Causal-Consistent Reversibility

Ivan Lanese, Claudio Antares Mezzina, Francesco Tiezzi

CAUSAL-CONSISTENT REVERSIBILITY*

Ivan Lanese†
Focus Team, University of Bologna/INRIA, Italy
ivan.lanese@gmail.com

Claudio Antares Mezzina
SOA Unit, FBK, Trento, Italy
mezzina@fbk.eu

Francesco Tiezzi
IMT Institute for Advanced Studies Lucca, Italy
francesco.tiezzi@imtlucca.it

Abstract

Reversible computing allows one to execute programs both in the standard, forward direction, and backward, going back to past states. In a concurrent scenario, the correct notion of reversibility is causal-consistent reversibility: any action can be undone, provided that all its consequences (if any) are undone beforehand. In this paper we present an overview of the main approaches, results, and applications of causal-consistent reversibility.

1 Introduction

Reversibility in computer science refers to the possibility of executing a program both in *forward* and *backward* directions. The former is the standard kind of execution, while the latter enables programs to go back to past states by undoing the effects of forward computations previously performed. Reversibility has been studied for different motivations. A first motivation dates back to the observation by Landauer [29] that only irreversible computations need to consume energy, thus reversible computations are interesting in a scenario of low-energy computing. Reversibility has also been studied for modeling systems which are naturally

*This work has been partially supported by the MIUR PRIN project CINA n. 2010LHT4KM.
†This author has been also partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

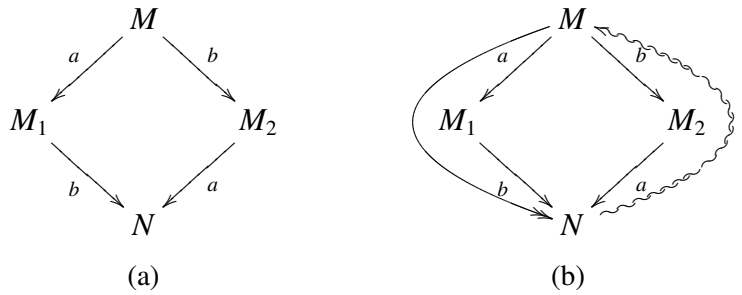

Figure 1: Example of causal-consistent executions

reversible. This is the case of biological or chemical systems, where the direction of computation depends on environment conditions such as temperature or pressure, but also of quantum computations [1], where most operations are naturally reversible. Another motivation comes from the desire to better explore a computation, analyzing different possibilities, as in debugging [21] or in state-space exploration, e.g., in Prolog. Recently [16, 30], reversibility has also been put forward as a common framework to study constructs for system reliability, such as transactions or checkpoints, with the idea that in case of error a system can automatically go backward to a more stable state.

Reversibility in a sequential setting is well understood (see, e.g., [34]). To reverse the execution of a sequential program one has simply to recursively undo the last action performed by the program. The definition of reversibility in a concurrent setting (including also distributed systems) is trickier, since there is no concept of "last action". Indeed, many actions are performed concurrently.

A suitable definition of reversibility in a concurrent scenario has been proposed by Danos and Krivine in their seminal paper [15]. Intuitively, the definition says that any action can be undone provided that all its consequences, if any, are undone beforehand. For instance, two threads that run without interactions can be reversed independently, while if a thread T_1 sends a message to thread T_2, the receipt from T_2, as well as all the consequent interactions, should be undone before the send from T_1 can be undone. This definition highlights the link between reversibility in a concurrent setting and causality, and is then called *causal-consistent*.

To better understand this crucial concept, consider the example in Figure 1. In Figure 1a, from state M there are two possible paths leading to state N, one executing first a and then b (on the left), and the other executing first b and then a (on the right). If the two actions a and b are concurrent, possibly executed by physically remote components, it may be difficult to distinguish the two computations. Thus one should be able to reverse any of the two executions by reversing the other, i.e. if the forward computation proceeds by executing first a and then b

(double-pointed arrow in Figure 1b), not only undoing first b and then a, but also undoing first a and then b (wave arrow in Figure 1b) is a valid reverse computation.

Given this definition of reversibility, a first line of research considered how to define reversible extensions of a given concurrent language. Actually, the research done so far on this topic concentrated mainly on process calculi such as CCS [36]. We discuss the different approaches in Section 2.

Those works focussed on defining backward transitions and computations, and on proving some relevant properties. They however provided no insight on when backward transitions should be preferred over forward ones and vice versa. A second line of research then considered different mechanisms to control reversibility. This research line is described in Section 3.

The descriptions above concentrated on the operational semantics of reversible processes. Later on, other aspects have been explored, such as the behavioral theory, the related logics, abstract representations as event structures or categories, the relations between different reversible and irreversible calculi, the cost of reversibility in terms of space, and the complexity of analyzing reversible systems. We discuss those aspects in Section 4.

With all these tools in place, causal-consistent reversibility was mature enough to be applied to various programming problems. Most of the applications have been found in the field of biological systems, of state-space exploration, of reliability and of debugging. Those applications are discussed in Section 5.

In Section 6 we summarize the state of research on causal-consistent reversibility and describe a few open challenges.

2 Uncontrolled Reversibility

In order to make a concurrent language reversible, two main issues have to be considered. First, as in sequential reversibility, history information has to be kept. This avoids that transitions cause loss of information, which would forbid to go back in a sound way. Second, information on causality has to be kept. This permits knowing which actions can be immediately reversed and which cannot according to the definition of causal-consistent reversibility. The first issue is clearly understood already in any sequential imperative language[1]. For instance, the assignment $x := 5$ deletes the old value of x, which needs to be stored if this assignment should be reversed. Note that this is not the case for any assignment: for instance, the assignment $x := x + 1$ causes no loss of information and can be reverted by executing $x := x - 1$ without the need for keeping history information. Actually,

[1]The first example of a reversible sequential programming model is the reversible Turing machine [5].

some approaches to reversibility restrict the language to actions of this second kind (called reversible updates [56]), although this is not the typical approach for causal-consistent reversibility. In process calculi the situation is even worst. For example, let us consider CCS [36]: a transition of the form $a.P + Q \xrightarrow{a} P$, executing action a and discarding alternative Q, will lose both the information on a and on Q, thus this information needs to be saved to enable reversibility.

Before discussing how the approaches in the literature deal with history information, let us discuss the issue of causality. As already said, concurrent actions as in $a \mid b$ can be reversed independently. Vice versa, dependent actions as in $a.b + b.a$ should be undone in reverse order of completion; e.g., if the left branch is chosen by executing first a and then b, action b should be undone before a. These two simple examples already provide many intuitions. First, reversibility distinguishes concurrency from non-determinism. Second, different instances of the same action may need to be distinguished, e.g., the two a actions in $a.b + b.a$ are different because they have different causal dependences. This explains why many approaches to reversibility [40, 32, 13] rely on unique indexes for processes. Last, actions in a thread are undone in reverse order of execution, while concurrent threads are reversed independently (unless there are synchronizations). This explains why in many approaches history information is attached to threads. Let us now discuss synchronization. Consider for instance the process $\nu b\,(a.b \mid \overline{b}.c)$. Here the only possible forward computation is given by an input on a, followed by a synchronization between b and $\overline{b}$, and by an input on c. Here, a and c are in different threads, and do not interact directly, however c blocks the undo of a via the synchronization on b.

Even in this simple setting, a few subtleties emerge, and one can ask whether there are correctness criteria for causal-consistent reversibility. In other words, given a language equipped with its reversible semantics, how can one check whether this is indeed a correct causal-consistent reversible semantics? Two properties have been put forward in the literature since [15].

The first property, called *Loop Lemma*, says that each transition has an inverse. Formally, for each forward transition $t : P \xrightarrow{\alpha}_f Q$ there is a backward transition $t^{-1} : Q \xrightarrow{\alpha^{-1}}_b P$ and vice versa, where α^{-1} is the inverse label of α and subscripts f and b are used to distinguish $f$orward transitions from $b$ackward ones. An obvious consequence of the Loop Lemma is that any (non trivial) program in a language with uncontrolled reversibility can diverge by doing and undoing the same action forever.

The second property, called *causal-consistency*, relates reversibility and concurrency. Indeed, it is based on a notion of concurrent transitions. If two transitions $t_1 : P \xrightarrow{\alpha_1} R_1$ and $t_2 : P \xrightarrow{\alpha_2} R_2$ starting from the same process P are concurrent then there exist two transitions $t_1/t_2 : R_2 \xrightarrow{\alpha_1} Q$ and $t_2/t_1 : R_1 \xrightarrow{\alpha_2} Q$

closing the square. One can define *causal equivalence*, written $\sim$, as the minimal equivalence on traces closed under composition and satisfying:

$$t_1; t_2/t_1 \sim t_2; t_1/t_2 \qquad t; t^{-1} \sim \epsilon \qquad t^{-1}; t \sim \epsilon$$

where ϵ is an empty trace and ; the sequential composition of traces. Causal-consistency says that two coinitial traces are cofinal iff they are causal equivalent. While quite obscure at first sight, this definition captures the essence of causal-consistent reversibility. Indeed, two computations starting from the same state (coinitial) that are causal equivalent should end in the same state (cofinal), since exchanging the order of concurrent actions and doing and undoing the same action should not change the final state. Vice versa, computations that differ for any other reason should not lead to the same state, i.e. should not be confused since they have different pasts. For instance, the two computations leading from $a.b + b.a$ to 0 should not be confused, since they have different pasts, and indeed history information should be added to the final state 0 to disambiguate the two possible histories.

Another interesting property is the so-called *Parabolic Lemma*, which states that each reversible computation can be decomposed into a backward computation followed by a forward one. Intuitively, this means that, up to causal equivalence, the process first goes backward to enable as many choices as possible, and only then goes forward. Consider, for instance, the CCS process $a+b \mid c.d$. Assume that we are in a state in which the process has performed the action a, that is the process has now the form $c.d$ (where, for the sake of simplicity, we omit the history information about the execution of action a). Assume also that the process $c.d$ proceeds with the computation $\xrightarrow{c}_f \xrightarrow{d}_f \xrightarrow{a^{-1}}_b$. According to the Parabolic Lemma, we can decompose the above trace into a backward one followed by a forward one, that is $\xrightarrow{a^{-1}}_b \xrightarrow{c}_f \xrightarrow{d}_f$. Notably, after the backward transition $\xrightarrow{a^{-1}}_b$, the process goes back to its initial state, where all possible forward computations are enabled.

Finally, since the reversible language is normally a reversible extension of an existing language, a correspondence theorem shows that the forward semantics of the reversible language indeed coincides with the standard semantics of the underlying language.

Given these premises, the works in the literature can be mainly classified according to which underlying language they consider, and whether they deal with a reduction semantics or a LTS one. Other differences lie in the technical means used for storing history and causality information. The approaches in the literature are summarized in Table 1.

Reversible CCS (RCCS) [15] is the first proposal of a causal-consistent reversible calculus. It introduces the notion of memories attached to threads to keep history information. In RCCS memories are also used as thread identifiers.

Table 1: Process calculi with uncontrolled reversibility

	Language	**Memories**	**Identifiers**	**Fork handling**	**Semantics**
RCCS [15]	CCS [36]	stacks	memories as ids	fork numbers	labeled
CCS-R [14]	CCS [36]	stacks	thread ids	empty memories	labeled
CCSK [40]	CCS [36]	– (static rules)	communication keys	–	labeled
$\rho\pi$ [32]	HOπ [47]	non-structured	thread ids	structured tags	reduction
Rπ [13]	π-calculus [37]	stacks	event ids	fork symbol	labeled
Reversible structures [8]	DSD [39]	– (pointers to next operations)	signal ids	–	reduction
RμOz [35]	μOz [35]	stacks	thread ids	–	reduction

CCS-R [14] extends RCCS to deal with recursion, and uses unique names to identify threads.

CCS with communication Keys (CCSK) [40] is the instance on CCS of a general procedure to make calculi, specified in a simple path format, reversible. The main novelty of the approach is that the structure of processes is not consumed, but simply annotated as executed. This is obtained by making all the rules defining the semantics static. Thus, no memories are needed. The approach can be applied to other calculi without name passing, such as ACP [6] or CSP [7], but it is not suitable for name-passing calculi, as the π-calculus.

The reversible higher-order π-calculus ($\rho\pi$) [32] is a reversible extension of the higher-order π-calculus [47]. Notably, differently from the approaches mentioned before, the semantics of $\rho\pi$ is only given in a reduction style.

A reversible extension of π-calculus is Rπ [13]. Rπ at the moment is the only reversible calculus with name passing that provides a labeled semantics. A main feature of Rπ is that two transitions are concurrent only if there is a context where the reductions generated by those two transitions are actually concurrent.

Reversible structures [8] are a simple computational calculus for modeling chemical systems composed by signals and gates. The gate structure is not consumed, and a cursor reminds which part of the gate already interacted. The novelty of this setting is that signal identifiers are not necessarily unique, following the idea that two instances of the same molecule are indistinguishable.

Finally, RμOz [35] is a reversible abstract machine for a functional language with threads and asynchronous communication via ports.

3 Controlling Reversibility

The works discussed above specify *how* a system can reverse a forward computation and what kind of information it should exploit, but they give no hint about *when* forward execution should be preferred over backward one and vice versa. At most, one can non-deterministically choose whether to go forward or backward, obtaining a system that may well diverge by doing and undoing the same action again and again. Those works have been useful to understand the basics of causal-consistent reversibility, but they are not directly suitable for most applications.

Indeed, in practice, the direction of computation is determined by various conditions, related to the application area. For instance, in biology, temperature and pressure conditions determine the chosen direction of computation. In state space exploration, normal computation is forward and backtracking is used when the forward computation gets stuck since no solution can be found from the reached state. Similarly, when programming reliable systems, backtracking is used when an error state is reached.

Those strategies can be implemented using different mechanisms. We describe below the main mechanisms that have been considered, following the categorization in [33].

Internal control: specific commands inside processes specify whether the process itself should go forward or backward. A possibility along this line has been explored in [16], where *irreversible* actions, i.e. actions that once performed cannot be undone, have been integrated in RCCS. The idea underlying irreversible actions is that, when the exploration reaches a desirable result, backtracking is disallowed by executing an irreversible action. In other words, an irreversible action acts as a sort of *commit*. A dual approach has been proposed in roll-π [31], an extension of $\rho\pi$. There, normal computation is forward, and an explicit *rollback* primitive is used to trigger backward execution. In this way, when an error state is reached, the rollback primitive can be used to go back to a past consistent state. To specify how far back to go, the rollback primitive takes as parameter a reference to a past action, and it undoes all (and only) the actions causally dependent on it. Note that, in a concurrent scenario, a (possibly more intuitive) behavior such as "go back n steps" is not meaningful, since there may not be a unique choice about which the last n actions have been. Irreversible actions and explicit rollback are dual, the former specifying when it is *forbidden* to go back, and the latter specifying when it is *required* to go back. The roll-π approach is satisfactory to deal with transient errors, but not with permanent errors. In this second case, rollback leads back to a past state, where the same computation can restart, leading again to the same error state. To

solve this problem, a construct, called *alternative*, is introduced in [30] to specify a (slightly) different behavior to be chosen after a rollback.

External control: this approach follows the separation of concerns principle: a process is potentially able to go both backward and forward, while another process is in charge of controlling it by deciding when it has to go backward and when it has to go forward. Such an approach is suitable, e.g., for hierarchical component-based systems, where the father component may decide when and how to roll-back its children, and the children notify the father in case of errors. Such a hierarchical structure for failure handling is typically the one advocated for Erlang systems [2]. External control naturally emerges in a reversible *debugger* such as the one proposed in [23] for μOz [35]: the user, through the debugger interface, decides whether the program under debugging should execute forward or should get back to a previous state. One can either go back step-by-step, and the debugger ensures that only actions with no causal dependences can be undone, or can choose a past action and undo it and all its consequences in a causal-consistent way, as done by the rollback primitive of roll-π described above. This is in contrast, e.g., to [12], where the user has to decide which actions to undo and in which order. External control has been applied also to biological reversible systems in [44]. There, a reversible CCS process P is controlled by a *controller* process C, which is again a CCS process. The controller C always computes forward, and it constrains the possible actions of P, thus decreasing the non-determinism due to reversibility and to concurrency. Together with a generalized form of prefix, this allows one to model different forms of reversibility, including reversibility that is not (always) causally consistent.

Semantic control: in this approach the semantics of the language is extended with guidelines on whether to go forward or to go backward. Consider the following scenario: a reversible program is used to perform a state-space exploration looking for some solution of a given problem. In this case reversibility is needed to backtrack in case a branch with no solution has been taken. One can imagine to add to the history information about whether and how many times a particular path has been taken and favor paths (and directions of execution) leading to less explored areas. For instance, one can label each action with the number of times it has been tried, and choose among the enabled actions (both forward and backward) one with the minimal value. It is clear that in such a way a finite state space is completely explored, allowing to find a solution if at least one exists. A refined approach has been explored in [4]. There RCCS is equipped with a probabilistic

Table 2: Process calculi with controlled reversibility

Mechanism	Calculus	Category
Irreversible actions [16]	RCCS [15]	Internal
Roll [31]	$\rho\pi$ [32]	Internal
Roll+alternatives [30]	$\rho\pi$ [32]	Internal
Debugger [23]	RμOz [35]	External
Controller [44]	CCSK [40]	External
Potential [4]	RCCS [15]	Semantic

semantics, where the rate of forward and backward steps is determined by energy *potential*. The paper shows that if the potential satisfies suitable conditions then a solution, if it exists, is reached in a finite amount of time even in an infinite state space. The main point is to avoid that the computation diverges in an infinite branch without solutions.

We conclude the section by summarizing the main approaches described above in Table 2.

4 Theoretical Results

We described till now the calculi that have been proposed to model reversible systems. The formalization enabled further studies, concerning in particular the behavioral theory, the related logics, abstract representations as event structures or categories, the relations between different reversible and irreversible calculi, the cost of reversibility in terms of space, and the complexity of analyzing reversible systems.

Concerning behavioral theory, the seminal paper [19] considers back and forth bisimulation, a variation of bisimulation where processes should match not only in the forward direction, but also in the backward one. It shows that in principle this allows one to distinguish concurrency from non-determinism. In fact, in $a \mid b$ one can do a, then b and then either undo a and then b or b and then a. Instead, in $a.b+b.a$, if one does a and then b, it is obliged to undo first b and then a. However, [19] quickly dismissed this setting by obliging processes to go back following the same path they were coming from. With this restriction, back and forth strong bisimilarity is as expressive as standard strong bisimilarity. This is not the case for other equivalences.

In causal-consistent reversible calculi, processes can go back along causal-equivalent paths. In this setting, the natural notion of bisimilarity allows forward (resp. backward) moves to match only forward (resp. backward) ones. This relation is more discriminating than standard strong bisimilarity for the reason above.

Indeed, [42] has shown that (in the context of stable configuration structures) this notion of reversible bisimilarity is as strong as hereditary history-preserving bisimilarity [26] (in the absence of *auto-concurrency*, i.e. when concurrent events do not have the same label).

Similar results are obtained also for logics. An extension of Hennessy-Milner Logic [25] with backward modalities has been studied in [38], which shows that, in the absence of auto-concurrency, the extended logic characterizes the hereditary history-preserving bisimulation [26]. In [41], an extension of Hennessy-Milner Logic with reverse modalities and event identifiers, used to distinguish the undo of different events with the same label, is proved to characterize hereditary history-preserving bisimulation even in presence of auto-concurrency. Sublogics of this logic are proved to characterize history-preserving bisimulation, weak history-preserving bisimulation and hereditary weak history-preserving bisimulation (see [53] for a description of these logics).

The above results about logics are obtained by defining backward moves inside standard stable event structures. A first notion of reversible event structure has been proposed in [45], by adding reversing events to Winskel event structures [55]. The approach is refined in [43, 51] by describing backward steps as the removal of an event from the current configuration. The approach is applied to define both reversible prime event structures and reversible asymmetric event structures. The expressive power of the two kinds of structures is studied. A main point of the approach is the possibility of specifying both causal-consistent and non causal-consistent reversibility.

An abstract model of reversibility based on category theory has been studied in [17]. The results therein essentially generalize the ones in [16], showing how given a category of computations including both reversible and irreversible actions one can define in a unique way (up-to isomorphisms) a category of histories allowing to correctly define causal-consistent reversible computations. Furthermore, it shows that the computations of systems with histories are essentially the same as the causal computations of the original system, i.e. the computations where only those reversible actions needed to execute irreversible actions are performed.

Given that many reversible languages exist in the literature, a natural question is the relationship between them, and with the non reversible languages. A main result in this setting has been presented in [32], where $\rho\pi$ has been encoded into the higher-order π-calculus. The correctness of the encoding has only been proved up-to weak barbed equivalence (mixing forward and backward reductions), but the actual correspondence is tighter. This is the only encoding of a causal-consistent reversible calculus into a non reversible one we are aware of. Concerning encodings of reversible calculi into other reversible calculi, in [8] an encoding of asynchronous RCCS [15] into reversible structures is provided. An operational correspondence result shows that RCCS steps are correctly matched.

A separation result has been proved in [30], showing that the introduction of alternatives strictly increases the expressive power of the rollback operator of roll-π [31]. The separation result shows that no encoding can preserve both the possibility of performing a backward step, and termination.

The cost of irreversibility in terms of space has been studied in [35]. Indeed, this paper proves that reversible μOz stores history information which is linear in the number of steps for any given program. More importantly, it shows that in order to enable causal-consistent reversibility an amount of history information which is linear in the number of steps is actually needed in the worst case. The main point is that with less information one is not able to distinguish different pasts leading to the same state which are not causally equivalent.

Reversibility has been proved useful for specifying and proving correct systems based on state-space exploration. Indeed, classically one has to specify both the exploration strategy and how to escape from states from where no solution can be found. In a reversible system, only the exploration strategy needs to be specified, since the reversibility mechanism can be used to escape undesired states. This leads to smaller specifications, which are easier both to understand and to verify correct as shown in [18, 28].

The cost of deciding reachability has been studied in reversible structures [8], and proved quadratic in the number of gates for coherent reversible structures (where atoms have unique identifiers). Only an EXPSPACE bound is known for weak coherent reversible structures (where identifiers are not unique).

5 Applications

Causal-consistent reversibility is quite young. Nevertheless the first applications are emerging in different areas, in particular in the fields of biological systems, state-space exploration, reliability and debugging.

The modeling of systems and reactions from biology and biochemistry has been highlighted as an interesting application area for causal-consistent reversibility since the seminal work presented in [15]. The first real application has been presented in [14], where CCS-R is used to model a system in which the transcription of a protein is controlled by a competition between different reactants. The modeling of the beginning of the RTK-MAPK cascade [46] is also discussed. The Ras/Raf-1/MEK/ERK signaling pathway [9], which delivers mitogenic and differentiation signals from the membrane of a cell to its nucleus, has been modeled in CCSK with a control operator in [44]. A main point here is that reversibility is not causal-consistent. For instance, in a reaction it happens that an event a causes b, b causes c and c causes d. Now, according to causal-consistent reversibility, undoing the event a would cause the undoing of the other three events, but this is

not what happens in practice. The same application is also considered in [45], and modeled with a calculus relying on an explicit notion of bonding. In [43] the basic catalytic cycle for protein substrate phosphorylation by a kinase [50] is described using reversible event structures.

Another area where reversibility has been applied is state-space exploration. Indeed, one of the first languages with rollback capabilities is Prolog, which explores the state-space of SLD-derivations looking for a solution for a given goal. A first application of causal-consistent reversibility for state-space exploration has been presented in [18], where an algorithm for creating a set of trees from nodes equipped with their arities is specified in RCCS. In [28] a RCCS model of the classic dining philosophers problem is used as a benchmark for the efficiency of checking the conformance to a given specification. Finally, in [30] a concurrent solution of the 8-queens problem is presented, and then implemented in Maude [11]. These examples are noteworthy not for their performances (the ones in RCCS may even not terminate), but for their concise and clear definition, which takes advantage of the fact that backtracking is available in the language semantics. Also, they all refer to highly distributed algorithms, where a global solution is obtained by a set of peers through binary synchronizations. Indeed, this is the setting where reversibility shines, since the peers may easily reach states from where no solution can be found, and backtracking is needed to avoid deadlocks.

The first contribution on reversibility concerning constructs for programming reliable systems is in [16]. There, irreversible actions are introduced in RCCS. This gives rise in a natural way to a very simple notion of transaction. Indeed, while executing only reversible actions a process is conceptually executing inside a transaction, which can be aborted at any time simply by undoing the performed actions. Irreversible actions act as a commit, since they make the performed actions permanent (durability of transactions). In particular, all the actions on which the irreversible action depends are committed in this way.

This first attempt at modeling transactions is quite basic, since, e.g., it only considers non-nested transactions. A more refined approach is presented in [30]. There, CommTrans [20], a calculus featuring a notion of interacting transactions with compensation, is encoded in roll-π with alternatives. These transactions allow the transactional process to interact with the environment. In case of abort, the changes performed on the environment are undone, while the body of the transaction is replaced by a compensation. The same behavior can be obtained in roll-π with alternatives by rolling back the initial action of the transaction, and using alternatives to encode compensations. The correctness and completeness of the encoding is proved resorting to an ad hoc notion of bisimulation. Interestingly, roll-π has a more precise management of causality information than CommTrans, and this allows it to undo only those actions which are causally related to the transaction body, while CommTrans also features some spurious undo of unre-

lated actions.

Reversible debugging has been studied in the sequential setting since a long time [24, 57], and today various sequential reversible debuggers exist. Even GDB [22] provides support for reversible debugging since version 7. However, most of the available reversible debuggers do not support concurrency. A few of them do, and they can be divided in two classes:

Log-based: [10, 27] record the exact order of execution of actions, and they undo them exactly in reverse order of execution;

Without log: [54, 3, 52] replay the program till the desired point, but may use a different scheduling each time, thus potentially generating a different behavior, which may not highlight the same bugs.

According to the causal-consistent approach, actions should be undone in any order which is compatible with the causal dependencies. Such an approach has been explored in [23]. The main primitives considered there are the undo of a single action in a given thread, which is enabled only if the action has no dependences, and a primitive for undoing a given action including all its dependences. The semantics of this primitive essentially coincides with the one of the rollback primitive of roll-π. The primitive comes with different interfaces, which allow one to choose the action to be undone according to the kind of wrong behavior that is observed. For instance, if an unexpected message is seen, the interface allowing one to undo the send of a given message can be used. Similar interfaces allow one to undo the creation of a variable, the receipt of a message and the creation of a thread. Equipped with these primitives, the debugging strategy consists in finding a misbehavior and recursively undo the actions that may have caused it. Interestingly, the debugger autonomously finds which thread these actions belong to, and which are the dependencies. The only other work considering causality information for debugging is CauseWay [48], which is only a post-mortem trace analyzer, with no execution capabilities.

6 Conclusion

Causal-consistent reversibility, introduced in [15], is arguably the correct notion of reversibility in a concurrent scenario. Since its definition, different approaches and applications of causal-consistent reversibility have been considered. Nonetheless, the field is still at an initial stage, and while interesting ideas have been proposed, a systematic study of the related problems and of the possible application areas is still missing. Concerning the definition of uncontrolled reversibility, different approaches exist and have been outlined in Section 2, but the relationships among

them are not clear. The only approach trying to deal with multiple calculi in a uniform way [40] poses quite strict constraints on them. Furthermore, all these approaches deal with toy languages, with [35] probably being the one more in the direction of real languages. Indeed, the interplay between reversibility and various language features, with dynamic data structures, objects, error handling and type systems[2] being probably the most relevant, is still not understood. This would be useful, e.g., to write causal-consistent debuggers for real languages.

For controlled reversibility, different control mechanisms have been proposed. However, how they relate and compose and whether they are enough to specify all the needed control policies, or even which the needed control policies are, is not clear.

Coming to the theoretical treatment, different interesting aspects have been studied, but most of the approaches are tailored to one specific calculus, and it is not clear whether they can be applied in more general cases. A main open point is understanding which is the good notion (or, possibly, which are the good notions) of behavioral equivalence to reason about reversible systems. In the strong case, back and forth bisimilarity, allowing to undo actions according to any causal-consistent path, is a reasonable candidate. In the weak case, many more possibilities exist, since, e.g., it is not clear if backward (resp. forward) τ actions should be allowed when matching forward (resp. backward) actions. Also, reversible processes are equipped with history information, and it is not clear how to deal with this information when doing axiomatic reasoning on reversible programs.

Coming to applications, the preliminary results described in Section 5 are encouraging, but a deeper exploration is needed. For instance, in the field of constructs for reliable systems, the relation between causal-consistent reversibility and distributed checkpointing is still unclear. Also, it seems reasonable that reversibility can be handy to deal with software transactional memories, but a precise relationship has not been found yet. In the biological field, it seems that causal-consistent reversibility is not enough to capture all the observed behaviors, but how to generalize it to cope with them deserves further investigation.

References

[1] T. Altenkirch and J. Grattage. A functional quantum programming language. In *LICS*, pages 249–258. IEEE Computer Society, 2005.

[2] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, KTH, Stockholm, Sweden, 2003.

[2] A preliminary exploration of the interplay between reversibility and session types is presented in [49].

[3] K. Arya, T. Denniston, A. M. Visan, and G. Cooperman. FReD: Automated debugging via binary search through a process lifetime. *CoRR*, abs/1212.5204, 2012.

[4] G. Bacci, V. Danos, and O. Kammar. On the statistical thermodynamics of reversible communicating processes. In *CALCO*, volume 6859 of *LNCS*, pages 1–18. Springer, 2011.

[5] C. Bennett. Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[6] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1-3):109–137, 1984.

[7] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *J. ACM*, 31(3):560–599, 1984.

[8] L. Cardelli and C. Laneve. Reversible structures. In *CMSB*, pages 131–140. ACM, 2011.

[9] K.-H. Cho, S.-Y. Shin, H. W. Kim, O. Wolkenhauer, B. McFerran, and W. Kolch. Mathematical modeling of the influence of RKIP on the ERK signaling pathway. In *CMSB*, volume 2602 of *LNCS*, pages 127–141. Springer, 2003.

[10] Chronon Systems. Commercial reversible debugger. `http://chrononsystems.com/`.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[12] J. J. Cook. Reverse execution of java bytecode. *Comput. J.*, 45(6):608–619, 2002.

[13] I. Cristescu, J. Krivine, and D. Varacca. A compositional semantics for the reversible pi-calculus. In *LICS*, pages 388–397. IEEE Computer Society, 2013.

[14] V. Danos and J. Krivine. Formal molecular biology done in CCS-R. In *BioConcur*, volume 180(3) of *ENTCS*, pages 31–49. Elsevier, 2003.

[15] V. Danos and J. Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *LNCS*, pages 292–307. Springer, 2004.

[16] V. Danos and J. Krivine. Transactions in RCCS. In *CONCUR*, volume 3653 of *LNCS*, pages 398–412. Springer, 2005.

[17] V. Danos, J. Krivine, and P. Sobocinski. General reversibility. In *SOS*, volume 175(3) of *ENTCS*, pages 75–86. Elsevier, 2006.

[18] V. Danos, J. Krivine, and F. Tarissan. Self-assembling trees. In *SOS*, volume 175(1) of *ENTCS*, pages 19–32. Elsevier, 2007.

[19] R. De Nicola, U. Montanari, and F. W. Vaandrager. Back and forth bisimulations. In *CONCUR*, volume 458 of *LNCS*, pages 152–165. Springer, 1990.

[20] E. de Vries, V. Koutavas, and M. Hennessy. Communicating transactions. In *CONCUR*, volume 6269 of *LNCS*, pages 569–583. Springer, 2010.

[21] S. I. Feldman and C. B. Brown. Igor: A system for program debugging via reversible execution. In *Workshop on Parallel and Distributed Debugging*, 1988.

[22] GDB. The GNU Project Debugger. `http://www.gnu.org/software/gdb/`.

[23] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In *FASE*, volume 8411 of *LNCS*, pages 370–384. Springer, 2014.

[24] R. Grishman. The debugging system AIDS. In *AFIPS Spring Joint Computing Conference*, volume 36 of *AFIPS Conference Proceedings*, pages 59–64. AFIPS Press, 1970.

[25] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[26] A. Joyal, M. Nielsen, and G. Winskel. Bisimulation from open maps. *Information and Computation*, 127(2):164–185, 1996.

[27] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference, General Track*, pages 1–15, 2005.

[28] J. Krivine. A verification technique for reversible process algebra. In *RC*, volume 7581 of *LNCS*, pages 204–217. Springer, 2012.

[29] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, 1961.

[30] I. Lanese, M. Lienhardt, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Concurrent flexible reversibility. In *ESOP*, volume 7792 of *LNCS*, pages 370–390. Springer, 2013.

[31] I. Lanese, C. A. Mezzina, A. Schmitt, and J.-B. Stefani. Controlling reversibility in higher-order pi. In *CONCUR*, volume 6901 of *LNCS*, pages 297–311, 2011.

[32] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Reversing higher-order pi. In *CONCUR*, volume 6269 of *LNCS*, pages 478–493. Springer, 2010.

[33] I. Lanese, C. A. Mezzina, and J.-B. Stefani. Controlled reversibility and compensations. In *RC*, volume 7581 of *LNCS*, pages 233–240. Springer, 2012.

[34] G. B. Leeman. A formal approach to undo operations in programming languages. *ACM Trans. Program. Lang. Syst.*, 8(1):50–87, 1986.

[35] M. Lienhardt, I. Lanese, C. A. Mezzina, and J.-B. Stefani. A reversible abstract machine and its space overhead. In *FMOODS/FORTE*, volume 7273 of *LNCS*, pages 1–17. Springer, 2012.

[36] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[37] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[38] M. Nielsen and C. Clausen. Bisimulation for models in concurrency. In *CONCUR*, volume 836 of *LNCS*, pages 385–400. Springer, 1994.

[39] A. Phillips and L. Cardelli. A programming language for composable DNA circuits. *Journal of the Royal Society Interface*, 6(S4), 2009.

[40] I. Phillips and I. Ulidowski. Reversing algebraic process calculi. *J. Log. Algebr. Program.*, 73(1-2), 2007.

[41] I. Phillips and I. Ulidowski. A logic with reverse modalities for history-preserving bisimulations. In *EXPRESS*, volume 64 of *EPTCS*, pages 104–118, 2011.

[42] I. Phillips and I. Ulidowski. A hierarchy of reverse bisimulations on stable configuration structures. *Mathematical Structures in Computer Science*, 22(2):333–372, 2012.

[43] I. Phillips and I. Ulidowski. Reversibility and asymmetric conflict in event structures. In *CONCUR*, volume 8052 of *LNCS*, pages 303–318. Springer, 2013.

[44] I. Phillips, I. Ulidowski, and S. Yuen. A reversible process calculus and the modelling of the ERK signalling pathway. In *RC*, volume 7581 of *LNCS*, pages 218–232. Springer, 2012.

[45] I. Phillips, I. Ulidowski, and S. Yuen. Modelling of bonding with processes and events. In *RC*, volume 7948 of *LNCS*, pages 141–154. Springer, 2013.

[46] A. Regev, W. Silverman, and E. Y. Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.

[47] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis CST–99–93, University of Edinburgh, 1992.

[48] T. Stanley, T. Close, and M. S. Miller. Causeway: a message-oriented distributed debugger. Technical report, HPL-2009-78, 2009. Available at `http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html`.

[49] F. Tiezzi and N. Yoshida. Towards reversible sessions. In *PLACES*, volume 155 of *EPTCS*, pages 17–24, 2014.

[50] Ubersax Jeffrey A. and Ferrell Jr James E. Mechanisms of specificity in protein phosphorylation. *Nat. Rev. Mol. Cell. Biol.*, 8(7):530–541, 2007.

[51] I. Ulidowski, I. Phillips, and S. Yuen. Concurrency and reversibility. In *RC*, volume 8507 of *LNCS*, pages 1–14. Springer, 2014.

[52] Undo Software. Commercial reversible debugger. `http://undo-software.com/`.

[53] R. J. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Inf.*, 37(4/5):229–327, 2001.

[54] A. M. Visan et al. Temporal debugging using URDB. *CoRR*, abs/0910.5046, 2009.

[55] G. Winskel. Event structures. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.

[56] T. Yokoyama, H. B. Axelsen, and R. Glück. Principles of a reversible programming language. In *5th Conference on Computing Frontiers*, pages 43–54. ACM, 2008.

[57] M. V. Zelkowitz. Reversible execution. *Commun. ACM*, 16(9):566, 1973.