestamp order irrespective of the order in which events may arrive or what timestamps they may carry. This poses a synchronization problem.

In contrast to optimistic PDES, conservative synchronization in conservative PDES uses conventional process blocking primitives along with extra knowledge about the simulation model (called *lookahead* information) to prevent the execution from ever getting into a situation in which an event message arrives at an LP with a timestamp in its past. Conservative synchronization is limited to models with static communication graphs.

Optimistic synchronization, by contrast, employs speculative execution to allow dynamic communication graphs and exposure of more parallelism. As a result, there is the danger of a *causality violation* when an LP that is behind in simulation time, e.g. at t_1, sends an event message with a (future) timestamp $t_2 > t_1$ that arrives at a receiver that has already simulated to time $t_3 > t_2$ due to its optimistic execution. In that case the receiver has already simulated past the simulation time when it *should* have executed the event at t_2, but it would be incorrect to execute events out of order because this may produce different results. Whenever that occurs, the simulator needs to roll back the LP from t_3 to the state it was in at time t_2, cancel all event messages the LP had sent after t_2, execute the arriving event, and then re-execute forward from time t_2 to t_3 and beyond. All event executions are therefore *speculative* or *provisional*, and are subject to rollback if the simulator detects a local causality conflict.

Each LP computes its local virtual time (LVT) based on the time stamps of event messages it receives. Because of rollbacks the LVT can also be reset to an earlier point in time. The global virtual time (GVT) is defined to be the minimum of all of the LVTs. Several algorithms exist to compute an estimate of the GVT during the simulation. Any events with time stamps older than GVT can be *committed* because it is guaranteed that they never need to be reversed. For more detail see [3,5]. That events are committed once they are older than GVT, allows to delete all information that may have been stored to enable reversibility. This commit operation is the same that we also use for incremental state saving, described in Sect. 5, to dispose recorded execution traces of memory modifying operations.

3 PDES Model Benchmark

In order to validate the simulator and also check the correctness of automatically generated code suitable for reversible computation, we have developed a new discrete event benchmark model. It is similar to the classic PHOLD benchmark model, but includes some extra state variables and computations which aid in detecting simulation errors. The state of each LP contains two square matrices: an accumulation matrix A, and a transformation matrix T, each of size $n \times n$, where n is an integer constant chosen by the user. Each event message contains the transformation matrix of the sender, and upon execution of an event the receiving LP multiplies its accumulation matrix to the right with the received

transformation matrix. When an event is executed the receiving LP schedules a new event for a randomly selected LP at an exponentially distributed time delay.

At the end of the simulation, the matrices of all LP's are multiplied together, in LP ID (rank) order. The resulting matrix is the output of the simulation. Since matrix multiplication is in general non-commutative, the output depends on the individual events being executed in the correct order. The output serves as a check sum or hash of the simulation, and its size can be controlled by choosing the matrix size and the number of bits in the matrix elements.

The kernel of the event execution is a matrix multiplication, which (in the conventional implementation that we use) takes $O(n^3)$ arithmetic operations for $n \times n$ matrices. Reverse computation involves calculating a matrix inverse (or solving a matrix equation $A' = A \times T$ for A), which also requires $O(n^3)$ arithmetic operations. Each event or event message contains an $n \times n$ matrix and requires n^2 words of storage, and the same amount of data to be transmitted if communicated over a network. For bench-marking studies we can tune the ratio of arithmetic operations to memory/communication needs. This ratio is $O(n)$ for $n \times n$ matrices. We want to emphasize that this model is perfectly reversible, in the sense that no extra state besides the event itself is needed to undo the forward event: We simply invert the matrix in the event message and multiply the accumulation matrix to the right with this inverse.

We let the matrix elements be of a standard unsigned integral data type (e.g. 8, 16, 32, or 64 bits). For each of these types, the standard computer multiplication, addition, and subtraction perform arithmetic in an associated finite integer ring; Z_{2^k} where k is the number of bits in the data type, e.g. $k \in \{8, 16, 32, 64\}$. In these finite rings, all odd numbers have an inverse, and so half of the numbers in each ring can be used as denominators in division.

In this chapter we are interested in comparing different approaches to generate reversal of events to support roll-back. One of these approaches is reverse computation. In order for reverse computation to be applicable, events execution need to be reversible. To guarantee that, we select the transformation matrices to be non-singular over the integer ring of their elements. To simplify the expression of reversible multiplication, we additionally pick the transformation matrices so that Gaussian elimination can be completed successfully without pivoting.

3.1 Ring Inverses and Non-singular Matrices

The C++ language provides us with addition, subtraction, and multiplication in the relevant integer rings. We also need a division, which can be implemented as multiplication with the inverse. In order to find a ring inverse, we can use Euclid's extended algorithm. To be specific, we use the following implementation:

The function in Listing 1.1 returns the inverse of b if b is invertible in Z_{2^k}, otherwise it returns zero. We have the relation $b \equiv 1 \mod 2 \Rightarrow b * \text{intinv}(b) = b$.

```
myuint intinv(myuint b) {
  // Find inverse in integer ring of Z_{2^k}, where k is
  // the number of bits in the myuint data type. It is
  // expected that myuint is an unsigned integer type.
  myuint t0 = 0,t = 1,q,r;
  myuint a = 0; // Want initial a to be 2^k, which can not be
                // represented, so we use the lower order bits,
                // i.e. a = 0.

  if(b <= 1) return b;

  q = (~a) / b; // Surrogate for 2^k div b, where 'div'
                // is standard integer division (/). Unless
                // b is a power of 2, 2^k div b = = (2^k-1) div b.

  if(b*q+b = = 0) return 0; // Catches when b is power of 2.

  r = a - q*b;
  while(r > 0) {
    const myuint temp = t0 -q*t;
    t0 = t;
    t = temp;
    a = b;
    b = r;
    q = a/b;
    r = a - q*b;
  }
  if(b = = 1) return t;
  else return 0;
}
```

Listing 1.1. Computation of inverse in Z_{2^k}.

One might initially worry that it can be hard to find non-singular matrices over Z_{2^k}. It turns out that a significant fraction of such matrices where the elements are picked from a uniformly random distribution are non-singular. We can determine this as follows. First, a matrix is non-singular if and only if Gaussian elimination with row pivoting can be completed successfully. We note that since we work with a finite set of numbers (ring), there is no need to worry about stability – all calculations are exact and there are no round-off errors. Let M be an $n \times n$ matrix with elements independently selected uniformly from Z_{2^k}, where $k > 0$ is an integer. To perform Gaussian elimination on a M we first need to find a pivot element p in the first row. Any invertible element will do. The probability that we find one is $1 - \left(\frac{1}{2}\right)^n$. Assume p is in column j. Now swap column j and column 1. For all rows r and for all columns c in M, set $M'_{rc} = M_{rc} - M_{r1}p^{-1}M_{1c}$. Gaussian elimination proceeds by recursively performing elimination of the submatrix S of M' resulting from removing its first row and first column. For $r > 1$ and $c > 1$, the parity (oddness) of M'_{rc} is swapped if $M_{r1}M_{1c}$ is odd, and unchanged otherwise. The parity of M_{rc} is

uniformly random, and the parity of $M_{r1}M_{1c}$ is independent of M_{rc}. Therefore the parity of M'_{rc} is also uniformly random, since an independent flip does not change the distribution. By induction, the probability of finding a pivot element in S is $1 - \left(\frac{1}{2}\right)^{n-1}$, and carrying out the recursion to the end, yields the probability of M being non-singular to be

$$\prod_{i=1}^{n} \left(1 - \left(\frac{1}{2}\right)^{n}\right) \approx 0.288788\ldots.$$

This means that a little bit over one quarter of all uniformly random matrices over Z_{2^k} are non-singular. Therefore we can find suitable ones relatively efficiently by trial and error. Further, in order to create matrices for which we can do Gaussian elimination without pivoting, we pick a non-singular matrix T, and then permute the columns in the schedule dictated by the pivot columns given by computing Gaussian elimination with row pivoting on (a copy of) T.

4 Forward/Backward Code from Reversible Programs

The defining property of reversible programming languages is their forward and backward determinism, that is, in each computation state not only the successor state is uniquely defined, but also the predecessor state [8]. The computation is information preserving. In contrast, mainstream (irreversible) programming languages, such as C, are forward, but not backward deterministic.

In a reversible imperative programming language, such as Janus, every assignment statement is non-destructive, that is a *reversible update*, such as `x -= e`, where variable `x` may not occur in expression `e` on the right side (e.g., `x -= x` is not backward deterministic). In case of an assignment to an array element, for example `a[i,j] -= a[k,l]`, a runtime check ensures that `i` $\neq$ `k` or `j` $\neq$ `l`.

All control-flow statements, such as conditionals and loops, are equipped with assertions, in one way or another, to ensure their backward determinism. The variant of Janus used for the programs in this paper has a two-way deterministic loop **`iterate`** `i = e1` **`to`** `e2; s;` **`end`**, where neither the index variable `i` nor the variables occurring in expressions `e1` and `e2`, defining the start- and end-values of `i`, may be modified in the body statement `s`, which is executed once per iteration. Hence, the number of iterations is known before and after the loop.

An advantage of reversible programming languages is that their programs do not require instrumentation to restore a previous computation state from the current state, which is usually necessary in irreversible languages. Backward determinism opens new opportunities for program development because a procedure `p` cannot only be called by a usual **`call`** `p`, but its inverse semantics can be invoked by an **`uncall`** `p`. Forward and backward execution of a procedure are equally efficient, thus is makes no difference which direction is implemented in a program, which therefore is usually the one that is easier to write. We will make use of this possibility to reuse code by uncalling a procedure.

```
procedure crout(int LDU[][], int n)
  iterate int j = 0 to n-1
    iterate int i = j to n-1
      iterate int k = 0 to j-1
        LDU[i][j] -= LDU[i][k] * LDU[k][j]
      end
    end
    iterate int i = j+1 to n-1
      iterate int k = 0 to j-1
        LDU[j][i] -= LDU[j][k] * LDU[k][i]
      end
      uncall mult(LDU[j][i], LDU[j][j])
    end
  end
```

Listing 1.2. Janus implementation of the Crout matrix decomposition.

Translation from Janus to C++. Reversible programs can be translated to a mainstream (irreversible) programming language, which in this paper is C++. Usually, this requires the implementation of additional runtime checks in the target program to preserve the semantics of the source program. Assuming that the source program is correct and only applied to values for which it is well defined, the runtime checks in the target program can be turned off. The translation of Janus into C++ which we use for the benchmarks is straightforward, e.g., **iterate** is translated into a **for**-loop, and no further optimizations are performed by the Janus-to-C++ translator.

Only the translation of an **uncall** `p` requires an unconventional step in the translator, namely first the *program inversion* of procedure `p` into its inverse procedure `p-inv`, both `p` and `p-inv` written in Janus, followed by the translation of `p-inv` into the target language and the replacement of every **uncall** `p` by the functionally equivalent **call** `p-inv`. The target program then contains the C++ implementation of `p` and its inverse `p-inv`. Program inversion is straightforward in a reversible language (cf. [6]), e.g., a reversible assignment `x -= e` is inverted to `x += e` and a statement sequence is inverted to the reversed sequence of its inverted statements.

As a non-trivial example, Listing 1.2 shows the Janus implementation of the Crout algorithm for LDU matrix decomposition. The translation from Janus into C++ for the forward code is straightforward, and a **uncall** `mult` in Janus becomes a call to `mult-inv` in C++. To illustrate the generated inverted code, its C++ translation can be found in Listing 1.3. The iteration is translated into nested for-loops and the reversible assignment in Janus requires only a minor adaptation to the C++ syntax. In the C++ listing the `mult(a,b)` is effectively a standard integer product $a := a \times b$ with appropriate assertions that it can be inverted, i.e. the inverse of b exists. `mult-inv` uses `intinv` from Listing 1.1 to compute the ring inverse.

```
  template<typename myuint>
  void crout_inv(myuint *LDU, int &n) {
    for (int j = n - 1 ; j != 0 + 0 - 1 ; j += 0 - 1) {
        for (int i = n - 1 ; i != j + 1 + 0 - 1 ; i += 0 - 1) {
      mult(LDU[j*n+i], LDU[j*n+j]);
      for (int k = j - 1 ; k != 0 + 0 - 1 ; k += 0 - 1) {
            LDU[j*n+i] += LDU[j*n+k] * LDU[k*n+i];
          }
       }
       for (int i = n - 1 ; i != j + 0 - 1 ; i += 0 - 1) {
          for (int k = j - 1 ; k != 0 + 0 - 1 ; k += 0 - 1) {
            LDU[i*n+j] += LDU[i*n+k] * LDU[k*n+j];
          }
       }
    }
}
```

Listing 1.3. Reverse code of C++ translation of Listing 1.2.

```
procedure matrix_mult(int A[][], int B[][], int n)
  call crout(B, n)     // In-place LDU decomposition of B
  call multLD(A, B, n) // A := A*LD in place
  call multU(A, B, n)  // A := A*U in place
  uncall crout(B, n)   // Revert LDU decomposition to recover B
```

Listing 1.4. Janus implementation of matrix multiplication.

Matrix Multiplication in Janus. A conventional matrix-matrix multiplication needs temporary storage, and the individual steps are not reversible. Since a reversible language requires each operation to be reversible we need a different approach. One approach is to use LU or LDU decomposition, which can be performed in place, and is step-wise reversible. Multiplication with the resulting triangular matrices can also be done in-place and step-wise reversible. In the approach here, to compute $A := A \times B$, we perform the Crout algorithm for LDU decomposition, $B = L \times D \times U$ in place, then the sequence $A := A \times L$, $A := A \times D$, $A := A \times U$. Finally we reverse the LDU decomposition in place, to recover the original input B. For a Janus implementation of the in-place matrix multiplication, see Listing 1.4. The code for multiplication with triangular matrices is shown in Listing 1.5. This approach needs no temporary storage and is step-wise reversible. The price for this reversibility and in-place operation is more arithmetic operations than a standard matrix product by a factor of about 5/3 (for sufficiently large n, say $n > 10$). In the full implementation, we used a local temporary variable to reduce the number of calls to the ring-inverse function for speed optimization, since it is much more costly than a multiplication or addition. This does not change any of the reversibility features.

```
procedure multLD(int A[][], int LDU[][], int n)
  iterate int i = 0 to n-1
    iterate int j = 0 to n-1
      call mult(A[j][i], LDU[i][i])
      iterate int k = i+1 to n-1
        A[j][ i] += LDU[k][i] * A[j][ k]
      end
    end
  end

procedure multU(int A[][], int LDU[][], int n)
  iterate int i = n-1 by -1 to 0
    iterate int j = 0 to n-1
      iterate int k = 0 to i-1
        A[j][ i] += LDU[k][i] * A[j][ k]
      end
    end
  end
```

Listing 1.5. Janus implementation of in-place multiplication with triangular matrices. `multLD(A,LDU)` computes $A := A*(LD)$ and `multU(A,LDU)` computes $A := A \times U$.

5 Automatic Generation of Reversible Code for the Forward-Reverse-Commit Paradigm

In the forward-reverse-commit (FRC) paradigm [5] the original code is transformed such that during its forward execution it stores all information required to reverse all effects of the forward execution and restore the previous state of the program, or commit (possibly deferred) operations at a later point in time. Hence, we add the history of the computation to each saved state, which is usually called a Landauer's embedding. In both reverse and commit functions the additional information stored in the forward code is eventually disposed. Before that the reverse function uses the stored data to undo all memory modifying operations, in the commit function performs the deferred memory deallocation.

We generate transformed forward code to implement incremental state saving. The idea is to only store information about what changes in the program state because of a state transition, not the entire state. This approach is also briefly described in [5] for the programming language C (called "incremental check pointing" by the author). After performing a forward execution of the transformed program followed by a corresponding reverse operation, the program is restored to its original state, i.e. the exact same state as the original program was before performing any operation. Therefore, the execution of a forward function and a reverse operation is equivalent to executing no code (i.e. a no-op).

After performing a forward execution of the transformed program followed by a commit operation, the program is in the exact same state as executing the original program. Therefore, the execution of a forward function and its corresponding commit operation performs the same changes to the program state as the execution of the original function.

This transformation can also be considered to turn the program into a transactional program, where each execution step can be reversed (undone) or committed after which it cannot be reversed since all information necessary to reverse it is disposed by the commit operation. This is an important aspect when performing long running discrete event simulations: the forward-commit pairs ensures that no additional memory is consumed after a commit has been performed. As we shall see, the optimistic parallel discrete event simulation ensures that such a point in time at which all events can be committed up to a certain point in the past, can always be computed during the simulation.

In [9] we have shown how this approach can be extended to address C++ without templates. In [10] we have applied this approach to all of C++98, including templates and in [7] we have shown that this approach is general enough to be applied to C++11 standard containers and algorithms.

Our approach to generating reversible forward code introduces one additional function call, an instrumentation, for each memory modifying operation. Memory modifying operations are destructive assignments and memory allocation and deallocation. We only instrument operations of built-in types. For user-defined types either the existing user-provided assignment operator is instrumented (like any other code), or we generate a reversible default assignment operator if it is not user-provided. This is sufficient to cover all forms of memory modifying operations – of built-in types as well as user-defined types – because our runtime library that is linked with the instrumented code performs all necessary book-keeping at run-time. In particular, it also contains C++11 compile-time predicates. Those predicates check whether a provided type is a built-in type or a user-defined type and handle assignments of user-defined types (e.g. entire structs) as fall-through cases because they are handled component-wise by the respective overloaded assignment operator (which is either user-provided and automatically instrumented or generated). For a formal definition of the semantics of the instrumentations we refer the reader to [7].

We have implemented our approach in a tool called *Backstroke*[2] as source-to-source transformation based on the compiler infrastructure ROSE[3]. The Backstroke compiler for generating reversible programs from C++ was released to the public in March 2017 (version 2.1.0). This was the first public release of Backstroke V2 using incremental state saving.

[2] https://github.com/LLNL/backstroke.
[3] https://www.rosecompiler.org.

```
template<typename myuint>
void matmul(int n,myuint A[],myuint B[],myuint AB[]) {
  for(int i = 0; i<n; i++) {
    for(int j = 0; j<n; j++) {
      myuint s = 0;
      for(int k = 0; k<n; k++) {
        s =  s + A[i*n+k]*B[k*n+j];
      }
      AB[i*n+j] = s;
    }
  }
}
```

Listing 1.6. Original C++ Matrix Multiplication Code Fragment from the Benchmark.

```
template<typename myuint>
void matmul(int n,myuint A[],myuint B[],myuint AB[]) {
  for(int i = 0; i<n; i++)
    for(int j = 0; j<n; j++) {
      myuint s = 0;
      for(int k = 0; k<n; k++) {
        (xpdes::avpushT(s)) = s +A[i*n+k]*B[k*n+j];
      }
      (xpdes::avpushT(AB[i*n+j])) = s;

    }
}
```

Listing 1.7. Backstroke Generated Reversible C++ Forward Code (non-optimized).

5.1 Backstroke Instrumented Code

Three variants of the matrix multiplication are shown: (1) the original C++ code in Listing 1.6 for the matrix multiplication, (2) the non-optimized Backstroke generated code in Listing 1.7, and (3) the optimized Backstroke generated code in Listing 1.8. Backstroke's optimization detects local variables and ensures that direct accesses to local variables are not instrumented because those never need to be restored since memory for local variables is reserved on the runtime stack. Backstroke instrumented code records memory modifications only for heap allocated data since only this data persists across event function calls. In the presence of pointers the accesses to memory locations on the stack may be instrumented, but a runtime check in the Backstroke library ensures that only heap allocated data is stored.

This runtime check is always performed in the `xpdes::avpush` function because due to pointer aliasing, in general it is not known at compile time where

```
template<typename myuint>
void matmul(int n,myuint A[],myuint B[],myuint AB[]) {
  for(int i = 0; i<n; i++) {
    for(int j = 0; j<n; j++) {
      myuint s = 0;
      for(int k = 0; k<n; k++) {
          s =  s + A[i*n+k]*B[k*n+j];
      }
      (xpdes::avpushT(AB[i*n+j])) = s;
    }
  }
}
```

Listing 1.8. Backstroke Generated Reversible C++ Forward Code (automatically optimized).

the data that a pointer is referring to may be allocated. This check is performed based on the memory addresses of the argument passed to avpush and the stack boundaries determined as part of the initialization of the Backstroke runtime library.

In the presented model only C++ assignments are instrumented because no memory allocation happens in the event functions. The memory for the matrices is allocated in the initialization of the simulation, i.e. in the initialization function for each LP.

The `avpush` function passes a reference to the memory section denoted by the respective expression as argument and stores a pair of the address (of the denoted memory location) and the value at that address in a queue in the Backstroke runtime library. It returns the very same address such that the code can execute as usual and perform the write access. Consequently, `avpush` always stores the old value before the assignment happens. When a previous state needs to be restored, the reverse function simply iterates over all those address-value pairs stored by the `avpush` function and restores the memory locations at those addresses to the stored value. The `avpush` functions are strictly typed, and restoration follows in exact reverse order, which is important in case a memory location is written more than once or any forms of aliasing occur. For more details on the instrumentation functions we refer the reader to [7].

The difference of the non-optimized version to the optimized version is that the instrumentation in the innermost loop is not necessary because it is a write to a local variable `s`. In Listing 1.8 the innermost loop is not instrumented and therefore the number of instrumentations is only executed n^2 times where n is the size of the quadratic matrices. Without this optimization the Backstroke generated code would always be slower than the Janus generated code as we will discuss in more detail in Sect. 7. In general, accesses to memory which only holds temporary data, not defining the state of an LP, need not be instrumented.

The more precise a static analysis is that determines this property, the more instrumentations to temporary memory locations can be avoided.

Backstroke also offers program annotations (through pragmas) for users to manually minimize the number of instrumentations and interface functions to turn on/off the recording of data at runtime. For example, with this feature one can add conditions in loops to only record data in the very first iteration, but not in subsequent iterations that write to the same memory location. Alternatively, one can unroll a loop and only instrument the first (unrolled) iteration and exclude the remaining loop from instrumentation. Thus, with Backstroke one can also manually optimize the recording of data.

6 ROSS Simulator

For execution of our model codes we use the ROSS general purpose discrete event simulator, developed at RPI by C. Carothers et al. [11]. ROSS has been developed for more than a decade. It has the capability of running simulations both sequentially and in parallel using either the YAWNS conservative or Time Warp optimistic mechanism. Time Warp is an optimistic approach, where each processor employs speculative execution to process any event messages it is aware of. Causality conflicts, such as when a previously unknown message which should already have been processed is received, are handled through local roll back. During roll back the effects of messages that were processed in error are undone.

In order to use Time Warp in a ROSS model, a reverse event function must be provided, which is responsible for undoing the state changes that the forward event function incurred for the same event.

6.1 Adaptations of the ROSS Simulator for the FRC Paradigm

For our evaluation we are using the same ROSS implementation as in [7]. This version offers a commit method. Whenever an event is committed (during fossil collection) a commit function is called for the corresponding LP with the event as an argument. This is a time when non-reversible functions such as file I/O can be called safely. In particular, this is very useful for Backstroke, since commit time is the earliest known moment at which the state saved by the Backstroke instrumented forward code can be released, and memory deallocated by the forward event can be returned to the system. In addition to the commit methods, we extended ROSS to support a C++ class for the simulation time data structure, as opposed to the default double data type for representing time. This allows the sender to encode additional bits in the message timestamp to help with tie breaking of events.

7 Evaluation

We have evaluated the performance of three different implementations for the forward and reverse code of the matrix mode: Original code with hand written

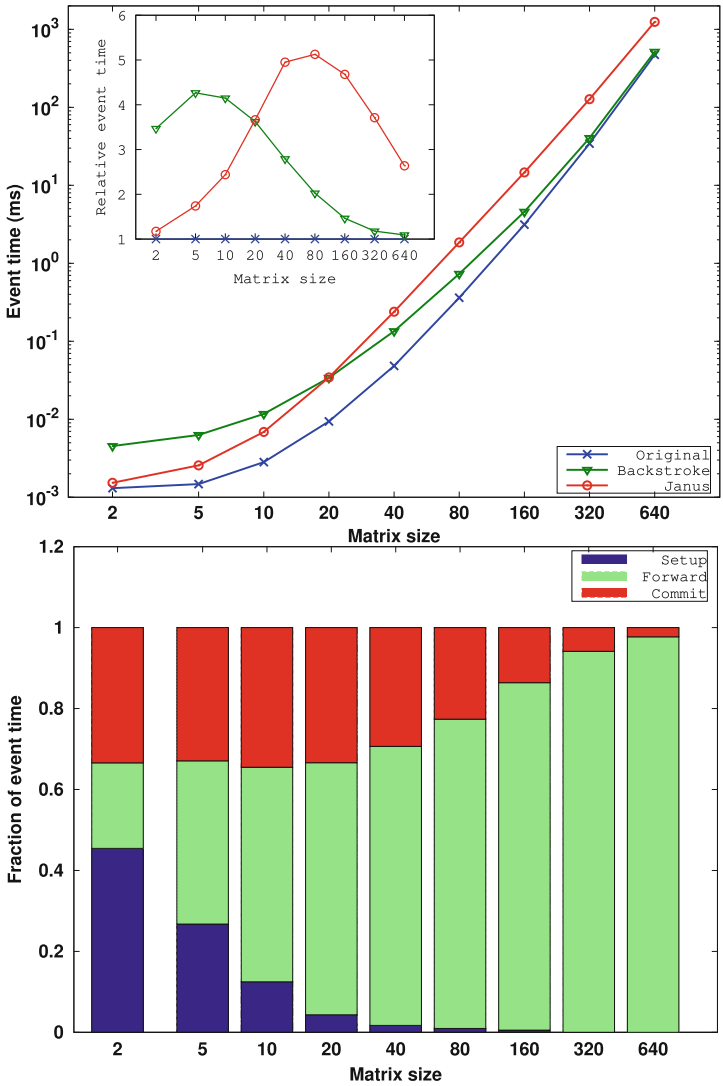


Fig. 1. *Top:* Performance of original, Backstroke, and Janus versions of the matrix model code. The graph shows the execution time per event for the three approaches. The inset shows execution time relative to the original code. *Bottom:* The time for the event function for the Backstroke code separated into event setup time, forward event time, and commit time costs.

reverse code, forward code implemented in Janus with reverse code generated by the Janus compiler, and forward code instrumented by Backstroke. For these performance evaluations we used the Backstroke code with local variable optimization.

First we focus on forward event code, which consists of three phases: event setup, forward computation, and commit. It is only the Backstroke instrumented code that has any significant work to perform in the setup and commit phases. We ran the matrix model sequentially using 8000 LP's and running up to 20 time units.

Figure 1 shows the matrix model performance as a function of matrix size for the four different reverse code approaches. The upper panel shows total event execution time, while the lower panel shows the relative cost of the three event execution phases for the Backstroke instrumented code.

The standard procedure, which we employ in the original code, for multiplying two $n \times n$ matrices performs n^3 multiplications and additions, and thus in general the execution time for an event should scale as $O(n^3)$ for sufficiently large n.

The Janus code must perform an LU factorization before carrying out the multiplications, and undo the factorization after the multiplication is complete. The total number of operations is about $\frac{5}{3}$ times as many as for the standard procedure. We can thus expect the Janus code to be almost twice as slow as the original code for large matrices. For very small matrices the number of operations of the Janus implementation is similar to the original code.

The Backstroke instrumented code with local variable optimization instruments $2n^2$ memory operations (n^2 for the matrix multiplication, and another n^2 for copying the result into the destination memory). Since there are $O(n^3)$ arithmetic operations, we expect the Backstroke instrumented code to incur negligible overhead for sufficiently large matrices.

We performed the runs using matrix sizes ranging from 2 to 640. The simulations were run on an cluster with Infiniband interconnect and 2.6GHz Intel Xeon E5-2670 cpus, 16 cores per node. We used the GNU g++ compiler with version 4.9.3, and the "-O3" optimization switches.

In the evaluation results we see that Janus performs best for small matrix sizes, whereas the Backstroke generated incremental state saving code performs better the larger the matrix size becomes, with a cross-over point at the size of a matrix size of 20 and for a matrix size of 640 the performance becomes almost the same as the non-instrumented version of the original forward code. The reason is that the Backstroke generated code only instruments those memory modifications that actually change the state of the simulation, i.e. elements in the matrix, whereas the computation of the intermediate results is not instrumented. This optimization is straightforward because this corresponds to not instrumenting accesses to local (stack-allocated) variables. Since optimistic PDES follows the forward-reverse-commit paradigm the trace only grows to a certain size, until the commit function is invoked by the simulator. The simulator guarantees that this happens in reasonable time intervals. The non-monotonic performance behavior for small matrices in Backstroke, and for intermediate size matrices in Janus (see inset in Fig. 1), is likely due to simulator and timing overhead, and cache effects, respectively.

The advantage of Janus generated forward/reverse code is that it does not need to store any additional data since the Janus implementation of the forward code is reversible. Saving memory is useful particularly in Time Warp simulations, since the amount of memory available dictates how much speculation can be performed. A challenge to implementing an algorithm in Janus is that it requires to writing assertions at the end of constructs that enable reverse execution to take the right execution path (i.e. reverse conditionals). In addition, reversibility may require algorithms that use inherently more operations than the most efficient ones available in traditional non-reversible computing.

8 Related Work

Jefferson started the subject of rollback-based synchronization in 1984 [3]. The paper discusses rollback implemented by restoring a snapshot of an old state, but today we are interested in using reverse computation and/or incremental state saving for that purpose. Also, that paper is written as if discrete event simulation is one of several applications of virtual time, but in fact it was then and is now the primary application. Although the term "virtual time" is used, you can safely read it as "simulation time".

In 1999 Carothers et al. published the first paper [4], that suggests using reverse computation instead of snapshot restoration as the mechanism for rollback, but it does not contemplate using a reversible language. It is written in terms of very simple and conventional programming constructs (C-like rather than C++ -like) and instrumenting the forward code to store near minimal trace information to allow rollback of side effects by reverse computation.

Barnes et al. demonstrated in 2013 [12], how important reverse computation can be in a practical application area. The fastest and most parallel discrete event simulation benchmark ever executed was done at LLNL on one of the world's largest supercomputers using reverse computation as its rollback method for synchronization. The reverse code was hand-generated, and methodologically we know that this is unsustainable. For practical applications we need a way of automatically generating reverse code from forward code, and this is what we address with the work presented in this paper - to have a tool available, Backstroke (version 2), for generating reverse code that can be applied to the full C++ language.

Kalyan Perumalla and Alfred Park discuss the use of Reverse Computation for scalable fault tolerant computations [13]. The paper is limited in a number of ways, but they make a fundamental point, which is that Reverse Computation can be used to recover from faults by mechanisms that are much faster than check pointing mechanisms.

In [14] Justin LaPre et al. discuss reverse code generation for PDES. The presented method is similar to one of our previous approaches in the work on Backstroke [15] as it takes control flow into account and generates code for computing additional information required to reconstruct the execution path that had been taken in the forward code. The approach we evaluate in this paper

is different as it does not need to take control flow information into account. Our initial discussion of incremental state saving was presented in [9], but was limited to C++ without templates. In this paper we evaluate a model that is implemented using C++ templates as well. The automatic optimization that we evaluate was also not present in [9].

An example for an optimistic PDES simulation with an automatically generated code using incremental state saving running thousands of LPs was published for a Kinetic Monte-Carlo model in [10]. In this crystal grain simulation, a piece of solid is modeled as a grid of unit elements. Each unit element represents a microscopic piece of material, big enough to be able to exhibit a well defined crystal orientation, but much smaller than typical grain sizes. These unit elements are commonly called spins, since the nature of grain evolution resembles evolution of magnetic domains. In the experiment the biggest model was run with a size of 768×768 spins divided into a grid of $96 \times 96 = 9216$ LPs with a slow-down factor in comparison to the hand-written reverse code of 4.7 to 4.3. In a new experiment presented in [7], the model was run at a much bigger scale with 1536×1536 spins in 256×256 logical processes (LPs) and implemented using C++ Standard containers and algorithms and user-defined types. After the transformation by Backstroke the model was run for 2 time units, or a total of 47633718 events on LLNL's IBM BlueGene/Q supercomputer with 16 cores per node, using up to 8192 cores. This version showed a penalty of 2.7. to 2.9 in comparison to the hand-written reverse code.

In [16] an autonomic system is presented that can utilize both an incremental and a full checkpointing mode. At run time both code variants are available and the system switches between the two variants, trying to select the more efficient checkpointing version. With our approach to incremental checkpointing we aim to reduce the number of instrumentations based on static analysis and offer a directive to the user for enabling or disabling the recording of data at runtime, allowing to also manually optimize instrumented code.

In [17] an instrumentation technique is applied to relocatable object files. Specifically, it operates on the Executable and Linkable Format (ELF). It uses the tool Hijacker [18] to instrument the binary code to generate a cache of disassembly information. This allows to avoid disassembly of instructions at run time. In contrast to our approach, the reverse instructions are built on-the-fly at runtime, and using pre-compiled tables of instructions. Similar to our approach there is also an overhead for each instrumentation. The information that it extracts from instructions, the target address and the size of a memory write, is similar to our address-value pairs. Recently progress has been made also in utilizing hardware transactional memory for further optimizing single node performance [19].

9 Conclusion

We have presented a new benchmark model for evaluating approaches to optimistic parallel discrete event simulation. We evaluated the performance of using

Janus generated forward/reverse code and incremental state saving (also called incremental checkpointing). The benchmark model has as its core operation a matrix multiplication.

From the results for our presented benchmark model we can conclude that depending on the matrix size either the Janus generated code or the Backstroke generated code performs best. Therefore, an implementation could include both codes and call the respective implementation dependent on the matrix size. If memory consumption becomes a limiting factor, the Janus implementation could be favored over the Backstroke implementation as well, since the Janus code does not store any additional data.

It also could be interesting to further explore how the Janus translator can be optimized and how this impacts the native C++ compiler. The Janus translator used in the benchmarks is non-optimizing, which means it implements every Janus statements in the target program, even when irreversible alternatives provide a faster implementation and some statements may be redundant in C++. Depending on the architecture, locality can be exploited to improve the runtime behavior, e.g., when translating summation `iterate ... A[i,j]+=e end` the use of a temporary variable in conventional assignments is an option: `s=A[i,j]; for ... s+=e end; A[i,j]=s;`. Some optimizations are performed by the native C++ compiler, others are better done by the Janus translator. Also, Janus may be extended with translator hints that allow a programmer to mark compute-uncompute pairs, which makes it easier to determine redundant statements.

Acknowledgments. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported by the LLNL-LDRD Program under Project No. 19-ERD-026. IM release number LLNL-BOOK-780059. The authors acknowledge the partial support of EU COST Action IC1405 on Reversible Computation—Extending Horizons of Computing.

References

1. Fujimoto, R.M.: Parallel and Distribution Simulation Systems, 1st edn. Wiley, New York (1999)
2. Omelchenko, Y., Karimabadi, H.: Hypers: A unidimensional asynchronous framework for multiscale hybrid simulations. J. Comp. Phys. **231**(4), 1766–1780 (2012)
3. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. **7**(3), 404–425 (1985)
4. Carothers, C.D., Perumalla, K.S., Fujimoto, R.M.: Efficient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul. **9**(3), 224–253 (1999)
5. Perumalla, K.S.: Introduction to Reversible Computing. CRC Press Book, Boca Raton (2013)
6. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Ramalingam, G., Visser, E. (eds.) Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, 15–16 January 2007, pp. 144–153. ACM (2007)

7. Schordan, M., Oppelstrup, T., Jefferson, D.R., Barnes Jr., P.D.: Generation of reversible C++ code for optimistic parallel discrete event simulation. New Generat. Comput. **36**(3), 257–280 (2018)
8. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible flowchart languages and the structured reversible program theorem. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_22
9. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 95–110. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_6
10. Schordan, M., Oppelstrup, T., Jefferson, D., Barnes, Jr., P.D., Quinlan, D.: Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In: Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. SIGSIM-PADS 2016, pp. 111–122. ACM (2016)
11. Holder, A.O., Carothers, C.D.: Analysis of time warp on a 32,768 processor IBM Blue Gene/L supercomputer. In: Bruzzone, A., Longo, F., Piera, M.A., Aguilar, R.M., Frydman, C. (eds.) Proceedings of the European Modeling and Simulation Symposium (EMSS), pp. 284–292 (2008)
12. Barnes, Jr., P.D., Carothers, C.D., Jefferson, D.R., LaPre, J.M.: Warp speed: executing time warp on 1,966,080 cores. In: Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. SIGSIM-PADS 2013, pp. 327–336. ACM (2013)
13. Perumalla, K.S., Park, A.J.: Reverse computation for rollback-based fault tolerance in large parallel systems. Cluster Comput. **17**(2), 303–313 (2013). https://doi.org/10.1007/s10586-013-0277-4
14. LaPre, J.M., Gonsiorowski, E.J., Carothers, C.D.: LORAIN: a step closer to the PDES "holy grail". In: Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. SIGSIM-PADS 2014, pp. 3–14. ACM (2014)
15. Vulov, G., Hou, C., Vuduc, R., Fujimoto, R., Quinlan, D., Jefferson, D.: The Backstroke framework for source level reverse computation applied to parallel discrete event simulation. In: Proceedings of the Winter Simulation Conference. WSC 2011, Winter Simulation Conference, pp. 2965–2979 (2011)
16. Pellegrini, A., Vitali, R., Quaglia, F.: Autonomic state management for optimistic simulation platforms. IEEE Trans. Parallel Distrib. Syst. **26**(6), 1560–1569 (2015)
17. Cingolani, D., Pellegrini, A., Quaglia, F.: Transparently mixing undo logs and software reversibility for state recovery in optimistic PDES. In: Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. SIGSIM PADS 2015, pp. 211–222. ACM (2015)
18. Pellegrini, A.: Hijacker: Efficient static software instrumentation with applications in high performance computing: poster paper. In: International Conference on High Performance Computing and Simulation (HPCS), pp. 650–655. (2013)
19. Santini, E., Ianni, M., Pellegrini, A., Quaglia, F.: Hardware-transactional-memory based speculative parallel discrete event simulation of very fine grain models. In: IEEE 22nd International Conference on High Performance Computing (HiPC), pp. 145–154 (2015)

Reversible Computation in Wireless Communications

Harun Siljak(✉)

CONNECT Centre, Trinity College, The University of Dublin, Dublin, Ireland
harun.siljak@tcd.ie

Abstract. This chapter presents pioneering work in applying reversible computation paradigms to wireless communications. These applications range from developing reversible hardware architectures for underwater acoustic communications to novel distributed optimisation procedures in large radio-frequency antenna arrays based on reversing Petri nets. Throughout the chapter, we discuss the rationale for introducing reversible computation in the domain of wireless communications, exploring the inherently reversible properties of communication channels and systems formed by devices in a wireless network.

1 Introduction

Wireless communication systems come in different shapes and sizes: from radio frequency (RF) systems we use in everyday life, to underwater acoustic communications (UAC) used where RF attenuation prevents use of radio communications. These two examples are of interest to this case study, as we explored the potential role of reversible computation in improving modern wireless communications in the RF and acoustic domains.

In the RF context, we examine the concept of distributed massive MIMO (multiple input multiple output) systems. The distributed massive MIMO paradigm will have an increasing relevance in fifth generation (5G) wireless systems and post-5G era, as it will allow formerly centralised base stations to operate as a group of hundreds (thousands) of small antennas distributed in space, serving many users by beamforming the signal to them, operating using distributed algorithms hence providing reduced power consumption and reduced computational overhead. Our aim is to explore the application of reversible computation paradigms in such systems to contribute in additional reduction of power consumption, but also to help in fault recovery and meaningful undoing of algorithmic steps in control and optimisation of such systems.

In the underwater acoustic context, we recognised the wave time reversal scheme as a physical example of reversibility, a physical method waiting for its reversible circuit implementation. The mechanism of wave time reversal is analogous to reversible computation as we know it, and as such it admits elegant and simple circuit implementation benefiting from all reversible computation advantages. With this inherent reversibility in mind, we take the question of wave

I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 208–221, 2020.
https://doi.org/10.1007/978-3-030-47361-7_10

time reversal in underwater conditions a step further, and ask about realistic models of such systems using reversible computation paradigms, and investigate the options of controlling the environment in which this process is used for communication.

Communication is inherently reversible: the communication channel changes direction all the time, with the transmitter and the receiver changing roles and transmitting through the same medium. Modulation and demodulation, coding and decoding all these processes aim for information conservation and reversibility. Hence the motivation for this study is clear: can reversible computation help in achieving goals of modern wireless communication: increasing access, decreasing latency and power consumption, minimising information losses?

In this chapter, we present results on optimisation schemes for massive MIMO based on reversing Petri nets, reversible hardware for wave time reversal, and some preliminary thoughts on our work in progress on modelling and control of wave time reversal in reversible cellular automata, as well as control of these automata in general.

2 Reversing Petri Nets and Massive MIMO

2.1 The Problem

In the distributed massive MIMO system described in the previous section, not all antennas need to be active at all times. Selecting a subset of antennas to operate at a particular time instant allows the system to retain advantages of a large antenna array, including interference suppression, spatial multiplexing and diversity [16] while reducing the number of radio frequency (RF) chains and the number of antennas to power [13]. The computational demand of optimal transmit antenna selection for large antenna arrays [11] makes it impractical, suggesting the necessity of suboptimal approaches. Traditionally, these approaches were centralised and based on the knowledge of the communication channel between every user and every antenna in the array; one widely used algorithm is the greedy algorithm [12] which operates iteratively by adding the antenna that increases the sum rate the most when joined with the set of already selected antennas. In decentralised algorithms similar procedures are conducted on much smaller subsets of antennas [21], leading to similar results in overall performance. Our approach here is decentralised, and it relies on Reversing Petri nets (RPN) [17] as the underlying paradigm. As this chapter focuses on applications, the reader interested in details about reversing Petri nets used in this example is advised to see [18]. The presentation here is based on [22].

The optimisation problem we are solving is downlink (transmit) antenna selection of N_{TS} antennas at the distributed massive MIMO base station with N_T antennas, in presence of N_R single antenna users. We maximise the sum-capacity

$$\mathcal{C} = \max_{\mathbf{P},\,\mathbf{H}_c} \log_2 \det \left(\mathbf{I} + \rho \frac{N_R}{N_{TS}} \mathbf{H}_c \mathbf{P} \mathbf{H}_c^H \right) \tag{1}$$

where ρ is the signal to noise ratio (SNR), $\mathbf{I}$ a $N_{TS} \times N_{TS}$ identity matrix, $\mathbf{P}$ a diagonal $N_R \times N_R$ power distribution matrix. $\mathbf{H}_c$ is the $N_{TS} \times N_R$ channel submatrix for a selected subset of antennas from the $N_T \times N_R$ channel matrix $\mathbf{H}$ [10].

In the case of receiver antenna selection, addition of any antenna to the set of selected antennas improves the overall sum-capacity, as its equivalent of Eq. (1) does not involve scaling by the number of selected antennas (i.e. there is not a power budget to be distributed over antennas in the receive case). This problem is submodular and has a guaranteed (suboptimal) performance bound for the previously described greedy algorithm. The greedy algorithm does not have performance bound for the transmitter antenna selection, as the case described by Eq. (1) does not fulfil the submodularity condition [24]; the addition of an antenna to the already selected set of antennas can decrease channel capacity.

As done in [21,24], we optimise (1) with two variables, the subset of selected antennas and the optimal power distribution over them successively: first, $\mathbf{P}$ is fixed to having all diagonal elements equal to $1/N_R$ (total power is equal to $\rho N_R/N_{TS}$), and after the antenna selection $\mathbf{P}$ is optimised by the water filling algorithm for zero forcing.

Figure 1 illustrates the proposed algorithm based on RPN: the antennas are Petri net *places* (circles A–G), with the *token* (bright circle) in a place indicating that the current state of the algorithm asks for that place (that antenna) to be on. The places are divided into overlapping *neighbourhoods* (N_1 and N_2 in our toy example) and each two adjacent places have a common neighbourhood. *Transitions* between places move tokens around based on the sum capacity calculations, with rules described below:

1. A transition is possible if there is a token in exactly one of the two places (e.g. B and G in Fig. 1) it connects. Otherwise (e.g. A and B, or E and F) it is not possible.
2. The enabled transition will occur if the sum capacity (1) calculated for all antennas with a token in the neighbourhood shared by the two places (for B and G, that is neighbourhood N_1) is less than the sum capacity calculated for the same neighbourhood, but with the token moved to the empty place (in case of B-G transition, this means $\mathcal{C}_{AB} < \mathcal{C}_{AG}$, sum-capacity of antennas A and B is smaller than that of A and G). Otherwise, it does not occur.
3. In case of several possible transitions from one place (A-E, A-D, A-C) the one with the greatest sum-capacity difference (i.e. improvement) has the priority.
4. There is no designated order in transition execution, and transitions are performed until a stable state is reached.

The algorithm starts from a configuration of n tokens in random places and converges to a stable final configuration in a small number (in our experiments, up to five) of iterations (passes) through the whole network. As the RPN conserves the number of tokens in the network, and our rules allow at most one token per place, the algorithm results in n selected antennas. Executing the algorithm on several RPNs in parallel (in our experiments, up to five) allows tokens to

traverse all parts of the network and find good configurations even with a relatively small number of antennas and users. The converged state of the RPN becomes the physical state of antennas: antennas with tokens are turned on for the duration of the coherence interval. At the next update of the channel state information, the algorithm proceeds from the current state.

The computational footprint of the described algorithm is very small: two small matrix multiplications and determinant calculations are performed at a node which contains a token in a small number of iterations. As such, this algorithm is significantly faster and computationally less demanding than the centralised greedy approach which is a low-complexity representative of global optimisation algorithms in antenna selection [11]. The worst case complexity of the RPN based approach is $\mathcal{O}(N_T^{\omega/a})$ (here, N_T denotes the number of antennas, and ω, $2 < \omega < 3$ is the exponent in the employed matrix multiplication algorithm complexity). The parameter a is related to the relative size of the neighbourhood as a reciprocal exponent, assuming that a neighbourhood of $N_T^{1/a}$, $a > 1$ suffices for RPN algorithm (as $\sqrt{N_T}$ suffices in our case, we went for $a = 2$). The constant factor multiplying the complexity is small because of few computing nodes (only those with tokens) and few iterations.

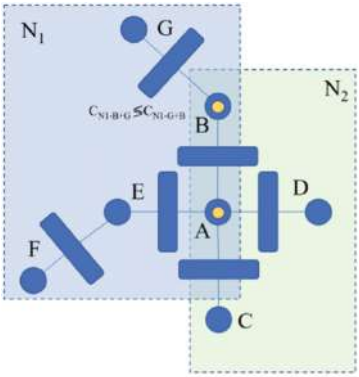
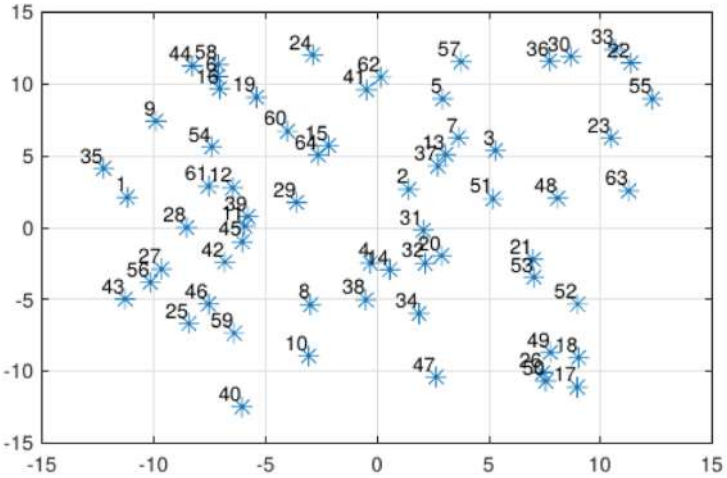


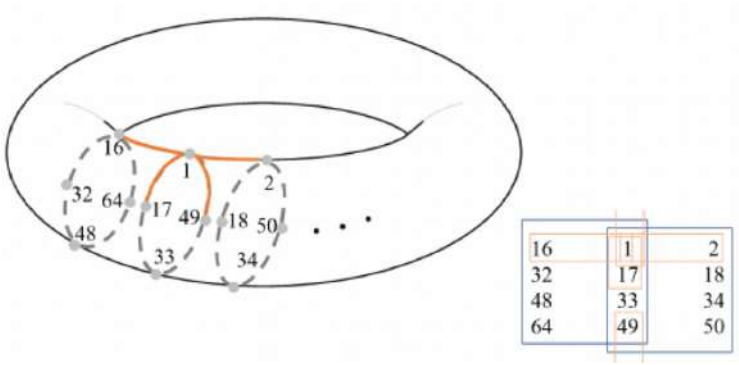
Fig. 1. A toy model of antenna selection on a reversing Petri net

2.2 Results and Discussion

The algorithm was tested using the raytracing Matlab tool Ilmprop [9] on a system composed of 64 omnidirectional antennas randomly distributed in space shown in Fig. 2(a). In all computations, channel state information (CSI) in matrix **H** was normalised to unit average energy over all antennas, users and subcarriers, following the practice from [10]. 75 randomly distributed scatterers and one large obstacle are placed in the area with the distributed base station. The number of (randomly distributed) users with omnidirectional antennas varied



(a) Randomly distributed antennas



(b) The mapping to RPN topology

Fig. 2. Antennas in physical and computational domain

from 4 to 16, and we used 300 OFDM (orthogonal frequency-division multiplexing) subcarriers, SNR $\rho = -5$ dB, 2.6 GHz carrier frequency, 20 MHz bandwidth. Antennas are computationally arranged in an 4×16 array folded into a toroid, creating a continuous infinite network, as shown in Fig. 2(b), e.g. antenna 1 is a direct neighbour of antennas 2, 16, 17 and 49. Immediate Von Neumann (top, down, left, right) neighbours can exchange tokens, and overlapping 8-antenna neighbourhoods are placed on the grid: e.g. for antenna 1, transitions to 16 and 17 are decided upon within the neighbourhood {16, 32, 48, 64, 1, 17, 33, 49} and the transitions to 2 and 49 are in {1, 17, 33, 49, 2, 18, 34, 50}. In Fig. 3 we compare greedy and random selection with two variants of our RPN approach: the average of five concurrently running RPNs, and the performance of the best RPN out of those five. The performance is comparable in all cases, and both

variants of our proposed algorithm tend to outperform the centralised approach as the number of users grows. This in practice means that a single RPN suffices for networks with a relatively large expected number of users.

The inherent reversibility of this problem and its solution generalises to the common problem of resource allocation in wireless networks, and sharing any pool of resources (power, frequency, etc.) can be handled between antennas (and antenna clusters) over a Reversing Petri Net. At the same time, such a solution would be robust to changes in the environment, potential faults, sudden changes in the mode of operation, and could operate on reversible hardware.

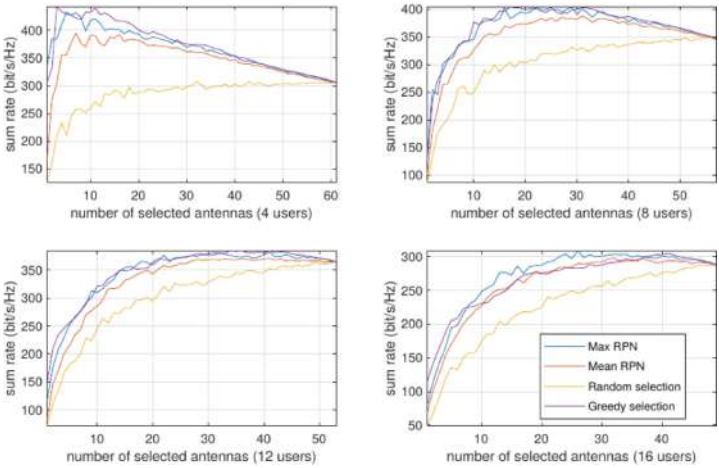


Fig. 3. Achieved sum rates for 4–16 users using the proposed algorithm vs random and centralised greedy selection

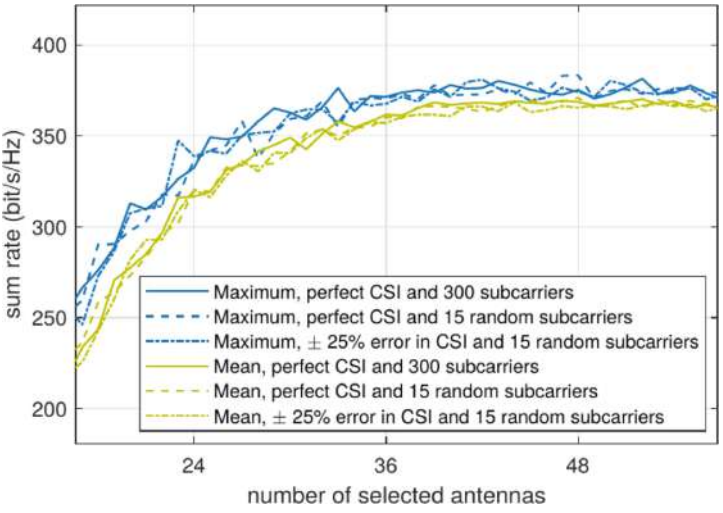


Fig. 4. The effects of imperfect CSI and random selection of subcarriers on optimisation

In [21], it has been shown that the distributed algorithms are resistant to errors in CSI and that they perform well even with just a (randomly selected) subset of subcarriers used for optimisation. Results in Fig. 4 in the case of 12 users confirm this for the RPN algorithm as well.

3 Reversible Hardware for Time Reversal

The technique called wave time reversal [6] has been introduced in acoustics almost three decades ago, and has since been applied to other waves as well–optical and RF. In our work, we focused on acoustic time reversal, thinking of its applications in acoustic underwater communications. However, it is worth noting that wave time reversal plays a significant role in RF communications as well–conjugate beamforming for MIMO systems is based on it. In the remainder of this section, we introduce the concept of wave time reversal and explain our proposed solution for its reversible hardware implementation. The presentation here follows the one in [20].

3.1 Wave Time Reversal

Time reversal mirrors (TRMs) [6] are based on emitter–receptor antennas positioned on an arbitrary enclosing surface. The wave is recorded, digitised, stored, time-reversed and rebroadcasted by the same antenna array. If the array on the boundary intercepts the entire forward wave with a good spatial sampling, it generates a perfect backward-propagating copy. The procedure begins when the source radiates a wave inside a volume surrounded by a two-dimensional surface with sensors (microphones) along the surface which record the field and its normal derivative until the field disappears (Fig. 5). When this recording is emitted back, it created the time-reversed field which looks like a convergent wavefield until it reaches the original source, but from that point it propagates as a diverging wavefield. This can be compensated by an active source at the focusing point cancelling the field, or a passive sink as a perfect absorber [3].

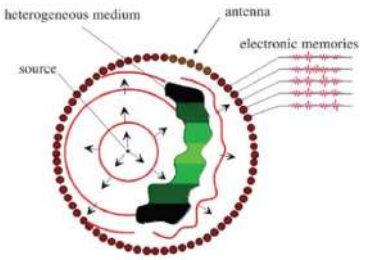


Fig. 5. A closed surface is filled with transducer elements [7]. The wavefront distorted by heterogeneities comes from a point source and is recorded on the cavity elements. The recorded signals are time-reversed and re-emitted by the elements. The time-reversed field back-propagates and refocuses exactly on the initial source.

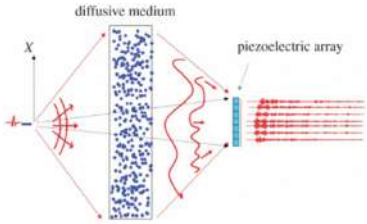


Fig. 6. Time-reversal experiment through a diffusive medium [7]

This description asks for the whole surface to be covered with the TRM transceivers, and for both the signal and the derivative to be stored: for practical purposes, less hardware-demanding solutions are needed. First, we note that the normal derivative of the field is proportional to the field in case the TRM is in the far field, halving the necessity for signal recording. Second, we note that a TRM can use complex environments to appear as an antenna wider than it is, resulting in a refocusing quality that does not depend on the TRM aperture [4]. Hence, it can be implemented with just a subset of transceivers located in one part of the boundary, as seen in Fig. 6.

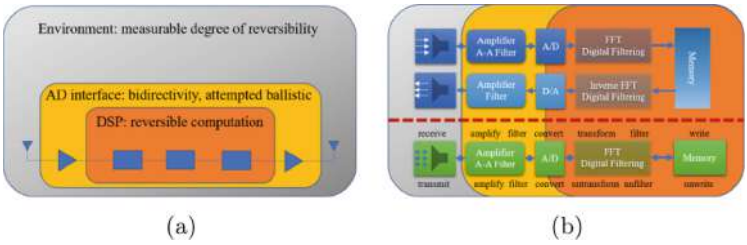


Fig. 7. (a) The three realms of reversibility, (b) The classical (top) and the reversible solution (bottom) for the classical time reversal chain

3.2 The Design

Figure 7 illustrates the challenge of designing a reversible hardware solution for a TRM:

1. The environment is reversible to an extent (we will return to this question later in this chapter). The physics of wave propagation in water is reversible, but issues arise as we lose information in the process.
2. The analog computation part of the TRM loses information due to filtering and analog-to-digital/digital-to-analog conversion (ADC/DAC), amplifiers accompanying the filters and the converters themselves, at the transition to the digital domain.

3. Finally, the digital computation part of the TRM is reversible and no increase in entropy is necessary: writing in memory and unwriting, in the fashion of Bennett's trick, enabling reuse of memory for the next incoming wave, while not increasing the entropy.

Analog Processing. The real amplifier is an imperfect device with a limited bandwidth, hence prone to losing signal information. By definition, it takes additional energy for the signal, so it asks for an additional power source. At the same time, the analog to digital and digital to analog converters both lose information because of the finite resolution in time and amplitude, preventing full reversibility. However, a single device can be both an ADC and a DAC depending on the direction [14]. In this solution, we assume bi-directional converters placed together with bi-directional amplifiers [14]. The conversion is additionally simplified in the one-bit solution [5] where the receivers at the mirror register only the sign of the waveform and the transmitters emit the reversed version based on this information. It is a special case of analog-to-digital and digital-to-analog conversion with single bit converters. The reduction in discretisation levels also means simplification of the processing chain and making its reversal (bi-directivity) even simpler. The question of the information loss is not straightforward: while the information about the incoming wave is lost in the conversion process (and the loss is maximal due to minimal resolution), spatial and temporal resolution are not significantly degraded. This scheme can also be called "one-trit" (trit is a ternary digit, analogous to a bit) reversal: there are three possible states in the practical implementation: positive pressure, negative pressure, and "off".

Digital Processing. The first, straightforward way of performing time reversal of a digitally sampled wave is storing it in memory and reading the samples in the reverse order (last in, first out, LIFO), analogous to storing the samples on the stack. The design of registers in reversible logic is a well-explored topic [15] and both serial and parallel reading/writing can be implemented. Design of latches in reversible logic is a well-studied problem with known solutions; a combination of latches makes a flip-flop, and a series of flip-flops makes a register (and a reversible address counter). In the case of wave time reversal, the recording of data is a large register being loaded serially with wave data. m bits from the ADC are memorised at the converter's sample rate inside a $k \times m$ bit register matrix (where k is the number of samples to be stored for time reversal). In the receiving process the bits are stored, in the transmission process they are unstored, returning the memory into the blank state it started from (uncomputation). We utilise Bennett's trick and lose information without the entropic penalty: the information is kept as long as it is relevant.

When additional signal processing, e.g. filtering or modulation is performed, it is convenient to reverse waves in the frequency domain: there, time domain reversal is achieved by phase conjugation, i.e. changing the sign of the signal's phase. The transition from the time to the frequency domain (and vice versa) in the digital domain is performed by the Fast Fourier Transform (FFT) and

its inverse counterpart, which are reversibly implementable [23]. The necessary phase conjugation is an arithmetic operation of sign reversal, again reversible. Any additional signal processing can be reversible as well: e.g. filter banks and wavelet transforms. These processes remain reversible with preservation of all components of signals [2].

Figure 8(a) gives a comparison of the bit erasures in different implementations of the digital circuitry: frequency domain (FFT) and time domain reversal performed by irreversible circuits, compared to reversible implementations. The number of erasures changes depending on two parameters: bit resolution of the ADC and the waiting time–the length of the interval in which samples are collected before reversal starts, equivalent to the number of digitised samples. The increase in both means additional memory locations and additional dissipation for irreversible circuits. The irreversible FFT implementation has an additional information loss caused by additional irreversible circuitry compared to the irreversible time domain implementation. Our implementation has no bit erasures whatsoever. The price that is paid reflects in the larger number of gates used in the circuit: the number of gates has only spatial consequences, information-related energy dissipation is zero thanks to information conservation.

On the other hand, Fig. 8(b) shows the information loss in the analog part of the system, and we differentiate two typical environments, the chaotic cavity and the complex (multiple scattering) medium. The chaotic cavity is an ergodic space with sensitive dependence on initial conditions for waves. In such an environment there is little to no loss in the information if the waiting time is long enough and the ADC resolution is high enough. In the complex media, the difference is caused by some of the wave components being reflected backwards by the scattering environment, hence not reaching the TRM. Again, more information is retained with the increase in the ADC resolution. However, as reported in [5], the information loss from low-resolution ADC use does not affect the performance of the algorithm. The analog part of the scheme remains a topic of our future work, as it leaves space for improvements of the scheme.

4 Reversible Environment Models and Control

Time reversal described in the previous section is an example of a reversible process in a nominally reversible environment. While dynamics of water subject to waves are inherently reversible, most of the sources of the water dynamics do not reverse naturally: e.g. the Gulf stream or a motion of a school of fish. Hence, even though it would rarely be completely reversed, the model for UAC should be reversible. We discuss the questions of reversible models following the exposition in [19], and the work in progress on control of reversible cellular automata (RCA).

RCA lattice gas models are cellular automata obeying the laws of fluid dynamics described by the Navier-Stokes equation. One such model, FHP (Frisch- Hasslacher-Pomeau) lattice gas [8] is simple and yet following the Navier-Stokes equations exactly. It is defined on a hexagonal grid with the rules of particle collision shown in Fig. 9. The FHP lattice gas provides us a two-dimensional

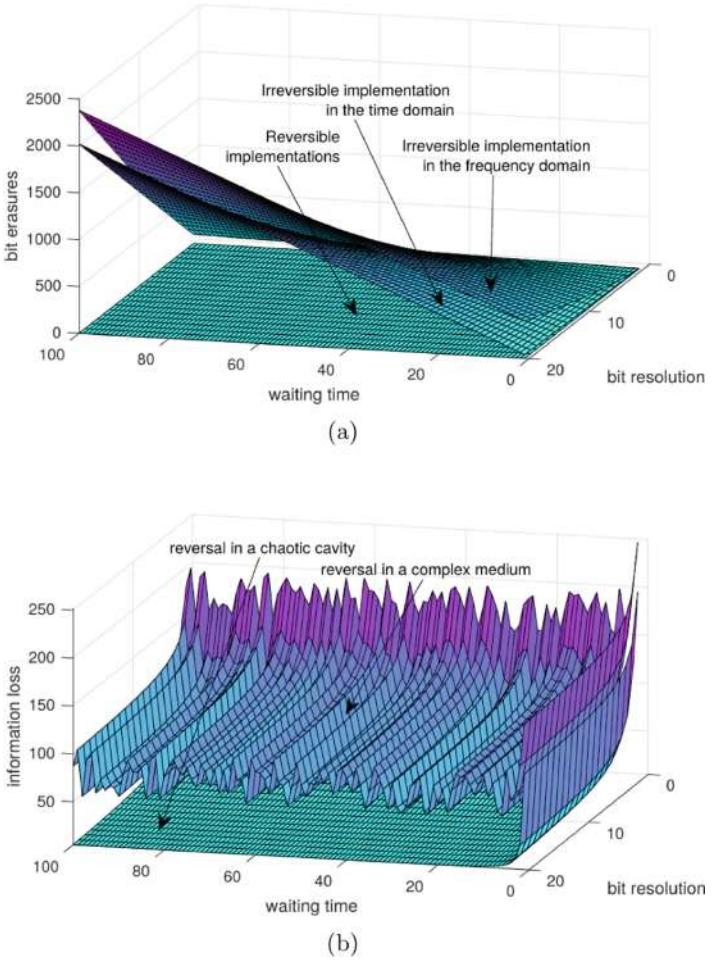


Fig. 8. Information loss in (a) the digital and (b) the analog part of the system. Units are omitted as the particular aspects of implementation are not relevant for the illustration of effects. Plot (a) is obtained by counting operations, plot (b) by simulation of back-scattering.

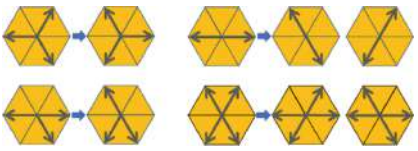


Fig. 9. FHP rules

model for UAC, easily implementable in software and capturing the necessary properties of the reversible medium.

Following the exposition in the previous section, we observe a model with an original source (transmitter) which causes the spread of an acoustic wave, the original sink (receiver) waiting for the wave to reach it, as well as scatterers and constant flows (streams) in the environment. The constant stream and the loss of information caused by some wave components never reaching the sink will result in an imperfect reversal at the original source. The measure of returned power gives us a directivity pattern (focal point). The amplitude of the peak will fluctuate based on the location of the original source and is a measure of reversibility, akin to fidelity or Loschmidt Echo. For us, it is a measure of the quality of communication, but in a more general context it can measure reversibility of a cellular automaton.

From the control viewpoint, it is interesting to ask the following: if a certain part of the environment is controllable (i.e. a number of cells of the RCA does not obey the rules of the RCA but allows external modification), how can it be used to achieve better time reversal? This is a compensation approach where we engineer the environment to compensate for effects caused by sources of disturbance out of our control. The approach we take is one of control of cellular automata [1], and it is expected that RCA are easier to control than regular CA, with easier search strategies and the ability to calculate control sequences.

5 Conclusions

In this chapter, we provided an overview of results obtained in the case study on reversible computation in wireless communications. Some of the presented work, such as optimisation in massive MIMO and reversible hardware for wave time reversal is finished and subject to further extensions and generalisations; other work, mainly the parts focused on RCA and modelling of reversible physics of communication, is still ongoing and more results are to come. This has been a pioneering study into reversibility in communications, and the results obtained promise a lot of space for improvement and applications in the future. We hope these efforts will serve as an inspiration and a trigger for the development of this field of research.

Acknowledgements. The work presented in this chapter was supported by the COST Association through the IC1405 Action on Reversible Computation, as well as a grant from Science Foundation Ireland (SFI) co-funded under the European Regional Development Fund under Grant Number 13/RC/2077 and European Union's Horizon 2020 programme under the Marie Skłodowska-Curie grant agreement No 713567. I am grateful to my collaborators, Prof Anna Philippou, Kyriaki Psara, Dr Julien de Rosny, Prof Mathias Fink, and Dr Franco Bagnoli for making this interdisciplinary research possible, and to Konstantin Popovic for the inspiring ideas.

References

1. Bagnoli, F., Rechtman, R., El Yacoubi, S.: Control of cellular automata. Phys. Rev. E **86**(6), 066201 (2012)
2. Chen, Y.-J., Amaratunga, K.S.: M-channel lifting factorization of perfect reconstruction filter banks and reversible M-band wavelet transforms. IEEE Trans. Circ. Syst. II Analog Digit. Signal Process. **50**(12), 963–976 (2003)
3. de Rosny, J., Fink, M.: Overcoming the diffraction limit in wave physics using a time-reversal mirror and a novel acoustic sink. Phys. Rev. Lett. **89**(12), 124301 (2002)
4. Derode, A., Roux, P., Fink, M.: Robust acoustic time reversal with high-order multiple scattering. Phys. Rev. Lett. **75**(23), 4206 (1995)
5. Derode, A., Tourin, A., Fink, M.: Ultrasonic pulse compression with one-bit time reversal through multiple scattering. J. Appl. Phys. **85**(9), 6343–6352 (1999)
6. Fink, M.: Time reversal of ultrasonic fields. I. Basic principles. IEEE Trans. Ultrason. Ferroelectr. Freq. Control **39**(5), 555–566 (1992)
7. Fink, M.: From Loschmidt daemons to time-reversed waves. Philos. Trans. Roy. Soc. A Math. Phys. Eng. Sci. **374**(2069), 20150156 (2016)
8. Frisch, U., Hasslacher, B., Pomeau, Y.: Lattice-gas automata for the Navier-Stokes equation. Phys. Rev. Lett. **56**(14), 1505 (1986)
9. Del Galdo, G., Haardt, M., Schneider, C.: Geometry-based channel modelling of MIMO channels in comparison with channel sounder measurements. Adv. Radio Sci. **2**(BC), 117–126 (2005)
10. Gao, X., Edfors, O., Tufvesson, F., Larsson, E.G.: Massive MIMO in real propagation environments: do all antennas contribute equally? IEEE Trans. Commun. **63**(11), 3917–3928 (2015)
11. Gao, Y., Vinck, H., Kaiser, T.: Massive MIMO antenna selection: switching architectures, capacity bounds, and optimal antenna selection algorithms. IEEE Trans. Signal Process. **66**(5), 1346–1360 (2017)
12. Gharavi-Alkhansari, M., Gershman, A.B.: Fast antenna subset selection in MIMO systems. IEEE Trans. Signal Process. **52**(2), 339–347 (2004)
13. Hoydis, J., Ten Brink, S., Debbah, M.: Massive MIMO in the UL/DL of cellular networks: how many antennas do we need? IEEE J. Sel. Areas Commun. **31**, 160–171 (2013)
14. Mirmotahari, O., Berg, Y.: Pseudo floating-gate and reverse signal flow. In: Recent Advances in Technologies. IntechOpen (2009)
15. Nayeem, N.M., Hossain, M.A., Jamal, L., Babu, H.M.H.: Efficient design of shift registers using reversible logic. In: 2009 International Conference on Signal Processing Systems, pp. 474–478. IEEE (2009)
16. Ozgur, A., Lévêque, O., Tse, D.: Spatial degrees of freedom of large distributed MIMO systems and wireless ad hoc networks. IEEE J. Sel. Areas Commun. **31**(2), 202–214 (2013)
17. Philippou, A., Psara, K.: Reversible computation in petri nets. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 84–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_6
18. Philippou, A., Psara, K., Siljak, H.: Controlling reversibility in reversing petri nets with application to wireless communications. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 238–245. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_15

19. Siljak, H.: Reversibility in space, time, and computation: the case of underwater acoustic communications. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 346–352. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_25
20. Siljak, H., de Rosny, J., Fink, M.: Reversible hardware for acoustic communications. IEEE Commun. Mag. **58**, 55–61 (2020)
21. Siljak, H., Macaluso, I., Marchetti, N.: Distributing complexity: a new approach to antenna selection for distributed massive MIMO. IEEE Wireless Commun. Lett. **7**(6), 902–905 (2018)
22. Siljak, H., Psara, K., Philippou, A.: Distributed antenna selection for massive MIMO using reversing Petri nets. IEEE Wireless Commun. Lett. **8**(5), 1427–1430 (2019)
23. Skoneczny, M., Van Rentergem, Y., De Vos, A.: Reversible Fourier transform chip. In: 2008 15th International Conference on Mixed Design of Integrated Circuits and Systems, pp. 281–286. IEEE (2008)
24. Vaze, R., Ganapathy, H.: Sub-modularity and antenna selection in MIMO systems. IEEE Commun. Lett. **16**(9), 1446–1449 (2012)

Error Reconciliation in Quantum Key Distribution Protocols

Miralem Mehic[1,3(✉)], Marcin Niemiec[2,3], Harun Siljak[4], and Miroslav Voznak[3]

[1] Department of Telecommunications, Faculty of Electrical Engineering, University of Sarajevo, Zmaja od Bosne bb, Kampus Univerziteta, 71000 Sarajevo, Bosnia and Herzegovina
miralem.mehic@ieee.org
[2] AGH University of Science and Technology, al. Mickiewicza 30, 30-059 Krakow, Poland
[3] Department of Telecommunications, VSB-Technical University of Ostrava, 17. listopadu 15, 70800 Ostrava-Poruba, Czech Republic
[4] CONNECT Centre, Trinity College Dublin, Dunlop Oriel House 34 Westland Row, Dublin 2, Ireland

Abstract. Quantum Key Distribution (QKD) protocols allow the establishment of symmetric cryptographic keys up to a limited distance at limited rates. Due to optical misalignment, noise in quantum detectors, disturbance of the quantum channel or eavesdropping, an error key reconciliation technique is required to eliminate errors. This chapter analyses different key reconciliation techniques with a focus on communication and computing performance. We also briefly describe a new approach to key reconciliation techniques based on artificial neural networks.

Keywords: Error reconciliation · Quantum key distribution · Performances · Reversibility

1 Introduction

QKD provides an effective solution for resolving the cryptographic key establishment problem by relying on the laws of quantum physics. Unlike approaches based on mathematical constraints whose security depends on the attacker's computational and communication resources, QKD does not put a limit on the available resources but limits the length of the link implementation [1]. A QKD link can be realized only to a certain distance and at certain rates since it involves usage of two channels: quantum/optical and public/classical.

This work has been partially supported by COST Action IC1405 on Reversible Computation - Extending Horizons of Computing, and partly by the European Union's Horizon 2020 Research and Innovation Programme, under Grant Agreement no. 830943, the ECHO project. This work was also supported by the Ministry of Education, Science and Youth of Canton Sarajevo, Bosnia and Herzegovina under Grant No. 11/05-14-27719-1/19 and partly by the Horizon 2020 project OpenQKD under grant agreement No. 857156.

I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 222–236, 2020.
https://doi.org/10.1007/978-3-030-47361-7_11

Quantum cryptography focuses on photons (particles of light), using some of their properties to act as an information carrier. Principally, information is encoded in a photon's polarization; a single polarized photon is referred to as a qubit (quantum bit) which cannot be split, copied or amplified without introducing detectable disturbances.

The procedure for establishing a key is defined by QKD protocol, and three basic categories are distinguished: the oldest and widespread group of discrete-variable protocols (BB84, B92, E91, SARG04), efficient continuous-variable (CV-QKD) protocols and distributed-phase-reference coding (COW, DPS) [2,3]. The primary difference between these categories is reflected in the method of preparing and generating photons over a quantum channel [4–6].

A quantum channel is used only to exchange qubits, and it provides the QKD protocol with raw keys. All further communication is performed over a public channel, and it is often denoted as post-processing. It includes steps that need to be implemented for all types of protocols [2], exchanging only the accompanying information that helps in the profiling of raw keys. The overall process is aimed at establishing symmetric keys on both sides of the link in a safe manner.

The initial post-processing step is called a sifting phase, and it is used to detect those qubits for which adequate polarization measurement bases have been used on both sides. Therefore, user B, typically designated Bob informs user A, usually named Alice in literature, about bases he used, and Alice provides feedback advising when incompatible measurement bases have been used. It is important to underline that information about the measurement results is not revealed since only details on used bases are exchanged. Bob will discard bits for cases when incompatible bases have been used, providing the sifted key.

Further, it is necessary to check whether the eavesdropping of communication has been performed. This step is known as error-rate estimation since it is used to estimate the overall communication error. The eavesdropper is not solely responsible for errors in the quantum channel since errors may occur due to imperfection in the state preparation procedure at the source, polarization reference frame misalignment, imperfect polarizing beam splitters, detector dark counts, stray background light, noise in the detectors or disturbance of the quantum channel. However, the threshold of bit error rate p_{max} for the quantum channel without the presence of eavesdropper Eve is known in advance, and this information can be compared with the measured quantum bit error rate (QBER) p of the channel. The usual approach for estimation of the QBER in the channel (p) is to compare a small sample portion of measured values. The selected portion should be sufficient to make the estimated QBER credible where the question about the length of the sample portion is vital [4,7,8]. After estimating QBER, the obtained value can be compared with the already known threshold value of p_{max}. If the error rate is higher than a given threshold ($p > p_{max}$), the presence of Eve is revealed which means that all measured values should be discarded and the process starts from the beginning. Otherwise, the process continues.

Although the estimated value is lower than the threshold value, there are still measurement errors that need to be identified, and those bits need to be corrected or discarded. The process of locating and removing errors is often denoted as "error key reconciliation". As shown in traffic analysis experiments [9, 10], error key reconciliation represents a highly time demanding and extensive computational part of the whole process. Depending on the implementation, a key reconciliation step may affect the quantum channel and considerably impact the key generation rate.

In the following sections, we analyze the most popular error reconciliation approaches. Cascade protocol is discussed in Sect. 2, overview of Winnow protocol is given in Sect. 3. Section 4 outlines LDPC approach while the comparison is given in Sect. 5. We introduce the new key reconciliation protocol in Sect. 6 and provide conclusion in Sect. 7.

2 Cascade

The most widely used error key reconciliation protocol is cascade protocol due to its simplicity and efficiency [11]. Cascade is based on iterations where random permutations are performed with the aim of evenly dispersing errors throughout the sifted key. The permuted sifted key is divided into equal blocks of k_i bits, and after each iteration and new permutations, the block size is doubled: $k_i = 2 \cdot k_{i-1}$. The results of the parity test for each block are compared, and a binary search to find and correct errors in the block is performed. However, to improve the efficiency of the process, the cascade protocol investigates errors in pairs of iterations in a recursive way.

Instead of rejecting error bits in the first stage, information about the presence of an error bit in the block is used in the further iterations to detect errors that have not been detected due to the measurement parity. For any error detected in further iterations, at least one matching error can be identified in the same block of the previous iteration which was previously considered as a block without errors. Using a binary search, a deep search for errors in such a block is performed, and the masked errors can be recursively detected. Two passes of cascade protocol are illustrated in Fig. 1.

The length of the initial block k_1 is a critical parameter which depends on the estimated QBER. The empirical analysis described in [11] proposes the use of value $k_1 = 0.73/p$ as the optimal value, where p is the estimated QBER. Sugimoto modified the cascade protocol to bring the cascading protocol closer to theoretical limits [12]. Besides, he confirmed that four iterations are sufficient for the effective key reconciliation as originally proposed in [11]. However, due to the dependence of the initial block's length on the estimated QBER, it is advisable to execute all the iterations (as long as the length of the block k_i is not equal to the length of the key). In [4], Rass and Kollmitzer showed that adopting block-size to variations of the local error rate is worthwhile, as the efficiency of error correction can be increased by reducing the number of bits revealed to an adversary [13].

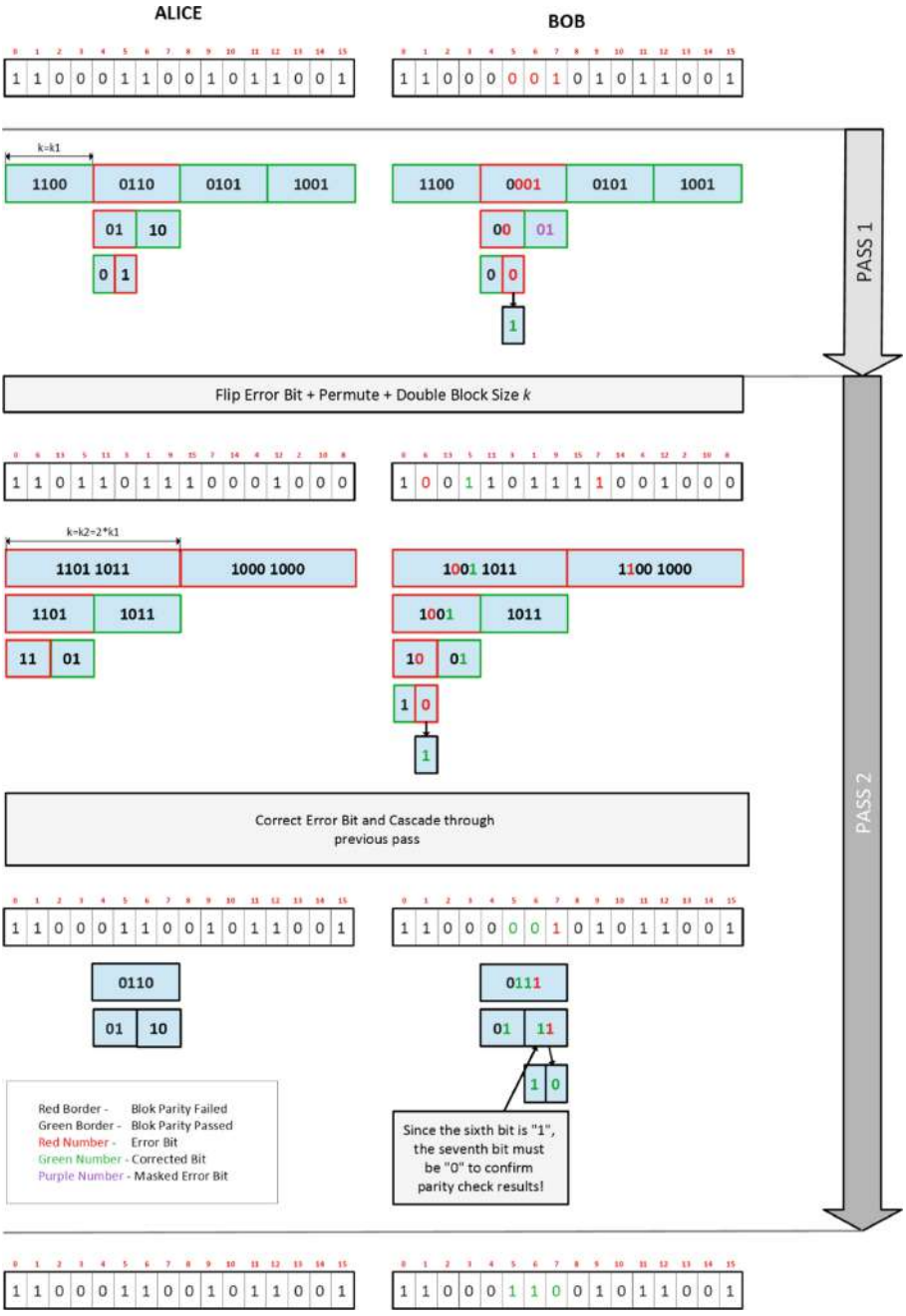


Fig. 1. Illustration of the first two passes of reconciliation using a Cascade protocol.

Cascade protocol relies on the use of the binary search to locate an error bit. The binary search includes further division of the block into two smaller subblocks for which the results of parity check values are compared until an error is found. For each block with an error bit, in total $1 + \lceil \log_2 k_i \rceil$ parity values are exchanged since $1 + \lceil \log_2 k_i \rceil$ is the maximum number of times that block k_i can be splitted, and only one parity value is exchanged for blocks without errors.

In addition to discarding the sample portion bits used to estimate QBER value, it is advised to discard the last bit of each block and subblock for which the parity bit was exchanged to minimize the amount of information gained by Eve. The maximum number of discared bits denoted as D_i can be calculated based on k_i value in the i^{th} iteration as follows:

$$\sum D_i = \sum_i \Big(\sum_{\substack{initially \\ even \\ blocks}} 1 + \sum_{\substack{initially \\ odd \\ blocks}} (1 + \lceil log_2\ k_i \rceil) + \sum_{\substack{other \\ errors \\ corrected}} \lceil log_2\ k_i \rceil \Big) \quad (1)$$

As proposed in [14], Eq. (1) can be shortened to:

$$D = \sum D_i = \sum_i \Big(\frac{n}{k_i} + \sum_{\substack{errors \\ corrected}} \lceil log_2\ k_i \rceil \Big) \quad (2)$$

where $k_i = 2 \cdot k_{i-1}, k_i < \frac{n}{2}$ and n denotes the amount of the measured values in sifting phase. The number of discarded bits depends on the QBER value and initial block size. However, Sugimoto showed [12] that most errors are corrected in the first two iterations. The empirical analysis of cascade protocol is given in [15], while the practical impact of cascade protocols on post-processing is considered in [9,16]. In [17], Chen proposed the extension of random permutations using interleaving technique optimized to reduce or eliminate error clusters from burst errors. Nguyen proposed modifying the permutation method used in cascade [18]. Yan and Martinez proposed modifications based the initial key's length in [19,20] while the use of Forward Error Correction was analyzed in [21] (Table 1).

Table 1. Error correction per passes using Cascade protocol

Iteration	1	2	3	4
Corrected errors (%)	54.522%	45.347%	0.451%	0.002%

3 Winnow

In 2003, Winnow protocol based on Hamming codes was introduced [22]. The aim was to increase the throughput and reduce the interactivity of Cascade by eliminating the binary search step.

Both parties, Alice and Bob, divide their random keys M_a and M_b into blocks of equal length (recommended starting size is $k = 8$) and calculate syndrome values S_a and S_b based on a Generator matrix G and a parity check Matrix H where $H \cdot G^T = 0$. For each block of size k, based on his key values M_b, Bob will generate and transmit his syndrome $S_b = H \cdot M_b$ to Alice, which will calculate the syndrome differences S_d. If S_d is non-zero, Alice will attempt to correct the errors with the fewest changes leading to syndrome zero values.

$$S_a = H \cdot M_a^T = \begin{bmatrix} 0\,1\,1\,0\,1\,0\,1 \\ 1\,0\,1\,1\,0\,1\,0 \\ 1\,1\,0\,0\,1\,0\,1 \end{bmatrix} \cdot \begin{bmatrix} 0\,1\,1\,0\,0\,1\,1 \end{bmatrix}^T = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

$$S_b = H \cdot M_b^T = \begin{bmatrix} 0\,1\,1\,0\,1\,0\,1 \\ 1\,0\,1\,1\,0\,1\,0 \\ 1\,1\,0\,0\,1\,0\,1 \end{bmatrix} \cdot \begin{bmatrix} 0\,1\,0\,0\,0\,1\,1 \end{bmatrix}^T = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$S_a \bigotimes S_b = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

$$1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 = 3 \ (\textit{bit on position } 3 \textit{ is the error}) \tag{3}$$

The Hamming distance d_{min} between codewords limits the number of errors that are suitable for correction where a code word with the number of errors greater than $\frac{d_{min}}{2}$ may closely resemble different code word then correcting the considered code word. Due to reliance on Hamming codes, the Winnow protocol may actually introduce errors, which is the main disadvantage of the shortly described approached. Its efficiency is lower when compared to Cascade for QBER values below 10% that are useful for practical QKD [23].

To achieve information-theoretical secrecy, Buttler suggested discarding an additional bit of each block of size k in the privacy maintenance phase [22].

4 Low Density Parity Check

With terrestrial links, Alice and Bob are usually not limited to execution time, computation and communication complexity. However, with satellite links, the parties need to consider significant losses in the channel, limited time to establish a key due to periodic satellite passage where communication and computation complexity puts additional constraint. Therefore, in previous years, researchers have turning to the application of Gallager's Low Density Parity Check (LDPC) codes that have recently been shown to reconcile errors at rates higher than those of Cascade and Winnow [24–26]. LDPC provides low communication overhead and inherent asymmetry in the amount of computation power required at each side of the channel.

LDPC linear codes are based on a parity check matrix H and a generator matrix G where a decoding limit of the code is defined with the minimum distance. The dimensions of H and G are $m \times n$ where $m = n \cdot (1-r)$ and r is defined

as code rate in range [0, 1]. The code rate value is usually defined beforehand; it defines the correcting power and efficiency. The reconciliation algorithm based on LDPC includes following steps:

- An estimation of QBER of the communication channel is performed,
- Based on estimated QBER, Alice and Bob choose the same $m \times n$ generator matrix G and parity check matrix H,
- For each sifted key, Bob calculates syndrome S_b and send it to Alice,
- Alice attempts to reconcile sifted key, assuming that Bob has the correct sifted key. Her goal is to resolve Bob's key vector x, based on her key vector y, received syndrome S_b, the parity-check matrix H, and estimated QBER value. Alice can use several techniques to decode LDPC such as belief propagation decoding algorithm (also known as the Sum-Product algorithm) or Log-Likelihood Ratios which significantly lower computational complexity [4,16,23].

Decoding LDPC code requires larger computational and memory requirements than either the Cascade or Winnow algorithms. However, it has a significant advantage due to the reduction of communication resources since only one information exchange is required. In networks with limited resources (bandwidth and latency), such tradeoff provides potentially large gains in overall runtime and secrecy. In the context of QKD, LDPC was firstly used as a base for the BBN Niagara protocol in DARPA QKD network [27].

5 Comparisson

For testing purposes, Cascade, Winnow and LDPC code were implemented in C++ programming language on servers Intel (R) Xeon (R) Silver 4116 CPU @ 2.10 GHz with 8 GB, and 512 GB HDD. For each value of QBER, 10.000 random keys were tested with the same random seed, which allowed repeating scenarios for different protocols used (Cascade, Winnow and LDPC). In total, 870,000 tests were performed.

The total number of leaked bits is defined as follows:

- Cascade: For each exchange of parity value, one bit is discarded.
- Winnow: For each block k, one bit is discarded.
- LDPC: Total length of syndrome S_b value exchanged.

Figure 2 shows that for small values of QBER (up to 0.05%), Cascade quickly finds and removes errors resulting in a small number of iterations. However, as the QBER value increases (up to 0.10%), LDPC shows better efficiency in terms of overhead and information exchanged.

Figure 3 shows that the overhead efficiency has its price in terms of execution time. Due to the simplicity of algorithms, Cascade and Winnow codes have almost fixed execution time, while in LDPC, the code execution time varies, and gradually increases with the QBER increase.

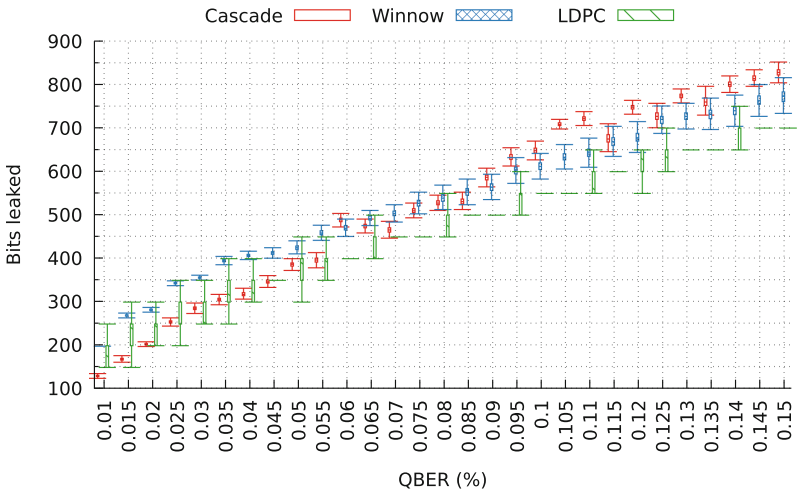


Fig. 2. The number of bits leaked (discarded) for different QBER values. Due to its simplicity, the binary search within Cascade protocol can locate errors in a short time for lower values of QBER. However, for more significant QBER values, binary search requires deeper checking of the sifted key, which increases communication. In the case of Winnow, syndrome message per each block of length k is exchanged which can be used to detect errors in early stages.

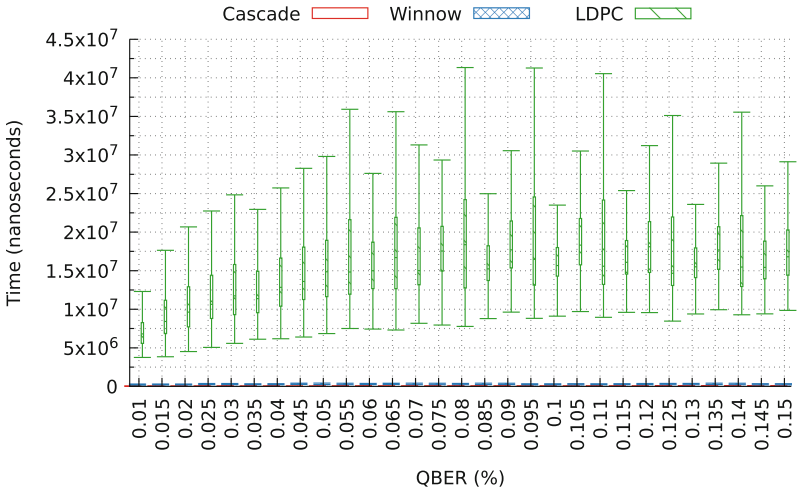


Fig. 3. The execution time for different QBER values. LDPC predominantly requires more time to execute key reconciliation tasks while due to its simplicity, the execution time of the Cascade and Winnow protocols is almost constant. LDPC based on the belief propagation algorithm was used for decoding.

6 Error Correction Based on Artificial Neural Networks

Using artificial neural networks for error correction during a key reconciliation process is a new concept, introduced in [28]. This proposal assumes the use of mutual synchronization of artificial neural networks to correct errors occurring during transmission in the quantum channel. Alice and Bob create their own neural networks based on their keys (with errors). After the mutual learning process, they correct all errors and can use the final key for cryptography purposes.

6.1 Tree Parity Machines

Tree parity machine (TPM) is a type of artificial neural networks (ANN) – a family of statistical learning models inspired by biological neural networks [29]. It consists of artificial neurons (analogous to biological neurons) which are connected and are able to transmit a signal from one neuron to another [30]. Neurons are usually organized in layers: the first layer consists of input neurons which can send the data to the second layer (called hidden). The last layer – called the output layer – consists of output neurons. TPM contains only one hidden layer and has a single neuron in the output layer. It consists of KN input neurons, where K is the number of neurons in the hidden layer and N is the number of inputs into each neuron in the hidden layer. An example of TPM is presented in Fig. 4.

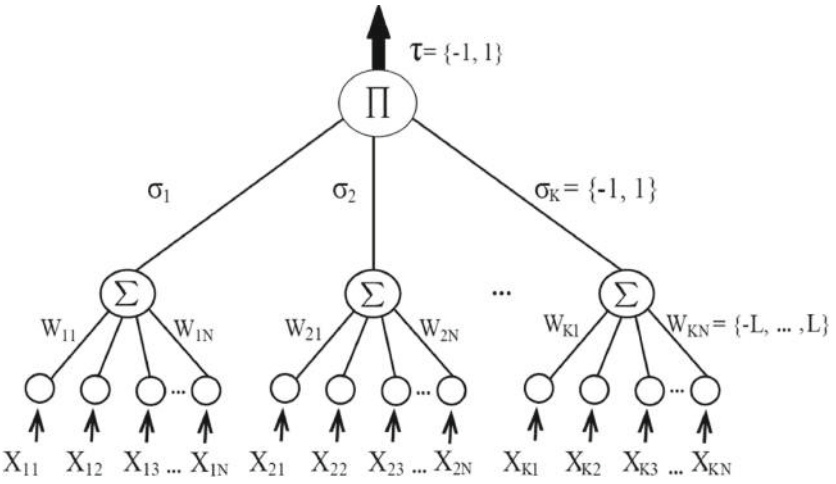


Fig. 4. Structure of TPM machine [28]

TPMs have another important feature: connections between neurons can store parameters (called weights) that can be manipulated during calculation.

Each connection between the input layer and hidden layer is characterized by its weight, which is an integer from the range $[-L, L]$. The output value of neuron k in the hidden layer depends on input x and weight w and is calculated as:

$$\sigma_k = sgn(\sum_{n=1}^{N} x_{kn} * w_{kn}) \tag{4}$$

where signum function is:

$$sgn(z) = \begin{cases} -1 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{5}$$

The output value of the neuron in the output layer is calculated as:

$$\tau = \prod_{k=1}^{K} \sigma_k \tag{6}$$

When Alice and Bob build their own TPMs with the same structure (K, N and L), they can synchronize these artificial networks after mutual learning [31]. At the beginning of this process, each TPM generates random values of weights, however after the synchronization process both users have TPMs with the same values of weights. Therefore, Alice and Bob can use this phenomenon to correct errors occurring in the quantum channel.

In order to synchronize neural networks, Alice or Bob generates random inputs and both users compute outputs from each TPM. If the outputs have the same value, they start the learning process, but if the outputs are different, a new string of bits must be generated. Alice and Bob can choose any learning algorithm; however, the generalized form of Hebbian method is the most popular in practical implementations [32]. This algorithm strengthens the connections which have the same value as the TPM output. The new weights are calculated by means of the following formula:

$$w_{kn}^{\star} = \nu_L(w_{kn} + x_{kn} * \sigma_k * \Theta(\sigma_k, \tau)) \tag{7}$$

where:

$$\Theta(\sigma_k, \tau)) = \begin{cases} 0 & \text{if } \sigma_k \neq \tau \\ 1 & \text{if } \sigma_k = \tau \end{cases} \tag{8}$$

and function ν_L limits values of connections to the range $[-L, L]$:

$$\nu_L(z) = \begin{cases} -L & \text{if } z \leq -L \\ z & \text{if } -L < z < L \\ L & \text{if } z \geq L \end{cases} \tag{9}$$

After the appropriate number of iterations, the synchronization process ends, and the weights of both TPM machines are the same. However, synchronization

of TPMs requires public channel for communication between Alice and Bob where Eve can eavesdrop and try to synchronize her own TPM machine with Alice and Bob. Fortunately, if the output of Eve's TPM machine is different than the outputs of Alice and Bob's machines, the learning process cannot be performed. Therefore, the synchronization of Eve's TPM is much slower than the synchronization of the TPMs belonging to Alice and Bob. An example of the synchronization process is presented in Fig. 5 (TPM machines with parameters: N = 8, K = 6, L = 2 and Hebbian learning algorithm). Alice and Bob synchronized neural networks before 200 iterations, but the attacker was not able to do it for 1000 iterations.

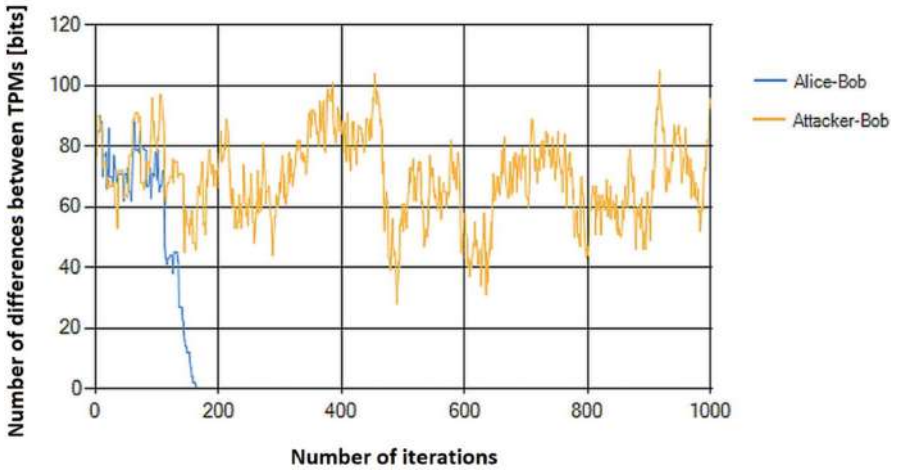


Fig. 5. Example of TPMs synchronization: Alice's TPM and Bob's TPM, Bob's TPM and Attacker's TPM (TPM machines with parameters: N = 8, K = 6, L = 2 and Hebbian learning algorithm)

6.2 Error Correction Based on TPMs

We can use the presented synchronization of the TPM machines to correct errors in the quantum cryptography. In the beginning, Alice and Bob create their own TPM machines based on their own strings of bits. The users change the string of bits into weights in their own TPM machines (bits into numbers from the range $[-L, L]$). Values $\{-L, -L+1, ...L-1, L\}$ become weights of connections between the input neurons and the neurons in the hidden layer. In this way, Alice and Bob construct very similar neural networks – the TPM machines have the same structure, and most of the weights are the same. The differences are located only in the places where errors occurred: for example, if QBER ≈3%, it means that ≈97% of bits are correct. After this, synchronization of the

TPM machines begins and continues until all weights in both machines become the same. When each random input is chosen (input strings have KN length), the users compute outputs and compare the obtained values. When the TPM machines are synchronized, the weights are the same in both neural networks. Therefore Alice and Bob can convert the weights back into bits because both strings are now the same. All errors have been corrected.

Importantly, Alice's binary string is very similar to Bob's string of bits. The typical value for QBER does not exceed a few percent; therefore we must correct only a small part of the whole key. This means that the TPM machines are close to synchronization and the learning process will finish much faster than in the case of synchronization of random strings of bits. Of course, this increases the security level significantly.

It is worth mentioning that this idea – using the mutual synchronization of neural networks to correct errors – is a special case when this process makes sense. In general, TPM machines cannot be used for error correction of digital information because we are not able to predict the final weights after the learning process.

7 Conclusion

In this chapter, we analyzed techniques of implementing the key reconciliation using Cascade, Winnow, Low-density parity-check code and the application of neural networks with a focus on communication and computing performances.

Our previous results [9] showed that key reconciliation process takes the dominant part of QKD post-processing. With increasing interest in satellite and global QKD connections, minimizing the duration of key establishment process is becoming an increasingly attractive area. It is necessary to take into account the possibilities of asymmetric processing, which simplifies the requirements for computing power budgets as well as requirements for minimizing the exchange of packets to reduce overhead and the ability to work in networks with weaker network performance (bandwidth and network delay).

Since the development of metropolitan QKD testbed networks [33–39], LDPC is increasingly being considered as an adequate basis for the key reconciliation process in QKD, and there are noticeable variations in how this protocol is implemented. However, techniques of reversibility or on artificial neural networks can significantly improve the process to reduce communication and computing resources and represent areas of great interest for further research.

References

1. Bennett, C.H., Brassard, G.: Quantum cryptography: public key distribution and coin tossing. In: Proceedings of IEEE International Conference on Computers, Systems and Signal Processing, New York, vol. 175, p. 8 (1984)
2. Scarani, V., Bechmann-Pasquinucci, H., Cerf, N.J., Dušek, M., Lütkenhaus, N., Peev, M.: The security of practical quantum key distribution. Rev. Mod. Phys. **81**(3), 1301–1350 (2009)

3. Assche, G.V.: Quantum Cryptography and Secret-Key Distribution. Cambridge University Press, Cambridge (2006)
4. Kollmitzer, C., Pivk, M.: Applied Quantum Cryptography, vol. 1. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-04831-9
5. Dodson, D., et al.: Updating quantum cryptography report ver. 1. arXiv preprint arXiv:0905.4325, May 2009
6. Dusek, M., Lutkenhaus, N., Hendrych, M.: Quantum cryptography. In: Progress in Optics, vol. 49, pp. 381–454. Elsevier, January 2006
7. Niemiec, M., Pach, A.R.: The measure of security in quantum cryptography. In: 2012 IEEE Global Communications Conference (GLOBECOM), pp. 967–972, December 2012
8. Mehic, M., Niemiec, M., Voznak, M.: Calculation of the key length for quantum key distribution. Elektron. Elektrotech. **21**(6), 81–85 (2015)
9. Mehic, M., Maurhart, O., Rass, S., Komosny, D., Rezac, F., Voznak, M.: Analysis of the public channel of quantum key distribution link. IEEE J. Quantum Electron. **53**(5), 1–8 (2017)
10. Mehic, M., et al.: A novel approach to quality-of-service provisioning in trusted relay quantum key distribution networks. IEEE/ACM Trans. Netw. **28**(1), 168–181 (2020)
11. Brassard, Gilles, Salvail, Louis: Secret-key reconciliation by public discussion. In: Helleseth, Tor (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 410–423. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-48285-7_35
12. Sugimoto, T., Yamazaki, K.: A study on secret key reconciliation protocol. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E83-A**(10), 1987–1991 (2000)
13. Lustic, K.: Performance analysis and optimization of the winnow secret key reconciliation protocol. Ph.D. thesis, Air Force Institute of Technology (2010)
14. Ruth, Y.: A probabilistic analysis of binary and cascade. math.uchicago.edu (2013)
15. Calver, T.: An empirical analysis of the cascade secret key reconciliation protocol for quantum key distribution. Master thesis (2011)
16. Pedersen, T.B., Toyran, M., Pearson, D., Pedersen, T.B., Toyran, M.: High performance information reconciliation for QKD with CASCADE. Quantum Inf. Comput. **734**(5–6), 419–434 (2013)
17. Keath, C.: Improvement of reconciliation for quantum key distribution. Ph.D. thesis, Rochester Institute of Technology, February 2010
18. Nguyen, K.C.: Extension des protocoles de réconciliation en cryptographie quantique. Université Libre de Bruxelles, Travail de fon d'études (2002)
19. Yan, H., et al.: Information reconciliation protocol in quantum key distribution system. In: Proceedings - 4th International Conference on Natural Computation, ICNC 2008, vol. 3, pp. 637–641 (2008)
20. Martinez-Mateo, J., Pacher, C., Peev, M., Ciurana, A., Martin, V.: Demystifying the information reconciliation protocol cascade. arXiv preprint arXiv:1407.3257, pp. 1–30, July 2014
21. Nakassis, A., Bienfang, J.C., Williams, C.J.: Expeditious reconciliation for practical quantum key distribution. In: Donkor, E., Pirich, A.R., Brandt, H.E. (eds.) Quantum Information and Computation II, vol. 5436, p. 28, August 2004
22. Buttler, W.T., Lamoreaux, S.K., Torgerson, J.R., Nickel, G.H., Donahue, C.H., Peterson, C.G.: Fast, efficient error reconciliation for quantum cryptography. Phys. Rev. A **67**(5), 052303 (2003)
23. Elkouss, D., Leverrier, A., Alleaume, R., Boutros, J.J.: Efficient reconciliation protocol for discrete-variable quantum key distribution, June 2009

24. Gallager, R.G.: Low-density parity-check codes. IRE Trans. Inf. Theory **8**, 21–28 (1962)
25. Elkouss, D., Martinez-Mateo, J., Vicente, M.: Information reconciliation for QKD. Quantum Inf. Comput. **11**(March), 226–238 (2011)
26. Elkouss, D., Martinez-Mateo, J., Martin, V.: Analysis of a rate-adaptive reconciliation protocol and the effect of leakage on the secret key rate. Phys. Rev. A - At. Mol. Opt. Phys. **87**(4), 1–7 (2013)
27. Elliott, C., Colvin, A., Pearson, D., Pikalo, O., Schlafer, J., Yeh, H.: Current status of the DARPA quantum network (Invited Paper). In: Donkor, E.J., Pirich, A.R., Brandt, H.E. (eds.) Quantum Information and Computation III. Proceedings of SPIE, vol. 5815, pp. 138–149, May 2005
28. Niemiec, M.: Error correction in quantum cryptography based on artificial neural networks. Quantum Inf. Process. **18**(6), 174 (2019)
29. Kanter, I., Kinzel, W.: The theory of neural networks and cryptography (2007)
30. Hadke, P.P., Kale, S.G.: Use of neural networks in cryptography: a review. In: IEEE WCTFTR 2016 - Proceedings of 2016 World Conference on Futuristic Trends in Research and Innovation for Social Welfare (2016)
31. Chakraborty, S., Dalal, J., Sarkar, B., Mukherjee, D.: Neural synchronization based secret key exchange over public channels: a survey. In: 2014 International Conference on Signal Propagation and Computer Technology, ICSPCT 2014 (2014)
32. Kriesel, D.: A brief introduction on neural networks. Technical report, December 2007. www.dkriesel.com
33. Elliott, C., Yeh, H.: DARPA quantum network testbed. Technical report July, BBN Technologies Cambridge, New York, USA, New York (2007)
34. Alleaume, R., et al.: SECOQC white paper on quantum key distribution and cryptography. arXiv preprint quant-ph/0701168, p. 28 (2007)
35. Korzh, B., et al.: Provably secure and practical quantum key distribution over 307 km of optical fibre. Nat. Photon. **9**(3), 163–168 (2015)
36. Sasaki, M.: Tokyo QKD network and the evolution to secure photonic network. In: CLEO:2011 - Laser Applications to Photonic Applications, vol. 1, JTuC1. OSA, Washington, D.C. (2011)
37. Dixon, A.R., Yuan, Z.L., Dynes, J.F., Sharpe, A.W., Shields, A.J.: Continuous operation of high bit rate quantum key distribution. Appl. Phys. Lett. **96**(2010), 2008–2011 (2010)
38. Salvail, L., Peev, M., Diamanti, E., Alléaume, R., Lütkenhaus, N., Länger, T.: Security of trusted repeater quantum key distribution networks. J. Comput. Secur. **18**(1), 61–87 (2010)
39. Shimizu, K., et al.: Performance of long-distance quantum key distribution over 90-km optical links installed in a field environment of Tokyo metropolitan area. J. Lightwave Technol. **32**(1), 141–151 (2014)

Author Index