# Software and Reversible Systems: A Survey of Recent Activities

Claudio Antares Mezzina[1(✉)], Rudolf Schlatte[2], Robert Glück[3], Tue Haulund[4], James Hoey[5], Martin Holm Cservenka[6], Ivan Lanese[7], Torben Æ. Mogensen[3], Harun Siljak[8], Ulrik P. Schultz[9], and Irek Ulidowski[5]

[1] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Urbino, Italy
[2] Department of Informatics, University of Oslo, Oslo, Norway
[3] DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
torbenm@di.ku.dk
[4] A.P. Moller Maersk, Copenhagen, Denmark
[5] School of Informatics, University of Leicester, Leicester, UK
[6] Practio ApS, Copenhagen, Denmark
[7] Focus Team, University of Bologna/Inria, Bologna, Italy
[8] CONNECT Centre, Trinity College Dublin, Dublin, Ireland
[9] University of Southern Denmark, Odense, Denmark

**Abstract.** Software plays a central role in all aspects of reversible computing. We survey the breadth of topics and recent activities on reversible software and systems including behavioural types, recovery, debugging, concurrency, and object-oriented programming. These have the potential to provide linguistic abstractions and tools that will lead to safer and more reliable reversible computing applications.

## 1 Introduction

The notion of reversible computation has a long history [37] which started by studies on the thermodynamic cost of irreversible actions. It was noted that since computation is usually irreversible, information loss causes dissipation of heat. Therefore it could be possible to execute reversible computations in a heat dissipation free way. This was the motivation that gave rise to several reversible computation models such as *reversible Turing machines* [6] and *conservative logic* [22]. Since then there has been a huge effort to introduce reversibility at the level of programming languages and software systems [7,44], where it can bring additional benefits towards reliability, robustness and scalability of conventional software systems. Part of this effort has been carried out by the Working Group (WG) 2: Software and Systems of the COST Action IC1405 Reversible Computation – Extending Horizons of Computing.

This work has been partially supported by COST Action IC1405 on Reversible Computation - Extending Horizons of Computing.


I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 41–59, 2020.
https://doi.org/10.1007/978-3-030-47361-7_2

Software plays a central role in all aspects of reversible computing. We survey the breadth of topics and recent activities on reversible software and systems including behavioural types, recovery, debugging, concurrency, and object-oriented programming. These have the potential to provide linguistic abstractions and tools that will lead to safer and more reliable reversible computing applications.

The rest of the chapter is structured as follows: Sect. 2 reports on reversibility and behavioural types; Sect. 3 reports on the interplay between reversibility and recovery for distributed systems; Sect. 4 reports on reversibility and object orientation; Sect. 5 reports on reversing imperative programs with shared memory concurrency and its possible application on reversible debugging; Sect. 6 reports on reversibility and message passing systems, with a special focus on reversible (core) Erlang and its reversible debugger. Section 7 reports on reversibility and control theory. Section 8 concludes the chapter.

## 2 Behavioural Types

The interest in behavioural types [35] stems from the fact that it is easier to work with a system whose behaviour (in terms of communications) is strongly disciplined by a type theory. Among behavioural types we distinguish: *binary session types* and *contracts*, *multiparty session types* and *choreographies*. Choreographies will be discussed in Sect. 3.

Reversibility and monitored semantics for *binary session types* have been recently studied by Mezzina and Pérez [46,47,49]. In their work, they propose a *monitor as memory* mechanism in which information about the monitor of a process can be used to enable its reversibility. Moreover, by adding *modalities* information at the level of session types, reversibility can be controlled.

In the context of multiparty session types, *global types* describe the message-passing behaviour of a set of participants in a system from a global point of view. A global type can be projected onto each participant so as to obtain local types, which describe individual contributions to the global protocol. The work [48] extends global and local types to keep track of the stage of the protocol that has been already executed; this enables reversible steps in an elegant way. The authors develop a rigorous process framework for multiparty communication, which improves over prior works by featuring asynchrony, decoupled rollbacks and process passing. In this framework, concurrent processes are untyped but their forward and backward steps are governed by monitors. The main technical result is that the developed multiparty reversible semantics is causally-consistent. Finally, [15] proposes a Haskell implementation of the asynchronous reversible operational semantics for multiparty session types proposed in [48]. The implementation exploits algebraic data types to faithfully represent three core ingredients: a process calculus, multiparty session types, and forward and backward reduction semantics. This implementation bears witness to the convenience of pure functional programming for implementing reversible languages.

In a series of works [11,16] *multiparty session types* (aka global types) have been enriched with checkpoint labels on choices that mark points of the protocol where the computations may roll back. In [16], a simple model is developed in which rollback could be done any time after a participant has crossed the checkpointed choice. In [11] a more refined model is presented, in which the programmer can define points where the computation may revert to a checkpointed label, and rollback has to be triggered by the participant that made the decision.

*Behavioural contracts* are abstract descriptions of expected communication patterns followed by either clients or servers during their interaction. Behavioural contracts come naturally equipped with a notion of *compliance*: when a client and a server follow compliant contracts, their interaction is guaranteed to progress or successfully complete. In [5] two extensions of behavioural contracts are studied: *retractable contracts* dealing with *backtracking* and *speculative contracts* dealing with *speculative execution*. These two extensions give rise to *the same notion of compliance*. As a consequence, they also give rise to the same *subcontract relation*, which determines when one server can be replaced by another while preserving compliance. Moreover, compliance and subcontract relation are both decidable in quadratic time. The above paper also studies the relationship between retractable contracts and calculi for reversible computing.

## 3 Recovery

Distributed programs are hard to get right because they are required to be open, scalable, long-running, and tolerant to faults. This problem is exacerbated by the recent approaches to distributed software based on (micro-)services where different services are developed independently by disparate teams. In fact, services are meant to be composed together and run in open context where unpredictable behaviours can emerge. This makes it necessary to adopt suitable strategies for monitoring the execution and incorporate recovery and adaptation mechanisms to make distributed programs more flexible and robust. The typical approach that is currently adopted is to embed such mechanisms in the program logic, which makes it hard to extract, compare and debug.

An approach that employs formal abstractions for specifying failure recovery and adaptation strategies has been proposed in [10]. Although implementation-agnostic, these abstractions would be amenable to algorithmic synthesis of code, monitoring and tests. Message-passing programs (à la Erlang, Go, or MPI) are considered, since they are gaining momentum both in academia and industry. In [20] an instance of the framework proposed in [10] is given. More precisely, this approach imbues the communication behaviour of multi-party protocols with minimal decorations specifying the conditions triggering monitor adaptations. It is then shown that, from these extended global descriptions, one can (i) synthesise actors implementing the normal local behaviour of the system prescribed by the global graph, but also (ii) synthesise monitors that are able to coordinate a distributed rollback when certain conditions (denoting abnormal behaviour) are met. The synthesis algorithm produces Erlang code. For each role in the global

description, two Erlang actors are generated: one actor implements the normal (forward) behaviour of the system and a second one (the monitor) is in charge of implementing the reversible behaviour of the role. When certain conditions are detected at runtime, the monitors will coordinate with each other in order to bring back the system if possible. One interesting property of this approach is that the two semantics are highly decoupled, meaning that the system is always able to normally execute (i.e., going forward) even in case of a monitor crash.

A static analysis, based on multiparty session types, to efficiently compute a safe global state from which to recover a system of interacting processes has been integrated with the Erlang recovery mechanism in [50]. From a global description of the program communication flow, given in multiparty protocol specification, causal dependencies between processes are extracted. This information is then used at runtime by a recovery mechanism, integrated in Erlang, to determine which process has to be terminated and which one has to be restarted upon a node failure. Experimental results indicate that the proposed framework outperforms a built-in static recovery strategy in Erlang when a part of the protocol can be safely recovered.

In [26] a rollback operator, based on the notion of causal-consistent reversibility, is defined for a language with shared memory. A rollback is defined as the minimal causal-consistent sequence of backward steps able to undo a given action. The paper [69] explores the relationship between the Manetho [17] distributed checkpoint/rollback scheme (based on causal logging) and a reversible concurrent model of computation based on the $\pi$-calculus with imperative rollback called `roll`-$\pi$ [38]. A rather tight relationship between rollback based on causal logging as performed in Manetho and the rollback algorithm underlying `roll`-$\pi$ is shown. The main result is that `roll`-$\pi$ can faithfully simulate Manetho under weak barbed simulation, but that the converse only holds if possible rollbacks are restricted.

## 4 Reversibility and Object-Oriented Languages

Object-oriented (OO) programming uses classes as a means to encapsulate behaviour and state. Classes permit programmers to define new abstractions, such as abstract data types. The key elements of reversible OO languages were initially introduced with a prototype of the Joule language [60] and subsequently formally described for the ROOPL language [29]. Joule and ROOPL demonstrate that well-known object-oriented concepts such as encapsulation, inheritance, and virtual methods can be captured reversibly by extending a base Janus-like imperative language [71] with support for such features.

This approach allows standard OO programming patterns, such as the factory and iterator design patterns [23], to be used reversibly [59], and well-known structures such as an OO-style collection hierarchy (i.e., OO abstract data types but with reversible operations) can similarly be implemented in such languages. Reversible data types [13], that is data structures with all of its associated operations implemented reversibly, are enabled by dynamic allocation of constructor

terms in the heap of a reversible machine [1]. Data structures are safe in OO languages because they require no explicit pointer arithmetic in user programs, which is notoriously error prone.

Memory handling is a key concern for reversible object-oriented languages. The original Joule prototype relied on static stack allocation of objects, which does not permit full OO programming: common patterns such as factories are for example not possible [60]. Joule was subsequently extended into Joule$^R$ which uses region-based [24,66] memory management [59]. Regions are sufficient to support the implementation of standard OO programming patterns and a collection hierarchy. The initial presentation of the ROOPL language relied exclusively on stack allocation [29], and was subsequently extended with a reversible heap-based memory manager [13] based on Knuth's Buddy Memory algorithm [36]. With this extension, data structures such as min-heaps and circular buffers can be implemented [13]. The language is reversibly universal (r-Turing complete), which means it has the computational power of reversible Turing machines (cf. [71]). See Figs. 1, 2, and 3 for example programs in Joule and ROOPL, which will be described in the next section.

### 4.1 Object Orientation and Data Structures

As exemplified by the representation of abstract-syntax trees in the reversible Janus self-interpreter [73], even complex data structures can be expressed in reversible languages with simple type systems including only integers and arrays. However, more effort is required to represent and manipulate the data structures and as the resulting code base grows, the problem exacerbates.

Reversible object-oriented languages allow for easier code reuse and extensibility by encapsulating data and methods in classes, thereby also abstracting from the underlying memory model of the reversible machine. See Figs. 1 and 2 for two classic object-oriented examples in Joule and ROOPL, respectively.

The example in Joule in Fig. 1 models a single point in a two-dimensional space by a class `Point` with two integer coordinates (`x`, `y`) and two methods that translate a point by adding an integer displacement to the respective coordinate (`add_to_x`, `add_to_y`). Here, `this.x` refers to the `x`-coordinate of the point to which the displacement parameter `x` is added when `add_to_x` is applied to a point object.

The example in ROOPL in Fig. 2 illustrates a simple *class hierarchy* of geometric shapes in a two-dimensional space. The two shapes `Rectangle` and `Circle` *inherit* the reference point (`x`, `y`) from their superclass `Shape` and *extend* it with the length and width (`l`, `w`) in the case of `Rectangle` and with the radius `r` in the case of `Circle`. The two subclasses add a class-specific method `getArea` that defines how to calculate the area of the respective shape. All methods defined in these three classes are implemented by reversible statements that are similar to those in Janus and reversible flowcharts [71,73]. Methods can also be implemented using reversible control-flow operators (conditionals, iteration) and recursive method calls and uncalls, as illustrated in the next example. It is

```
1  class Point {
2      int x; int y; // private fields, zero-initialised
3
4      Point(int x, int y) { // constructor, runs after allocation
5          this.x += x; this.y += y; // this.x is a field, x a parameter
6      }
7
8      procedure add_to_x(int x) { this.x += x; }
9      procedure add_to_y(int y) { this.y += y; }
10 }
```

**Fig. 1.** Example Joule class modelling a single point in two-dimensional space, originally from [60]

```
1  class Shape                              // superclass Shape
2      int x, y                             // reference point
3
4      method getArea(int out)              // abstract method
5          skip
6
7      method translate(int dx, int dy) // common method
8          x += dx
9          y += dy
10
11 class Rectangle inherits Shape           // subclass Rectangle
12     int l, w                             // length, width
13
14     method getArea(int out)              // concrete method
15         out ^= l * w
16
17 class Circle inherits Shape              // subclass Circle
18     int r                                // radius
19
20     method getArea(int out)              // concrete method
21         out ^= PI * r * r
```

**Fig. 2.** Example ROOPL class hierarchy modeling basic geometric shapes in two-dimensional space, originally from [13]

important to note that a *reversible method* cannot overwrite any of the encapsulated data, only perform a *reversible update* [2]. This makes reversible OO languages different from their mainstream counterparts, such as Java or C++, which can perform destructive updates.

The reversible min-heap in Fig. 3 serves as an example of the expressiveness afforded by the richer type systems and memory models of these languages. The `insert` method reversibly inserts a node in the heap, where the only output is the depth of the inserted node, maintaining the min-heap property in the process. This procedure can be used to reversibly extract the minimal value of a data set. The class `Node` recursively defines a binary tree structure by including two nodes, `left` and `right`. The integer `v` is the value of a node.

The `insert` method makes use of a *reversible conditional* `if...fi` (lines 5 to 16), which means it contains not only an entry predicate (`v < w`) but also an exit predicate (`counter > 0`). As usual in reversible languages, both predicates are checked at runtime: both must be true when control passes along the then-branch and both must be false when control passes along the else-branch; otherwise, the

```
1  class Node
2      Node left, right /* roots of subtrees */
3      int v            /* value of node     */
4      method insert(int w, int counter) /* counter initially 0 */
5          if v < w then
6              if left = nil then
7                  new Node left
8                  left.v <=> w
9              else call left::insert(w, counter)
10             fi left.right = nil
11             counter += 1 /* counter > 0 */
12         else
13             v <=> w
14             call insert(w, counter)
15             counter -= 1 /* counter = 0 */
16         fi counter > 0
17         left <=> right
18         /* at return, w = 0 and counter = depth of insertion */
```

**Fig. 3.** Recursive min-heap value insertion implemented in ROOPL using reversible updates and reversible conditionals, originally from [13]

program is undefined (cf. [71,73]). Method calls and uncalls refer to an object. For example, `call left::insert(w, counter)` recursively applies the insert method to the left node `left` with the integer parameters `w` and `counter`. This allows to work with recursively-defined data structures, which in our case are binary trees.

Objects, which are instances of the classes defined in a program, can be allocated and deallocated at runtime in any order using explicit statements. For example, a new object of class node is created by statement `new Node left` where the object's reference is assigned to `left` (line 7). When a new object is created all its fields are initialised with default values, here integer `v` is initialised with zero and references `left` and `right` with the null pointer `nil`.

Reversible programming demands certain sacrifices compared to mainstream programming because data cannot be overwritten and join points in the control flow require explicit tests (e.g., the exit predicate in `if...fi`), which can also be seen in the case of the insert method. As a consequence, conventional algorithms and data structures need to be rethought in a reversible context regardless of the data structures offered by a reversible language [13,27,28,72]. However, the abstraction and expressiveness of OO reversible data structures ease the task.

With the addition of Joule and ROOPL, reversible programs can now be expressed in a modern programming paradigm like OO programming, with dynamic memory management of variably sized records and programmer-defined recursive data structures that can grow to an arbitrary size at runtime. These new features significantly broaden the applicability of reversible languages and support increased complexity in reversible programs.

## 5 Reversing Imperative Concurrent Programs

Adding reversibility to irreversible imperative languages has been studied for many years, for example in [9,52,57,58,70]. A proof of correctness is often

missing from work in this area. Hoey and Ulidowski introduce a small imperative while language and describe a state-saving approach to reversing executions [33]. This was then extended to support an imperative concurrent language, using identifiers to capture the specific interleaving order and to ensure statements are reversed in the correct order [34]. The proof of correctness provided shows that the reversal is both correct and garbage free. A simulation tool implementing this approach is mentioned in [32] and described in more detail in [30]. Performance evaluation carried out using this simulator indicates that overheads associated with saving and using of reversal information is reasonable. Finally, a link between this simulator and debugging is explored in [32].

### 5.1 Language and Program State

The imperative language used in this approach contains assignments, conditional statements (branching) and loops (iteration), much like a while language. Details on reversing this imperative while language are available in [33]. This is later extended with block statements containing local variable or procedure declarations, as well as (potentially recursive) procedure calls. With the ability for multiple variables to share a name as a result of local variables, the syntax of this language contains *construct identifiers* (unique names given to complex constructs including block statements) and *paths* (sequence of block names in which a statement resides capturing the position needed for evaluation). Block statements allow the declaration of local variables or procedures, and as such are extended to "clean" up at the end of its execution by "un-declaring" these via *removal statements*. The final addition is that of *interleaving parallel composition*, where the execution of two (or more if nested) programs can be interleaved. The syntax of this language follows.

```
P ::= ε | S | P; P | P par P
S ::= skip I | X = E (pa,A) | if In B then P else Q end (pa,A) |
      while Wn B do P end (pa,A) | begin Bn BB end |
      call Cn n (pa,A) | runc Cn P end
BB ::= DV; DP; P; RP; RV
DV ::= ε | var X = v (pa,A); DV      DP ::= ε | proc Pn n is P end (pa,A); DP
RV ::= ε | remove X = v (pa,A); RV  RP ::= ε | remove Pn n is P end (pa,A); RP
```

The program state is represented as a series of environments, including the *variable environment* $\gamma$ (linking variables to memory locations), the *data store* $\sigma$ (linking memory locations to values), the *procedure environment* $\mu$ (storing multiple copies of procedure bodies being executed in parallel) and the *while environment* $\beta$ (storing multiple copies of loops being executed in parallel) [34].

### 5.2 Annotation, Inversion and Operational Semantics

The considered approach is state-saving, where any information required for inversion that is lost during traditional execution is saved [52]. Two versions of an original program are produced. The first, named the *annotated version* and generated via *annotation*, performs the expected forwards execution and saves any required information, named *reversal information*. A design choice made to aid the correctness proof is to store all reversal information in an *auxiliary store* $\delta$ separate to the program state. This store is a collection of stacks (ideal for reversal due to their FIFO nature), one for each variable name (all versions share a stack to handle races), two stacks for loops (one for capturing the loop count and one for identifiers), one for conditional statements and one for procedure calls.

The information required depends on the type of statement. Each assignment is destructive as the old value of the variable is lost. This old value is crucial for reversal, thus it is saved into the stack for that variable name on $\delta$ prior to each assignment. Conditions are not guaranteed to be invariant, meaning this approach cannot rely on re-evaluation during inversion to behave correctly. For each conditional statement, the result of evaluation is saved onto the stack for conditionals on $\delta$. Loops are handled similarly, with a sequence of booleans saved to capture the number of iterations (onto the first stack for loops). A second design choice made is to save a sequence over implementing a loop counter in order to aid the correctness proof, avoiding modifying the loop code and therefore the behaviour with respect to the program state. Lastly, the final value of a local variable is saved prior to its removal, into the stack for that variable name.

Supporting interleaving parallel composition also requires further information to be saved. Interleaving allows different execution orders to be followed, which must then be correctly inverted. The specific execution order is captured using *identifiers* similarly to Phillips and Ulidowski [55,56]. The next identifier is assigned to a statement as it executes, stored into a stack of integers associated with each required statement during annotation. Consider the small example shown in Fig. 4 and the executed forwards version shown in Fig. 4a. This is a simple interleaving of three statements, captured via the identifiers 1–3, where the first statement of the right hand side is executed first, before interleaving to the left and finally completing the right. Assuming `X` and `Y` are initially `1`, this interleaving produces the final state `X = 4` and `Y = 3`. These identifiers also create a link between a statement and its reversal information, as all entries on $\delta$ contain the corresponding identifier. For example, the stack `X` on $\delta$ will contain the pair `(2,1)` (statement with identifier `2` overwrote the value `1`). For loops or procedure calls (potentially multiple copies of the same code in execution across a parallel), identifiers are assigned to the specific copy within $\mu$ or $\beta$. Since local copies are removed at the end of their execution, the final example of reversal information is the identifiers assigned to such a copy (saved onto the second stack for loops or the stack for calls).

```
X = Y+2 [2]; par Y = X+2 [1];
                X = 4 [3];
```

(a) Executed annotated program

```
X = Y+2 [2]; par X = 4 [3];
                Y = X+2 [1];
```

(b) Inverted program

**Fig. 4.** Identifier use example

The execution of an annotated program is defined in terms of small step operational semantics, where each rule performs the expected forwards execution alongside the saving of reversal information and assigning of an identifier [34].

The second version of an original program produced, called the *inverted version*, is generated via *inversion* and has an inverted statement order with all declaration statements changed to removals and vice versa. This forwards-executing program simulates reversal using the saved information and identifiers.

Throughout the inverse execution, the decision of which statement to execute next (that is, invert) is made using the identifiers in descending order to force backtracking order. Returning to the example in Fig. 4, the identifiers are used in the order 3–1, meaning any incorrect inverse execution path cannot be followed. Each statement also uses the identifiers to access the correct reversal information. Assignments will no longer evaluate the expression and instead retrieve the old value from $\delta$. From the example in Fig. 4b, execution of the statement with identifier 2 uses the pair `(2,1)` to restore the variable to `1`. Similarly conditionals and loops retrieve the result of condition evaluation from $\delta$. Declaring a local variable during an inverse execution initialises it to the final value it held during forward execution (retrieved from the stack). Lastly, whenever a copy of a loop or procedure body is made during the inverse execution, it is populated with the required identifiers from $\delta$.

As before, inverse execution is defined by small step semantics, with each rule using identifiers and reversal information to undo the effects of a statement (or step). Complete inverse execution undoes the effects of all statements, producing a state equivalent to that of prior to the forward execution. We refer to the previous property, coupled with the property that all reversal information is consumed (the approach is *garbage free*), as *correct inversion*.

### 5.3 Correctness of Annotation and Inversion

This approach is proved to perform correct reversal information saving as well as correct and garbage-free inversion. The two results are described in [34] and extended to hold for all programs including parallel composition in [30]. The first, named the *annotation result*, states that an original program and its annotated version executed on the same initial program state will produce equivalent final program states, with the obvious exception of the annotated execution populating the auxiliary store with the required reversal information.

The second result, named the *inversion result*, states that provided an annotated execution has been performed producing the final program state and auxiliary store, then the corresponding inverse execution ran on these final stores will

produce a program state and auxiliary store equivalent to that of prior to the forwards execution. This means the inverse execution reverses all effects of the original program, as well as using all of the reversal information saved (the approach is garbage free). These two results together show that no state is reached that was not originally reached in either the forward or reverse execution.

### 5.4 Simulator and Performance Evaluation

A simulator implementing this approach has been developed, originally for the purposes of testing [30]. The simulator reads a program written in a simplified language (omitting paths, construct identifiers and removal statements as these can be automatically inserted), parses it and sets up the initial program state. Key features include *complete* or *step-by-step* execution, *viewable program state and reversal information* at any point, *random or manual interleaving* and *record mode* (storing further details including interleaving decisions/rule applications).

This simulator has been used for performance evaluation. Design choices (mentioned above) have been made to aid the proof and may not be the most efficient solution, and no optimisation techniques have yet been applied. This analysis concerns the overhead associated with annotation (time required to save reversal information), and the overhead associated with inversion (inverse execution time compared to annotated forward execution time). From figures in [32], the annotated execution experiences a reasonable overhead of between 4.2%–13.4%, while the inverted execution experiences an again reasonable overhead of between −14.7%–1.9%. As expected, the inverse execution is sometimes faster as there is no evaluation (values retrieved from $\delta$).

### 5.5 Application to Debugging

Many works including [12,18,25,40,41,68] have described how reversibility can be beneficial for debugging. The link between this approach to reversibility and debugging is explored in [32], showing that this simulator (not originally developed as a debugger) helps with finding errors. Benefits include bugs being reproducible should a user wish to re-execute a program forwards (for example, a randomly interleaved program experiences a bug that can only be reproduced by luck, with inversion obviously still possible), the ability to pause executions and to view program state at any point. In [32] and [31], this simulator is used to debug an example atomicity violation.

## 6 Reversible Debugger for Message Passing Systems

A relevant research thread in WG2 has tackled the problem of debugging concurrent message-passing applications using the so called *causal-consistent* approach. Causal-consistent reversibility [14] stems from the observation that in concurrent systems, events (e.g., sending and receive of messages) are not always totally ordered since there may be no unique notion of time. Even if events are totally

ordered in principle, such an order is not relevant since it depends on the speed of execution of the various processes, and it is difficult to observe and even more to control. Instead, events naturally form a partial order dictated by causality: causes precede their consequences, while there is no order between concurrent events. The corresponding notion of reversibility, causal-consistent reversibility, allows one to undo any event, provided that its consequences, if any, are undone beforehand. A main property of this notion of reversibility is that states reachable via backward computation are also reachable via forward computation from the initial state, hence reversibility does not introduce new states but only provides different ways of exploring states of forward computations.

This observation led to the development of *causal-consistent reversible debugging* [25], which allows one to explore a concurrent computation backward and forward, looking for the causes of a given misbehaviour, e.g., a wrong value printed on the screen. Indeed, a misbehaviour is due to a bug, that is a wrong line of code, and the execution of the wrong line of code is a cause of the misbehaviour. More precisely, causal-consistent reversible debugging provides primitives to undo past events, including all and only their consequences. For instance, if variable x has a wrong value, one can go back to where variable x has been assigned. If the wrong value is in a message payload, one can go back where the message has been sent. By iterating this technique, one can look for causes of the misbehaviour until the bug is found.

Inside WG2 the research focused on how to apply this approach to a real programming language, and Erlang was the language of choice. Erlang features native primitives for message-passing concurrency, and has been used in relevant applications such as some versions of Facebook chat [45]. For simplicity, the research thread does not deal directly with Erlang, but with Core Erlang [8], which is an intermediate step in Erlang compilation, essentially removing some syntactic sugar from Erlang.

The research thread started with an investigation on the reversible semantics of Core Erlang, aiming at defining a rollback operator to undo a past action in a causal-consistent way [51]. The study was further developed in [42], where relevant properties of the approach were proved, e.g., that the rollback operator indeed satisfies the constraints of causal-consistent reversibility. The focus on debugging started in [41], where CauDEr [40], a Causal-consistent Debugger for (core) Erlang, was described. CauDEr provided the primitives above for causal-consistent reversible debugging, paired with primitives for forward execution and with a graphical interface to show the runtime structure of the program under analysis and the relevant concurrent events in the computation.

A main limitation of CauDEr was that if the user went too far back, there was no automatic way to go forward again with the guarantee to replay the misbehaviour under analysis. This is a relevant problem, since in concurrent systems misbehaviours depend on the scheduling, and of course it is not possible to debug a misbehaviour that does not appear when executing the wrong application inside the debugger. To solve this problem, the research studied techniques for tracing a computation and replay it inside the debugger. This lead to the

definition of a new form of replay, called *causal-consistent replay* [43], which allows one to redo a future event of a traced computation, including all and only its causes. One can notice that causal-consistent reversibility and causal-consistent replay are dual, and together they allow one to explore a wrong computation back and forward, always concentrating on events of interest. Also, this approach ensures that if a misbehaviour occurred in the traced computation then the same misbehaviour occurs also in each possible replay (provided that execution goes forward enough). A tracer for Erlang compatible with CauDEr was produced and is available at [39]. An example of application of this framework to a simple Erlang program can be found in [21].

## 7 Control Theory

The challenge of reversible control is its interaction with the irreversible object of control. Even when the object is reversible, (e.g. motion of a fluid) often the ability to reverse it is not controllable [61]. Disturbance in the system can be fully reversible, but inacessible to the control mechanism. We explored the elements of reversible control in an applied setting of wireless communications, through two different realistic examples, one of resource management in large antenna arrays, and one of wave time reversal in underwater acoustic communications [62].

In the first example [64], we perform antenna selection in a large distributed antenna array which serves as a distributed base station in a next generation cellular network: at any point in time, we want to use $n$ out of $m$ available antennas to serve $k < n$ users in the cell. The subset of antennas to be used is selected so to maximise the Shannon capacity of the communication channel between the base station and the users, which is a non-trivial optimisation task: selecting simply the antennas with the strongest signal does not help as they tend to be correlated and not contributing to the diversity in the channel. We propose a solution using reversing Petri nets [53] with controlled transitions: tokens (indicating antennas that are "on") move between places (antennas) based on simple calculations at the transitions (do the channel sum rates increase with the change of token position, i.e. reconfiguration of the array?) [54]. The results of experiments with varying number of users show that this distributed approach delivers results on par with computationally demanding centralised approaches, and tend to outperform the competition as the number of users increases. The approach we proposed here is not limited to the problem of antenna selection: in the ongoing work, we extend it to general resource management in wireless setting, using the advantages offered by having a reversible control algorithm, namely fault recovery, partial reversal of the system and repetitive motion handling [65].

In the second example, we focus on wave time reversal, the idea of reconstructing a wave (e.g. an acoustic pulse) by measuring the incoming wave at the boundary of a cavity and then re-transmitting the collected samples in reverse, producing a wave that reconverges at the original source [19]. It is straightforward to see how this scheme can be used to establish a communication channel, and

hence be used in a communication scheme in e.g. underwater acoustic communications. We selected sound propagation in water as an example of a reversible (but rarely reversed) medium under control, and proposed a reversible hardware architecture for this task [63]. Here we recognised another control challenge: disturbance compensation. If there is a source of disturbance in the medium (e.g. strong stream in the water) the reconstructed pulse will be distorted and hence the quality of communication will degrade. If we cannot remove the source of disturbance, but are in position to control a different part of the environment based on measurements from sensors in the medium, how can we improve the quality of wave time reversal? The more general question we pose here is whether control of a reversible medium is simpler than control of an irreversible one, and the model we chose to work on is one provided by reversible cellular automata. These automata, in the form of lattice gases, have been extensively used for fluid modelling. In cellular automata, the control problem revolves around the question of reaching a certain configuration from an arbitrary initial configuration [3]. In our consideration of reversible cellular automata, instead of observing the question of reaching a microstate, we investigate the problem of reaching a statistical macrostate in a region of the automaton [4]. The idea of reversible automata control being easier than the general automata control stems from the fact that states in reversible automata have unique predecessors, hence minimising the combinatorics of the arc of transition between an initial and a final state, which is an important element of cellular automata control.

## 8 Conclusions

We have summarised the main results obtained by the Working Group 2 on Software and System of the COST Action IC1405. In these four years the WG was active and produced important results, as witnessed by this document. Research in applying reversibility to software and systems is ongoing, and some of the guidelines and topics indicated in the MOU [67] were not exhaustively investigated during the lifetime of WG2. The interplay between reversibility and the so called recovery patterns deserves to be further investigated. Also, the integration of reversibility in software development is still at an early stage.

**Acknowledgement.** The WG2 has been led by Claudio Antares Mezzina and Rudolf Schlatte. For both of us, it has been an enormous honour to lead such WG, to organise the WG meetings and to interact with all the people involved in the working group. A witness of the liveness of the working group is the list of authors who happily contributed to this document. We would also thank Irek Ulidowski (chair) and Ivan Lanese (vice-chair) who wisely have led this COST Action and the Management Committee (MC) who appointed us as leader and co-leader (respectively) of this WG.

## References


1. Axelsen, H.B., Glück, R.: Reversible representation and manipulation of constructor terms in the heap. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 96–109. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38986-3_9
2. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_9
3. Bagnoli, F., Rechtman, R., El Yacoubi, S.: Control of cellular automata. Phys. Rev. E **86**(6), 066201 (2012)
4. Bagnoli, F., Siljak, H.: Control of reversible cellular automata (2019, Manuscript in preparation)
5. Barbanera, F., Lanese, I., de'Liguoro, U.: A theory of retractable and speculative contracts. Sci. Comput. Program. **167**, 25–50 (2018)
6. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. **17**(6), 525–532 (1973)
7. Bishop, P.G.: Using reversible computing to achieve fail-safety. In: Proceedings the Eighth International Symposium on Software Reliability Engineering, pp. 182–191, November 1997
8. Carlsson, R., et al.: Core Erlang 1.0.3. Language specification (2004). https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf
9. Carothers, C.D., Perumalla, K.S., Fujimoto, R.: Efficient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul. **9**(3), 224–253 (1999)
10. Cassar, I., Francalanza, A., Mezzina, C.A., Tuosto, E.: Reliability and fault-tolerance by choreographic design. In: Francalanza, A., Pace, G.J. (eds.) Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017. EPTCS, vol. 254, pp. 69–80 (2017)
11. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Concurrent reversible sessions. In: Meyer, R., Nestmann, U. (eds.) International Conference on Concurrency Theory, CONCUR 2017. LIPIcs, vol. 85, pp. 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
12. Chen, S.-K., Fuchs, W.K., Chung, J.-Y.: Reversible debugging using program instrumentation. IEEE Trans. Softw. Eng. **27**, 715–727 (2001)
13. Cservenka, M.H., Glück, R., Haulund, T., Mogensen, T.Æ.: Data structures and dynamic memory management in reversible languages. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 269–285. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_19
14. 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
15. 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
16. Dezani-Ciancaglini, M., Giannini, P.: Reversible multiparty sessions with checkpoints. In: Gebler, D., Peters, K. (eds.) Proceedings Combined 23rd International Workshop on Expressiveness in Concurrency and 13th Workshop on Structural Operational Semantics, EXPRESS/SOS 2016. EPTCS, vol. 222, pp. 60–74 (2016)

17. Elnozahy, E.N., Zwaenepoel, W.: Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit. IEEE Trans. Comput. **41**(5), 526–531 (1992)
18. Engblom, J.: A review of reverse debugging. In: System, Software, SoC and Silicon Debug, pp. 1–6. IEEE (2012)
19. Fink, M.: Time reversal of ultrasonic fields. I. Basic principles. IEEE Trans. Ultrason. Ferroelectr. Freq. Control **39**(5), 555–566 (1992)
20. 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
21. Francalanza, A., Mezzina, C.A., Tuosto, E.: Towards choreographic-based monitoring. In: Ferreira, C., Lanese, I., Schultz, U., Ulidowski, I. (eds.) Reversible Computation: Theory and Applications. LNCS, vol. 12070. Springer, Heidelberg (2020)
22. Fredkin, E., Toffoli, T.: Conservative logic. Int. J. Theor. Phys. **21**, 219–253 (1982)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
24. Gay, D., Aiken, A.: Language support for regions. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 70–80. ACM (2001)
25. 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
26. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. J. Log. Algebr. Methods Program. **88**, 99–120 (2017)
27. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. Comput. Softw. **33**(3), 108–128 (2016)
28. Glück, R., Yokoyama, T.: Constructing a binary tree from its traversals by reversible recursion and iteration. Inf. Process. Lett. **147**, 32–37 (2019)
29. Haulund, T., Mogensen, T.Æ., Glück, R.: Implementing reversible object-oriented language features on reversible machines. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 66–73. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_5
30. Hoey, J.: Reversing imperative concurrent programs. Ph.D. thesis, University of Leicester (2020)
31. Hoey, J., Lanese, I., Nishida, N., Ulidowski, I., Vidal, G.: A case study for reversible computing: reversible debugging. In: Ferreira, C., Lanese, I., Schultz, U., Ulidowski, I. (eds.) Reversible Computation: Theory and Applications. LNCS, vol. 12070. Springer, Heidelberg (2020)
32. Hoey, J., Ulidowski, I.: Reversible imperative parallel programs and debugging. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 108–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_7
33. Hoey, J., Ulidowski, I., Yuen, S.: Reversing imperative parallel programs. In: Peters, K., Tini, S. (eds.) Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS. EPTCS, vol. 255, pp. 51–66 (2017)
34. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: Pérez, J.A., Tini, S. (eds.) Proceedings Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics, EXPRESS/SOS. EPTCS, vol. 276, pp. 69–86 (2018)

35. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
36. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley, Boston (1998)
37. Landauer, R.: Irreversibility and heat generated in the computing process. IBM J. Res. Dev. **5**, 183–191 (1961)
38. 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
39. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr tracer website. https://github.com/mistupv/tracer/
40. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr website. https://github.com/mistupv/cauder
41. 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
42. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebr. Methods Program. **100**, 71–97 (2018)
43. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. In: Pérez, J.A., Yoshida, N. (eds.) FORTE 2019. LNCS, vol. 11535, pp. 167–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21759-4_10
44. Leeman Jr., G.B.: A formal approach to undo operations in programming languages. ACM Trans. Program. Lang. Syst. **8**(1), 50–87 (1986)
45. Letuchy, E.: Erlang at Facebook (2009). http://www.erlang-factory.com/conference/SFBayAreaErlangFactory2009/speakers/EugeneLetuchy
46. Mezzina, C.A., Pérez, J.A.: Reversible semantics in session-based concurrency. In: Proceedings of the 17th Italian Conference on Theoretical Computer 2016, Volume 1720 of CEUR Workshop Proceedings, pp. 221–226 (2016). CEUR-WS.org
47. Mezzina, C.A., Pérez, J.A.: Reversible sessions using monitors. In: Proceedings of the Ninth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016. EPTCS, vol. 211, pp. 56–64 (2016)
48. Mezzina, C.A., Pérez, J.A.: Causally consistent reversible choreographies: a monitors-as-memories approach. In: Vanhoof, W., Pientka, B. (eds.) Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, pp. 127–138. ACM (2017)
49. Mezzina, C.A., Pérez, J.A.: Reversibility in session-based concurrency: a fresh look. J. Log. Algebr. Methods Program. **90**, 2–30 (2017)
50. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: 26th International Conference on Compiler Construction, pp. 98–108. ACM (2017)
51. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 259–274. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_15
52. Perumalla, K.: Introduction to Reversible Computing. CRC Press, Boca Raton (2014)
53. 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

54. 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
55. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Logic Algebraic Program. **73**(1–2), 70–96 (2007)
56. 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
57. 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
58. Schordan, M., Oppelstrup, T., Jefferson, D., Barnes Jr., P.D., Quinlan, D.J.: Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In: SIGSIM-PADS 2016 (2016)
59. Schultz, U.P.: Reversible object-oriented programming with region-based memory management. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 322–328. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_22
60. Schultz, U.P., Axelsen, H.B.: Elements of a reversible object-oriented language. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 153–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_10
61. 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
62. Siljak, H.: Reversible computation in wireless communications. In: Ferreira, C., Lanese, I., Schultz, U., Ulidowski, I. (eds.) Reversible Computation: Theory and Applications. LNCS, vol. 12070. Springer, Heidelberg (2020)
63. Siljak, H., de Rosny, J., Fink, M.: Reversible hardware for acoustic wave time reversal. IEEE Commun. Mag. **58**(1), 55–61 (2020)
64. Siljak, H., Psara, K., Philippou, A.: Distributed antenna selection for massive MIMO using reversing Petri nets. IEEE Wirel. Commun. Lett. **8**(5), 1427–1430 (2019)
65. Siljak, H., Psara, K., Philippou, A.: Reversing Petri nets for resource management in wireless networks (2019, Manuscript in preparation)
66. Tofte, M., Talpin, J.-P.: Region-based memory management. Inf. Comput. **132**(2), 109–176 (1997)
67. Ulidowski, I.: IC1405 - Reversible Computation: extending horizons of computing - Memorandum of Understanding. https://e-services.cost.eu/files/domain_files/ICT/Action_IC1405/mou/IC1405-e.pdf
68. Undo Software: Undodb. Commercial reversible debugger. http://undo-software.com/
69. Vassor, M., Stefani, J.-B.: Checkpoint/Rollback vs causally-consistent reversibility. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 286–303. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_20
70. Vulov, G., Hou, C., Vuduc, R.W., Fujimoto, R., Quinlan, D.J., Jefferson, D.R.: The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In: WSC 2011 (2011)

71. 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
72. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29517-1_2
73. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation, Proceedings, pp. 144–153. ACM (2007)