Towards a Taxonomy for Reversible Computation Approaches*

Robert Glück[1][0000−0001−6990−3935], Ivan Lanese[2][0000−0003−2527−9995], Claudio Antares Mezzina[3][0000−0003−1556−2623], Jarosław Adam Miszczak[4][0000−0001−8790−101X], Iain Phillips[5][0000−0001−5013−5876], Irek Ulidowski[6,7][0000−0002−3834−2036], and Germán Vidal[8][0000−0002−1857−6951]

[1] DIKU, Department of Computer Science, University of Copenhagen, Denmark
[2] Focus Team, University of Bologna/INRIA, Italy
[3] Dipartimento di Scienze Pure e Applicate, Università di Urbino, Urbino, Italy
[4] Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, Gliwice, Poland
[5] Imperial College London, England
[6] Department of Applied Informatics, AGH, Kraków, Poland,
[7] SCMS, University of Leicester, England
[8] MIST, VRAIN, Universitat Politècnica de València, Spain

Abstract. Reversible computation is a paradigm allowing computation to proceed not only in the usual, forward direction, but also backwards. Reversible computation has been studied in a variety of models, including sequential and concurrent programming languages, automata, process calculi, Turing machines, circuits, Petri nets, event structures, term rewriting, quantum computing, and others. Also, it has found applications in areas as different as low-power computing, debugging, simulation, robotics, database design, and biochemical modeling. Thus, while the broad idea of reversible computation is the same in all the areas, it has been interpreted and adapted to fit the various settings. The existing notions of reversible computation however have never been compared and categorized in detail. This work aims at being a first stepping stone towards a taxonomy of the approaches that co-exist under the term reversible computation. We hope that such a work will shed light on the relation among the various approaches.

Published as *Glück, R, Lanese, I., Mezzina, C.A., Miszczak, J.A., Phillips, I., Ulidowski, I., and Vidal, G. (2023). Towards a Taxonomy for Reversible Computation Approaches. In: Kutrib, M., Meyer, U. (eds) Reversible Computation. RC 2023. Lecture Notes in Computer Science,* to appear*. Springer Cham.*

* I. Lanese has been partially supported by French ANR project DCore ANR-18-CE25-0007 and INdAM-GNCS Project CUP_E55F22000270001 "Proprietà qualitative e quantitative di sistemi reversibili". I. Ulidowski has been partially supported by JSPS Fellowship grant S21050. G. Vidal has been partially supported by grant PID2019-104735RB-C41 funded by MCIN/AEI/ 10.13039/501100011033. J.A. Miszczak has been partially supported by NCN grant 2019/33/B/ST6/02011.

Keywords: Reversible computing · Taxonomy · Models and languages.

1 Introduction

Reversible computation is a paradigm considering computation to proceed not only in the usual, forward direction, but also backwards [?, ?, ?]. Reversible computation has been studied in a variety of models, including sequential programming languages (both imperative [?, ?, ?, ?], functional [?, ?, ?, ?], and object-oriented [?, ?, ?]), concurrent programming languages [?, ?, ?], process calculi [?, ?, ?, ?], universal logic elements [?, ?, ?], Turing machines [?, ?, ?, ?], automata [?, ?], cellular automata [?, ?, ?, ?], modal logics [?], Petri nets [?, ?, ?], event structures [?], term rewriting [?, ?], Markov chains [?], circuits [?, ?], and others. Also, it has found applications in areas as different as low-power computing [?, ?], debugging [?, ?], bidirectional transformations [?, ?], database design [?], simulation [?], robotics [?, ?], quantum computing [?, ?], and biochemical modeling [?, ?, ?, ?]. In some of those applications, including quantum computing, the reversibility of the computational process is enforced by the very nature of the physical process of computation. In some other areas, the reversibility is treated as a crucial feature, implemented, for example, by database transactions. Thus, while the broad idea of reversible computation is the same in all the areas, it has been interpreted and adapted to fit the various settings. The existing notions of reversible computation however have never been compared and categorized in detail. This work aims at being a first stepping stone towards a taxonomy of the approaches that co-exist under the term reversible computation.

We remark that defining a taxonomy for a field as heterogeneous as reversible computation is a very difficult task, and as far as we are aware this is the first effort in this direction. As such, we provide a *possible* classification, with the aim to start the discussion. We do not claim that our taxonomy is the final word on the subject, and indeed other dimensions may be worth considering, in addition to or instead of some of the ones that we propose (cf. Section 2), and of course we do not claim to be complete on our coverage of models[9] and languages (cf. Section 3). Indeed, our examples are concentrated in the area of formal methods and programming languages, which are the main expertise of most of the authors, and we may have missed very significant examples from other areas.

2 Taxonomy

In this section we present our taxonomy for approaches in the area of reversible computation. The taxonomy includes six *dimensions*, and for each dimension

[9] In the following we mainly use the term model to refer to the instances of reversible computation that we consider. Indeed, many of our examples are (formal) models. However, we think that our taxonomy can be applied also to more concrete entities, such as languages, applications or systems.

we describe different *positions* that a given approach may fit. In many cases, positions on a dimension are ordered, in the sense that one generalizes the other. Hence, of course, if an approach fits a position it fits also all the more general ones. We write (dim, pos) to refer to position pos in dimension dim. We write (d1, p1) $\implies$ (d2, p2) to say that pair (d1, p1) implies (d2, p2), and dually the latter is a generalization of the former.

We note that the same approach may fit different positions, depending on the level of abstraction. Vice versa, very different models may fit the same position in the classification. This is the case for instance of the reversible imperative language Janus [**?**] and of reversible logic elements [**?**]. This is partially by construction, in the sense that we tried to focus in the taxonomy on the features of the reversibility mechanisms, abstracting away as far as possible from the features of the underlying model.

We describe below each dimension, by explaining the different positions, with examples of approaches that fit each of them. We refer to Section 3 for a more comprehensive description about where models from the literature fit in our taxonomy.

Reversibility focus (FOC): This is the main dimension in the proposed taxonomy. It refers to which aspects of a model are looked at to check whether it is reversible. It features three positions, listed below.

- **Functional behavior (FUN):** In this case a system is said to be reversible if it computes an injective function. Indeed, injectivity ensures that there is a single input which can result in a given output, hence from the output one can recompute the input. As examples, reversible Turing machines, Janus programs, reversible circuits, quantum circuits, and the biorthogonal automata of [**?**] all define injective functions, hence they fit this position. The functional behavior can be computationally as powerful as reversible Turing machines (r-Turing-complete) [**?**], or subuniversal [**?**] and total (always terminating) such as in reversible primitive recursive programs [**?**] and reversible circuits [**?**]. Reversible circuits compute exactly the bijective Boolean functions, which are a proper subset of the partial injective functions that are computable by reversible Turing machines and r-Turing complete reversible languages, like Janus.
- **Reachable states (STA):** In this case a system is said to be reversible if it can go back to past states. Checkpointing and SVN are real world techniques fitting this position. Some notions of reversibility in Petri nets [**?**], requiring that the initial state is reachable from any state, fit this dimension too. Notably, this class of approaches does not consider how past states are reachable, allowing one to reach them via transitions unrelated to the ones used in the past of the computation. Actually, approaches such as rollback directly restore past states, without taking a step-by-step approach to reach them. Notably, not all past states may be reachable, or they may be reachable only with some approximation.
- **Undoing steps (UND):** In this case a system is said to be reversible if it can undo previous steps. This may require or not using special memory

or history information. Reversible process calculi [**?**, **?**, **?**, **?**, **?**], cellular automata [**?**] and Janus [**?**] fit in this position. Note that Janus fits the **FUN** position too: the position depends on the level of abstraction. If we consider a small step semantics, then Janus fits position **UND**; if we abstract away execution details and just look at the functional behavior, then Janus fits position **FUN**.

Note that, if one is able to undo steps, then by undoing steps one can reach past states. Hence, we have the relation (FOC, UND) $\implies$ (FOC, STA).

Also, if a functional behavior can be defined, by undoing steps one can compute the unique inverse function. Hence, the computed function is injective (keeping into account additional memory if present), and we have the implication (FOC, UND) $\implies$ (FOC, FUN).

Resources for reversibility (RES): This dimension refers to whether a model is directly reversible, or whether additional resources (e.g., memory) are needed to enable backward execution.

None (NON): The model is directly reversible, without needing additional memory. Janus, reversible Turing machines and reversible cellular automata fit here. Janus [**?**] is the standard representative of the class of *clean* (without garbage) reversible programming languages, which all fit this position [**?**]. This class includes imperative [**?**, **?**], functional [**?**, **?**, **?**, **?**], and object-oriented [**?**, **?**, **?**] languages; reversible flowchart languages [**?**] to model the high-level structured languages, as well as low-level machine code for reversible von Neumann machines [**?**, **?**].

We remark that models designed without reversibility in mind (e.g., mainstream programming languages) in most of the cases do not fit this position (quantum circuits are an exception to this observation though). In order for models to fit this position, one normally restricts a general class of models. For example, the reversible Turing machines [**?**] are a forward- and backward-deterministic subset of the Turing machines.

Inside the model (INS): In order to enable reversibility some history information is needed. This information is represented in the same formalism as the original system. This happens, e.g., for some Petri nets [**?**], where additional places and tokens can be used to keep such history information. This is also the case for reversible rewrite systems [**?**], where some additional terms are added to make a function injective, an approach which is similar to the addition of a *complement* in the bidirectionalization of functional programs [**?**, **?**]. Another example is the Reverse C Compiler [**?**] that instruments C programs with statements that trace the computation history. Earlier work that trace at the source level are for Pascal programs [**?**] and for irreversible Turing machines [**?**].

Outside the model (OUT): In order to enable reversibility some history information is needed, but to represent this information the model needs to be extended. This happens normally in process calculi [**?**, **?**, **?**] and when mainstream programming languages are made reversible: RCCS

processes [**?**] are not CCS processes, and reversible Erlang [**?**] is not plain Erlang (since the interpreter is instrumented to additionally store history information).

In reversible event structures, additional relations on events such as precedence or direct causation are used to work out how to reverse events [**?**].

It is easy to note that no history information is a particular case of history information, and history information outside the model can mimic history information inside the model. Thus, (RES, NON) $\implies$ (RES, INS) and (RES, INS) $\implies$ (RES, OUT).

Moreover, the classification in this dimension depends on the definition of the model. Notably, by considering a model together with the additional memory needed to make it reversible, one moves from position OUT to INS, or even to NON if one considers history information as part of the normal runtime information of the system. Notably, a model of category NON is able to run backwards without having first run forwards, while for models in category IN or OUT one first needs to run forward to generate and store history information. However, if one looks at history information as part of the state, then one can imagine running backwards directly, just by providing history information as part of the starting state. In practice, the history is often difficult to construct without running a program because it depends on the operational internals of the program.

To summarize this discussion, the categorization of a model inside this dimension critically relies on a clear definition of which is the basic model and which is the history information. This distinction comes out naturally when a reversible model is obtained by extending a non-reversible one: in this case what is added to the non-reversible model can be considered as history information kept to enable reversibility. This is the case of, e.g., RCCS [**?**], which extends CCS with reversibility by equipping each process with a dedicated memory, and in general of Landauer embedding [**?**].

When reversibility is enabled (WHE): This dimension considers whether reversibility is always enabled or not.

- **Always (ALW):** Reversibility is always enabled, one can take any state and compute backwards. This happens, e.g., in Janus. Process calculi require history information to compute backwards, but we fit them here if they can always go backwards provided that history information is available. The distinction between the Janus case and the process calculi case can be made by looking at dimension RES.
- **Sometimes (SOM):** Reversibility is not always enabled, i.e. there are irreversible steps or other conditions that need to be satisfied for enabling reversibility. Some of the examples include RCCS with irreversible actions [**?**] (and in general models or languages featuring control mechanisms for reversibility [**?**]), robotics [**?**], where some actions (e.g., gluing

objects together and drilling holes) cannot be physically reversed, or hybrid quantum-classical algorithms, where only part of the calculation is executed using a reversible quantum circuit.

In this dimension we have (WHE, ALW) $\implies$ (WHE, SOM). Notably, we stated above that Janus fits position ALW, since one can execute backwards from any state, however Janus also has mechanisms to change the direction of execution, in particular the *uncall* of a function computes its inverse function, which can be seen as a control mechanism to decide when reversibility is enabled. Clean reversible programming languages, including reversible machine code, typically include mechanisms that allow to change the computation direction at run time. However, no such mechanism is available in reversible Turing machines [**?**].

Order of undoing (ORD): This dimension is a sub-dimension of the location (FOC, UND), and refers to which transitions can be reversed at a given point in the execution.

- **Reverse order (REV):** This requires actions to be undone in reverse order of completion. This is the typical notion of reversibility in sequential systems (e.g., reversible Turing machines, Janus), and backtracking in concurrent systems [**?**] is also an example of REV. Notably, REV ensures that at any point in time a single backward action is enabled, hence the model is backward deterministic.
- **Causal order (CAU):** This requires actions to be undone only if their consequences, if any, are undone beforehand. Equivalently, causal dependent actions need to be undone in reverse order, while independent actions can be undone in any order. This approach, born in the area of process calculi, is known as causal-consistent reversibility [**?**, **?**]. This is the typical notion of reversibility in concurrent process calculi and languages (e.g., RCCS [**?**], reversible Erlang [**?**, **?**], ...). It has also been used in reversible event structures [**?**], and reversible Occurrence Nets [**?**]. In this position the notion of backward determinism from position REV is weakened into backward confluence.
- **Out of causal order (OCO):** This position does not prescribe any constraint on when actions can be undone. This has been used, e.g., in biological systems and models for them (in some process calculi [**?**, **?**], some Petri nets [**?**, **?**]) and in modeling distributed antenna selection for massive MIMO [**?**] systems.

We have (ORD, REV) $\implies$ (ORD, CAU) and (ORD, CAU) $\implies$ (ORD, OCO). Some Petri net models [**?**] can be tuned so as to cover all three positions in this dimension.

State reachability (STR): This dimension is a sub-dimension of (FOC, STA), and roughly corresponds to the dimension ORD above. This describes which states can be reached by backward execution.

- **Only past states (PAS):** In this position only past states can be reached. This is typical of sequential models (e.g., Janus) or concurrent models when backtracking is used.

Only past states up-to concurrency (CON): Only states that could have been reached in the past by swapping the order of concurrent actions can be reached. This is the typical behavior of concurrent systems based on the causal-consistent approach, such as concurrent process calculi and languages (e.g., RCCS [**?**], reversible Erlang [**?**]).

States reachable by going forward (FOR): In this case backward execution does not introduce new states, but may allow to reach states in different ways. This happens for instance in Petri nets [**?**], where one would like to avoid introducing new states, but it does not matter whether the states were in the past of the computation or not.

Also states not reachable by going forward (NOT): In this case backward execution allows computation to reach new states. This behavior may happen in the presence of out of causal order reversibility (ORD, OCO), hence typically in biological systems. In Petri nets there is a line of work [**?**] trying to understand whether the specific net falls under location NOT or under location FOR.

For state reachability, we have (STR, PAS) $\Longrightarrow$ (STR, CON), (STR, CON) $\Longrightarrow$ (STR, FOR) and (STR, FOR) $\Longrightarrow$ (STR, NOT). This dimension is clearly related to dimension ORD: if a system can be looked at both from the point of view of undoing actions and from the point of view of reachable states, (ORD, REV) corresponds to (STR, PAS), (ORD, CAU) to (STR, CON), and (ORD, OCO) to either (STR, FOR) or (STR, NOT). It would be interesting to find a position in classification ORD corresponding to (STR, FOR), but it is not clear whether any such position exists.

Preciseness of reversibility (PRE): This dimension refers to whether by going backwards one perfectly undoes the effect of forward moves or not.

Precise (PRC): Going forwards and then backwards exactly restores the original state. This happens in most of the models (e.g., Janus, process calculi). This has been captured in causal-consistency theory by the Loop Lemma [**?**].

With additional information (ADD): When going backwards one keeps some information on the undone computation, e.g., that an unsuccessful try has been performed (to avoid doing the same try again), or that a possible solution of the problem has been found (but one would like to find all the solutions). This approach has been partially explored in the area of reversible process calculi using alternatives [**?**] (which allow one to select a different computation upon rollback) or predictions [**?**] (which are not involved in backward computation, hence keep trace of what happened). It has also been studied in the field of session types [**?**], where branches of a choice are discarded upon rollback, and of reversible contracts [**?**], where different alternatives are explored looking for a compatible behavior with another process.

Approximate (APP): By going forwards and backwards one can reach a state which is close in some sense to the starting one, but not exactly the

same. This happens typically in long-running transactions with compensations [**?**, **?**], where the compensation does an approximate undo, and in robotics [**?**], where perfect reversibility is not possible due to small imprecisions in physical actions. Similarly, in reversible neural networks when inputs are recalculated from outputs (not using precise arithmetic), one only gets inputs equal to the original ones up to some small error [**?**, **?**].

We have (PRE, PRC) $\Longrightarrow$ (PRE, ADD) and (PRE, ADD) $\Longrightarrow$ (PRE, APP).

Another possible dimension concerns control of reversibility, namely whether there is any policy to decide which action to take when more than one (forward or backward) action is enabled. Possible positions include uncontrolled (no such policy), semantic control (policy hardwired in the language definition), internal control (there are specific constructs in the model to specify the policy) and external control (the policy comes from outside the program, e.g., from the user or from another program). This dimension has been discussed in [**?**]. We note that frequently uncontrolled reversibility corresponds to (WHE, ALW) while forms of control correspond to (WHE, SOM), since the policy may disallow backward actions under some conditions.

3 Application of the Taxonomy

While in the previous section we discussed the different dimensions of the taxonomy, here we focus on which approach fits which position in the taxonomy. While there is a partial overlap with the examples given in the previous section, this dual view provides interesting insights as well. The results of this section are captured in Table 1.

Research on reversible computing first tackled sequential models of computation, such as finite state automata and Turing machines. The basic idea was to take the original models and restrict to those instances which were reversible. This naturally led to approaches focused on undoing actions at the small step level, computing injective functions at the global level. Actions were undone in reverse order, as natural for sequential systems, leading back to past states in a precise way. This is the case, e.g., of the language Janus and the biorthogonal automata of [**?**]. In turn, some sequential models were extended in order to become reversible by introducing a so-called Landauer embedding [**?**]. Here, we find, e.g., reversible rewrite systems [**?**] and the *bidirectionalization* of functional programs in [**?**].

Such an approach was less suitable for concurrent systems, where reverse order of undoing was too strict in many cases, and one would like to be able to undo independent actions in any order, while undoing dependent actions in reverse order. This was first argued in [**?**], which introduced the notion of causal-consistent reversibility. Instead of restricting calculi to their injective part, memories were added to keep track of past execution (thus fitting position (RES, OUT)), and enable backward computation. Given that in concurrency functional behavior

Formalism	Approach	FOCus	RESource	WHEn	ORDer	STate R.	PREcis.
Reversible Turing machines	[**?**, **?**]	**FUN UND**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Janus	[**?**]	**FUN UND**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Biorthogonal automata	[**?**]	**FUN UND**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Reversible cellular automata	[**?**, **?**]	**FUN UND**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Reversible logic elements	[**?**, **?**]	**FUN UND**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Reversible rewriting	[**?**]	**UND**	**INS OUT**	**ALW**	**REV**	**PAS**	**PRC**
Causal-consistent calculi	[**?**, **?**] [**?**, **?**] [**?**, **?**]	**UND**	**OUT**	**ALW**	**CAU**	**CON**	**PRC**
Calculi + control	[**?**, **?**]	**UND**	**OUT**	**SOM**	**CAU**	**CON**	**PRC**
Calculi + predictions	[**?**]	**UND**	**OUT**	**ALW**	**CAU**	**NOT**	**ADD**
Reversible Erlang	[**?**, **?**]	**UND**	**OUT**	**ALW SOM**	**CAU**	**CON**	**PRC**
Petri nets	[**?**]	**STA**	**NON**	**ALW**	**OCO**	**FOR NOT**	**PRC**
Reversing Petri nets	[**?**, **?**]	**UND**	**INS**	**ALW**	**CAU OCO**	**CON FOR NOT**	**PRC**
Occurrence nets	[**?**]	**UND**	**NON**	**ALW**	**CAU**	**CON**	**PRC**
Petri nets	[**?**]	**UND**	**INS**	**ALW**	**CAU**	**CON**	**PRC**
Biological models	[**?**, **?**]	**UND**	**OUT**	**SOM**	**OCO**	**NOT**	**PRC**
Event structures	[**?**, **?**]	**UND**	**OUT**	**SOM**	**CAU OCO**	**CON FOR NOT**	**PRC**
Quantum circuits	[**?**, **?**]	**FUN UND**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Quantum programming languages	[**?**, **?**]	**UND**	**INS**	**SOM**	**REV**	**PAS**	**PRC**
Reversible neural networks	[**?**]	**UND**	**NON OUT**	**SOM**	**REV**	**NOT**	**APP**
Reversible Markov chains	[**?**]	**STA**	**NON**	**ALW**	**REV**	**PAS**	**PRC**
Sagas	[**?**]	**UND**	**INT**	**SOM**	**CAU**	X	**APP**
SVN	[**?**]	**STA**	**INT**	**ALW**	X	**PAS**	**PRC**

Table 1. Application of the taxonomy to sample approaches from the literature

is of limited interest, since interaction with the environment is important, the focus is mainly on undoing actions. A similar approach has been applied to programming languages for concurrency, in particular Erlang [**?**, **?**], where causal

consistency is ensured for both forward (replay) and backward computations during debugging [**?**, **?**].

While the first approaches considered precise reversibility which was always enabled, further studies introduced control mechanisms [**?**, **?**] as well as forms of reversibility which were not precise [**?**]. Some applications, most notably in the biochemical setting, triggered the need for weakening causal order, thus introducing out of causal order reversibility [**?**, **?**]. We note that CCSK [**?**], with the addition of a control mechanism in the form of a rollback operator inspired by [**?**], has been modeled using reversible event structures exploiting out of causal order reversibility [**?**].

Petri nets, while being a model for concurrency like process calculi, resulted in a number of different approaches. The fact that Petri nets have a clear representation of state (in terms of tokens inside places), triggered approaches [**?**] focusing on state reachability more than on action undoing. Approaches based on action undoing were also considered and contrasted with the ones based on state reachability [**?**]. Other works [**?**] considered the causal-consistent approach, thus matching the one of process calculi. Further work tailored Petri nets for biological applications [**?**], allowing one to explore different forms of reversibility, most notably the out of causal order one.

In the quantum circuit model [**?**, **?**] used for developing most quantum mechanical algorithms, the set of allowed operations is represented by unitary matrices or unitary gates. Such matrices act on an isolated physical system and, in this scenario, one is always able to undo the last action. Hence, the term reversible is, in quantum computer science, synonymous with the term 'unitary'. Compared with classical reversible gates, unitary matrices provide us with a larger set of operations. However, to read out the result of the computation, one needs to translate the final state into the classical result. Such a process requires a measurement which is achieved through interaction with the system executing the computation. The main feature of such a process is its irreversibility. Thus, reversibility is lost at the moment of 'interfacing' with a classical machine or with the readout procedure. Architecture-specific limitations of current quantum hardware lead to the problem of optimizing quantum circuits [**?**], most importantly taking into account the hardware topology [**?**]. Such optimization is part of the process of transpilation – translation of quantum circuits to the form suitable for the target quantum computer.

This need of interfacing between the reversible and the irreversible elements motivated the development of quantum programming languages [**?**, **?**, **?**]. Also, many quantum algorithms (NISQ algorithms in particular) use classical subroutines. Quantum programming languages include a specialized type system for handling quantum structures used in purely quantum, reversible computation. Additionally, they also include an irreversible subsystem, suitable for dealing with classical – which in this case means irreversible – computation.

Reversibility is used in *Convolutional Neural Networks* [**?**] (CNNs) to undo computation of the networks' layers. This removes the need to store, retrieve and delete layers' inputs and outputs, which can be recomputed instead. Some

layers perform transformations (of inputs to outputs) which have inverses, such as multiplication by a matrix, so are directly reversible. Other transformations, such as applying a convolution or max pooling, lose data so can only be reversed by enriching the network with additional components. A *Reversible Residual Network* [**?**] (ResNet) is a form of CNN that adds *shortcuts* between layers. This makes it possible to undo computation of most layers. Calculation is not in precise arithmetic, so only approximate values of inputs can be uncomputed from outputs (up to an agreed precision), and thus new states can be reached.

In the field of performance evaluation, a Markov chain is (time) reversible [**?**] if it has the same behavior as its inverse, in terms of probabilistic distribution. Hence the focus is on states, and, since the approach restricts attention to Markov chains which naturally satisfy the reversibility property, no additional resources are required. Reversible transitions are always enabled, though they are subject to a probabilistic distribution, and the order of reversing is the inverse of the forward one. An initial work relating causal-consistent reversibility with reversible Markov chains in the setting of a stochastic process algebra is described in [**?**].

We conclude the table with a few approaches which are at the boundary of reversible computation, namely Sagas [**?**], used to model long-running transactions with compensation, and the well-known tool SVN for version control [**?**, **?**]. Given the distance from classical reversible computation, it is not clear whether some of the dimensions make sense in these cases. We put 'X' in the cells which we believe are not interesting.

4 Conclusion, Related and Future Work

We have presented a first proposal of taxonomy for reversible computation approaches, and discussed how various models fit in it. We focused on approaches from programming languages and concurrency theory, hence in future work it would be good to put our taxonomy at work also on other kinds of models.

We are not aware of other works putting forward proposals of taxonomies for reversible computing. A partial analysis in this direction is the classification of control mechanisms in [**?**] and an account of reversible computing from a programming language perspective [**?**]. Also, a few works in the context of Petri nets contrast different approaches [**?**, **?**], taking advantage of the existence of many such approaches.

Table 1, while not covering all the literature, highlights some holes which are interesting targets for future work. For instance, a large part of the approaches concern precise reversibility, and indeed this is the main focus of the reversible computing community. Approaches where reversibility is not perfect are however of interest as well, motivated, e.g., by applications in robotics and neural networks, and are an interesting research direction for the reversible computation community. Another interesting point is that most of the approaches focus on undoing actions, while a focus on functional behavior and on states has been adopted only in a few cases. From a theoretical perspective, it would also be

interesting to investigate the computational power and inherent complexity of reversible computing models.

Acknowledgements This work refines and extends the results of discussions that occurred during the meetings of the COST Action IC1405 on Reversible Computation – Extending Horizons of Computing. We thank all the participants to such discussions. The authors were partially supported by the COST Action IC1405. We thank the anonymous referees for their helpful comments.