COST Action IC1405
Reversible computation
extending horizons of computing

State of the art report
Working Group 2
Software and Systems

Editors:
Claudio Antares Mezzina and Rudolf Schlatte

Contributors:
Manuel Alcino Cunha, Vasileios Koutavas,
Ivan Lanese, Claudio Antares Mezzina,
Jarosław Adam Miszczak, Rudolf Schlatte,
Ulrik Pagh Schultz, Harun Siljak,
Michael Kirkedal Thomsen, German Vidal

March 17, 2017

Contents

1 Notions of Reversibility **4**

2 Fully reversible languages: **4**
2.1 Sequential Imperative languages 4
2.2 Declarative languages 6
2.3 Compiler technology, program transformation 7
2.4 Concurrent language 7

3 Dependable System Abstractions **8**
3.1 Language support for transactions: 9
3.1.1 Database Transactions 10
3.1.2 Software Transactional Memory 10
3.1.3 Non-Isolated Transactions 11
3.2 Checkpoint and Rollback 12
3.3 Library support for limited reversibility 13
3.4 Debugging & Program Slicing 14

4 Execution replay **15**

5 Quantum Programming **16**
5.1 Imperative paradigm 16
5.2 Functional paradigm 20
5.3 Support in general purpose languages 23

6 Bidirectional Transformation **24**

7 Robotics **27**
7.1 Modular self-reconfigurable robots 27
7.2 Industrial robots 29

8 Control Theory **30**

Preface

The intent of this report is to enlist all the different guises of reversibility that can be found in software and systems. The work was initiated by Working Group 2 of the COST Action IC 1405 on reversible computations, as a continuation of the work done by the Working Group 1 on foundations of reversibility. The contents have been provided by the leading experts of the different research fields that are covered: see the Contents or each section for more details.

More on the project:

- `http://www.cost.eu/COST_Actions/ict/IC1405`
- `http://www.revcomp.eu/`

1 Notions of Reversibility

Thinking about reversibility intuitively leads to think also about undoing. As hinted by Bennett [12], any forward computation (or execution) can be transformed into a reversible one by just keeping an history of all the information overwritten and hence lost (for example a variable update) by the forward computation, and then use this information to reverse (or undo) the forward computation. The ability for a system to get back to an exact past state by deleting all its effects is what we call *full reversibility*. As we will see, the definition of full reversibility varies when moving from a sequential setting to the concurrent (possibly distributed) one.

In a distributed setting achieving full reversibility may be cumbersome if not impossible due to the fact that some actions are by definition irreversible (e.g. printing out a file). Several techniques used to build dependable systems such as transactions [51], system-recovery schemes [33] and checkpoint-rollback protocols [75], rely on some forms of undo or rollback facility.

Reversible computation is also at the core of newly-proposed programming paradigms for developing control systems and robots with a high level of autonomy and adaptation.

Reversibility is also related to *bidirectionality*, which implies a rich literature (see [26]). For the purpose and scope of this document we will just focus on two broad classes of work: reversible computing, where the main focuses are to provide a reversible computing model and to see what is the expressive power or the computational cost of reversibility; and dependable systems abstractions where dependable systems are obtained by means of some forms of reversibility.

Reversibility is also related with quantum computing, since quantum programs are always logically reversible.

The rest of the report is structured as follows: Section 2 describes fully reversible programming language both in the sequential and in the concurrent setting. Section 3 deals with techniques used to program dependable systems (e.g transactions, checkpoint and rollback techniques) and to debug them. Section 4 treats with replay techniques, which can be used for several aims such as program debugging, security and simulation. Section 5 is about quantum programming, while Section 6 is about bidirectional transformation. Section 7 is dedicated to the robotics domain, while Section 8 to control theory.

2 Fully reversible languages:

2.1 Sequential Imperative languages

Reversible programming languages can be dated back to 1986. At this time, Lutz, after a brief meeting with Landauer, sent him a letter about some work he did with Derby, about four years earlier, on a reversible imperative language called Janus [87]. Their work arose from an interest to investigate if it was possible to implement such a language and before 1986 Lutz and Derby did not know about Landauer's principle. The language was 'rediscovered' after the turn of the century and has since then been formalized and further developed at DIKU [144, 146, 147]. Other simple reversible imperative languages have been developed, e.g. Frank developed R [39] to generate instruction code for

the Pendulum processor and Matos [89] made a language for linear programs.

Listing 1: Janus code describing Fibonacci

```
procedure fib(int x1,int x2,int n)
   if n=0
      then x1 += 1
         x2 += 1
      else n -= 1
         call fib(x1,x2,n)
      x1 += x2
      x1 <=> x2
   fi x1=x2

procedure fib_fwd(int x1,int x2,int n)
   n += 4
   call fib(x1,x2,n) // forward execution

procedure fib_bwd(int x1,int x2,int n)
   x1 += 5
   x2 += 8
   uncall fib(x1,x2,n) // backward execution
```

Listing 1, taken from [143], shows a Janus procedure for computing *Fibonacci pairs*. Given an integer n, the procedure fib computes the $(n+1)$-th and $(n+2)$-th Fibonacci number. For example, the Fibonacci pair for $n = 4$ is $(5, 8)$. Returning a pair of Fibonacci numbers makes the otherwise non-injective Fibonacci function injective. Variables n, $x1$, $x2$ are initially set to zero. Parameter passing is pass-by-reference. One key point of Janus is the use of *reversible updates* $+=$ and $-=$ and conditional with with *entry* and *exit* guards. In the above code the operator $<=>$ is used to swap the value of two variables without resorting to a third one.

A general framework for adding *undo* capability to a sequential programming language is presented in [82]. The paper also present a rich survey about the undo operation and all its manifestations in different fields: such as text editors, programming languages, function inversion, backtracking and physics. The paper identifies two interpretations of the undo operation and motivates them by examples. Such interpretations are: $\texttt{undo}_\texttt{b}$ and $\texttt{undo}_\texttt{f}$. Let us show the basic ideas behind them. Consider that we are editing an empty file by issuing the following four command lines: insert w, insert x, insert y and finally insert z. Then, the history of the effects on the file is the following sequence:

$$\langle \Omega, w, wx, wxy, wxyz \rangle \tag{1}$$

where Ω represents the empty file, and $wxyz$ represents the state of the file containing four lines and four characters. Now, let us introduce a time parameter t such that in each unit of time one edit command is executed. So, if we apply the function $\texttt{undo}_\texttt{b}(1)$ to 1 we obtain the following history:

$$\langle \Omega, w, wx, wxy \rangle \tag{2}$$

if we continue by applying $\texttt{undo}_\texttt{b}(3)$ to 2 we will obtain Ω. That is $\texttt{undo}_\texttt{b}(t)$ destroys the last t actions in the file history. On the other hand, by allowing

the undo operation to move forward in the history a second interpretation is possible. For example, if we apply $\texttt{undo}_\texttt{f}(1)$ to 1 we obtain the following history:

$$\langle \Omega, w, wx, wxy, wxyz, wxy \rangle \tag{3}$$

and by applying $\texttt{undo}_\texttt{f}(3)$ to 3 we obtain the following history:

$$\langle \Omega, w, wx, wxy, wxyz, wxy, wx \rangle \tag{4}$$

So the $\texttt{undo}_\texttt{f}(t)$ function moves forward by just copying the state of t states before. The main differences between the two operations are: (i) $\texttt{undo}_\texttt{b}$ destroys history while $\texttt{undo}_\texttt{f}$ just moves forward preserving all the previous states; (ii) $\texttt{undo}_\texttt{b}$ moves the computation history to a point we already have seen before while $\texttt{undo}_\texttt{f}$ always creates a new history which never existed in the past. To bring these intuitions into programming languages, a notion of *undo-list* is introduced. An undo-list is composed by triplets of the form $\langle a, v, t \rangle$ where a is a variable name, v its value and t the instant in which the value has been assigned to the variable. Hence, the undo-list gives a means for restoring previous values to variables. A computational history is then made of states, each one represented by an undo-list. Two primitives mimicking the semantics of $\texttt{undo}_\texttt{b}$ and $\texttt{undo}_\texttt{f}$ are given along with two primitives for undo-list manipulation: one able to increase the instant t of current state variables and the second one able to copy the value of an undo-list into another.

2.2 Declarative languages

Though the first reversible programming language was imperative, reversible functional languages have lately received the most interest. This development started in 1996 when Huelsbergen presented SECD-H [64], an evaluator for the lambda calculus that extended Landin's SECD evaluator [77] with a history tape. (Kluge [71] similarly extended a machine that can reduce a program term to normal-form and back again.) This was followed by Mu et al. who, with applications in bidirectional computing in mind, presented a reversible relational language [101]. This research has been further developed by Matsuda et al. [90, 92], where a reversibilization technique (including two stages: injectivization and inversion) for a simple functional programming language is presented. The authors mainly follow a *Landauer's embedding*, although a number of analysis are introduced in order to reduce the added information as much as possible. More recently, work towards general purpose functional programming languages was presented independently by Yokoyama, Axelsen, Glück [145] (later extended [133]) and James, Sabry [67]. Another recent approach has been introduced by Nishida et al. [104], where reversibility in the context of *term rewriting* is considered. Term rewriting captures the essence of (first-order) functional computations. In [104], the authors first introduce a *reversible* (but conservative) extension of term rewriting (a Landauer's embedding). Then, a transformation for rewrite systems is also introduced, so that the transformed systems behave with standard rewriting as the original ones with the reversible extension (i.e., reversibility is compiled into the rewrite rules, which opens the door to several applications). Other related approaches include Abramsky's [4] approach to reversible computation with *pattern matching automata*, which could also be represented in terms of standard notions of

term rewriting. His approach requires a condition called *biorthogonality* (which, in particular, implies injectivity), so that the considered automata are reversible without adding any additional information. Abramsky's work can also be seen as a rather fundamental delineation of the boundary between reversible and irreversible computation in logical terms. Finally, let us mention other works on reversibility for closely related languages, e.g., a probabilistic guarded command language [151], a low level virtual machine [130], or combinatory logic [30], to name a few.

2.3 Compiler technology, program transformation

On a related topic, transformation of reversible languages have also received some interest lately (mainly at DIKU). Though the first compiler between two reversible language was made by Frank [39] (between his language R and a reversible instruction set called PISA), it was Axelsen who devised techniques for a compiler that could perform clean translation [9]. His translation was between Janus and PISA and was clean in the sense that the compiled program did not have more than a constant memory overhead over the original Janus program. It is likely that the PISA program will use more temporary memory/registers. Lately we have seen more elaborated schemes representation of heap structure and garbage collection [10, 99, 100].

Sometimes it is useful to have portions of code of a non-reversible program to be reversible. This is the case of parallel discrete event simulation (PDES) [40], where speculative simulation is used to boost the speed of the simulator. Naturally, if the speculation was made on a wrong assumption then all the computations caused by it have to be reverted. Works [117, 118] focus on automatic generation of C++ reverse code for parallel discrete simulations.

2.4 Concurrent language

The notions of reversibility used in the sequential setting are not suitable for concurrent and/or distributed systems. Indeed, in concurrent/distributed settings different actions can overlap their execution, hence the notion of "last action" is not well defined. In some distributed systems there may also not be a unique notion of time. This led to the proposal of *causal-consistent* reversibility [28, 78], which states that any action can be undone provided that its consequences, if any, are undone beforehand. Notably, this avoids any reference to the notion of time, while using causality to relate actions.

Up to now the only proposal of fully reversible concurrent language we are aware of is the causal-consistent reversible extension of μOz [85]. The μOz language is a kernel language of Oz [138]. μOz is a higher-order language with thread-based concurrency and asynchronous communication via ports (i.e., channels). The semantics of μOz is defined using a rather standard stack-based abstract machine. In order to make it reversible the abstract machine is extended with history information associated to each thread and each queue. This allows one to reverse any μOz statement, but does not provide control on when to reverse those statements. A form of control is provided in [44, 25] which provides a causal-consistent reversible debugger for μOz.

Listing 2 describes Fibonacci function in μOz. Differently from Listing 1, in μOz there is no need to resort to Fibonacci pairs to make the function bijective.

The procedure takes in input a number n and a channel res and sends on res the n-th Fibonacci number. Executing the code in the μOz interpreter [25] allows one to move forward and backward along the execution. While in Janus a program can be executed both forward and backward, in μOz one can execute a program backward only after it has been executed forward. Furthermore, in Janus there is no need to use extra memory to store history information since the language is naturally forward and backward deterministic, while in μOz one has to keep extra memory to remember past actions.

Listing 2: μOz code describing Fibonacci

```
let fib = proc {x res}
   if (x<=1) then
   let one = 1 in  {send res one} end
   else
   let z1 = port in
   let z2 = port in
      let w = x-1 in
      let u = w-1 in
         {fib w z1}; //call fib n-1
         {fib u z2}; //call fib n-2
         let fa = {receive z1} in
         let fb = {receive z2} in
            let n = fa + fb in {send res n} end
         end end
      end end
   end end
end end
in
   let ch = port in
   let num = 5 in
      {fib num ch}
   end end
end
```

3 Dependable System Abstractions

A *failure* of a system occurs when its behavior differs from the one that has been specified. The part of the system state leading to the failure is called an *error*. Here an error is always caused by a *fault*, which in many cases are physical (e.g., hardware faults) and inevitable. Hence faults are the cause of errors that lead to failures [135].

Fault tolerance eliminates system failures in the presence of faults, providing high system dependability and the required level of service [8]. The two main techniques for achieving fault tolerance are *fault masking* and *fault treatment*. The former aims at eliminating errors before they occur by reducing the faults leading to an error, employing for example data replication and component redundancy.

Fault treatment, on the other hand, seeks to handle certain system errors after they occur, stopping them from causing a failure. With this technique after

an error occurs the system is brought to a consistent state. If this is a previously saved state of the system, as it happens in transactional and checkpoint recovery, the fault tolerance mechanism is classified as *backward error recovery*. If the recovery involves correcting the error without resorting to a previous state, as in exception handling (e.g. [65]), then the mechanism is called *forward error recovery*. The latter can be made more efficient in resources, whereas the former can be more easily automated requiring little programmer intervention.

In this section we examine mechanisms for fault treatment, and in particular those employing backward recovery, as they are an instance of reversible computation in which errors trigger reversing actions. In this framework, the building blocks of fault treatment mechanisms can be seen as *defeasible* partial agreements in a distributed setting. Reversibility provides us with a high-level setting for examining and ultimately specifying, and verifying fault tolerant systems with backward recovery.

3.1 Language support for transactions:

According to [49], a system implementing transactions

> provides *operations* each of which manipulates one or more entities. The execution of an operation on an entity is called an *action*. [...] Associated with a database is a predicate on entities called the *consistency constraint*. A database state satisfying the consistency constraint is said to be *consistent*.

Transactions are used to implement the ACID properties [50] of traditional database systems.[1] Pertaining to the topic of this report, offering the consistency and isolation properties of transaction semantics requires an implementation of (limited) reversability.

Conceptually, transactions wrap a sequence of operations inside an atomic block such that either all or none of the actions occur. A transaction is completed by either committing or aborting. Aborting reverses the effects of all operations within the transaction, leaving the state coherent.

Systems offering transaction semantics differ in the following aspects:

- The entitites that are being protected by transactions. In software transactional memory, the entities are a number of shared variables; in database transactions it is the state of the database.
- The operations on these entities that can be committed or aborted, usually read/write but also create/delete.
- The semantics of aborting a transaction. After a transaction is aborted, there might be observable changes in the protected entities since other, concurrent transactions might have committed successfully at the same time.
- Characteristics of the transactions themselves, e.g., whether transactions can be long-lived or abort after some fixed time, or whether transactions can be nested.

[1] Atomicity, Consistency, Isolation, Durability – note that the Isolation property is not explicitly listed in the introduction of [50], although very much discussed in the body.

- Implementation aspects like optimistic versus pessimistic synchronization and commutativity of operations; these mostly influence performance of the system.

In the classical transactional model [14, 110] transactions are seen as sequences of read and write operations that map consistent database states to consistent states when executed in isolation. A concurrent execution of a set of transactions is represented as an interleaved sequence of read and write operations, and it is said to be serializable if it is equivalent to a serial (non-concurrent) execution. A transaction is a sequence of actions that have to be executed atomically: either it successfully completes (*commits*) and all its effect are visible to the other transactions; otherwise it *fails* and its effects are not visible.

Transactions, originally introduced in the field of DBMS (Data-Base Management Systems) models, provide good concurrency abstraction models in programming languages, since they ensure nice properties, such as atomicity and isolation, di cult to obtain if manually programmed. Indeed, if a developer were to ensure such properties, he would have to design the program relying on low level concurrent programming primitives (e.g. critical sections, semaphores, monitors), which is typically a difficult task, and even more, hard to debug. In this section we will review works dealing with transactional programming languages guaranteeing *atomicity*, *consistency*, *isolation* and *durability* (or ACID) properties. Works modelling non-classical notions of transactions (such as long-running transactions, open-nested transactions) will be reviewed in Sections 3.1.2 and 3.1.3.

3.1.1 Database Transactions

Transactions for database systems, and the associated rollback reversibility techniques, are usually implemented as SQL statements embedded into a host programming language via libraries such as JDBC. The state of the database is protected by a transaction; the SQL SELECT, UPDATE, INSERT and DELETE statements constitute the operations. A transaction can be aborted either by the client or by the database system. After an aborted transaction, the database state can be different than before the transaction was started, i.e., the reversion of the operations within the aborted transaction leads back to a consistent state, but not necessarily to a state identical to the starting state.

3.1.2 Software Transactional Memory

Software Transactional Memory (STM) is "a concurrency control mechanism for executing accesses to memory shared by multiple processes. A transaction, in this context, is a section of code that executes a series of reads and writes to the shared memory as one atomic indivisible unit." [125] Implementations of STM typically have lower overhead and are less error-prone than protecting shared memory via explicit locking. STM is used to synchronize threads running within the same process, or processes running on the same machine.

STM implementations in various languages offer different functionality. For example, in the Haskell implementation [88, Chapter 10], [58] the type system guarantees that its transactional variables are only accessed inside a transaction, while the Clojure implementation [56, Chapter 5] offers commuting operations

and validation functions, reducing the likelihood of rollbacks and offering additional consistency checks beyond the type system of the language, but does not statically guarantee that the transaction is side effect-free. Early research in STM was carried out largely in functional languages, other examples include SML/NJ: [54]; AtomCaml: [114]; and Scheme [69]. Implementations in imperative languages like Java [73] and C++ suffer from poor interoperability with code not aware of STM, relying on load-time instrumentation and similar techniques.

Again, STM implements a specific form of reversibility that rolls back a sequence of operations over shared state back to a consistent state.

3.1.3 Non-Isolated Transactions

Isolation in traditional ACID transactions is crucial for programming mutual exclusion. If two transactions access the same location of the system state, and one of them writes to this location, then the system aborts at least one of the transactions. However, a number of works [37, 86, 29, 32, 83, 127] have proposed dropping the Isolation principle to obtain a useful abstraction for concurrent programming. Although non-isolated transactions cannot encode mutual exclusion, they can be used to maintain a consistent distributed state [37], multi-party synchronisation without the use of a coordinator [32], and in general they provide a method to encode distributed consensus problems [128].

In [54] primitives to build composable transaction abstractions in ML are given. Transactions are factored into four separable features: *persistence*, *undoability*, *locking* and *threads* and then each composition of these properties gives rise to different transactional models. Following the idea of adding transactions to Objective Caml, the language AtomCaml [114] has been proposed. The language is endowed with the new function `atom`(f) of type (`unit` $\rightarrow$ `'a`) $\rightarrow$ `'a` whose purpose is to execute atomically the function `f`. The basic idea behind this function (and primitive) is that it tries to execute sequentially (hence not interleaved) the entire function block, if during the execution the thread executing it has been pre-empted by the scheduler then the function rolls-back and re-executes again. This implies that the function is executing in a monoprocessor setting, where true concurrency does not exist. Moreover there are a few limitations about the side effects that the atomic block can have: input operations are not allowed (since there is no way to reverse them); output operations are always buffered and *flushed* only if the block successfully completes, and exceptions that escape from the atomic block boundaries force the atomic block to complete. Changes made by an atomic block to mutable variables are logged, that is variables are like stacks of values, and in case of failure these changes are reverted.

Different works have approached the design of transactional languages from a semantic point of view, formally proving some properties. Transactional Featherweight Java [66] is an object calculus with support for nested and multi-threaded transactions. As usual a transaction is delimited by a special block, in this case the `onacid` statement. Each time an `onacid` is executed a new transaction (identified by a label) is created, and all the threads executing in it are bound with the transaction label. Each thread is executed into a transactional environments that keeps track of all the read and write operations that the thread performs on objects. Then by varying the semantics of operations on

transactional environment two kinds of semantics are given: *versioning semantics* and *strict two phase locking semantics*. The first one mimics the STM (we will discuss about STM later on) logging mechanism, since when a thread enters a new transaction, an empty log corresponding to the transaction is created. This log keeps track of all the objects that are modified within a transaction, for a write operation for example the old value is written in the log. A transaction successfully commits if its log is consistent with the father's one, otherwise it fails. A log is consistent with the father ones, if all the values of the objects read by the child thread have remained unchanged until the committing time. In the strict two phase locking semantics there is no more need of a log mechanism, since before modifying an object, a transaction has to require a lock on it. All the collected locks will be released when the transaction commits. Nested transactions inherit father locks. By using the lock mechanism, there is no way for a transaction to fail.

3.2 Checkpoint and Rollback

In distributed systems, checkpointing and roll-back (also known as checkpoint-recovery) is a technique of backward recovery (see [8]) for creating fault tolerant systems. The key concepts of this technique are: (i) periodic saves of system global state; (ii) in event of a fault, the state is restored via a rollback. This particular technique gives to a system (or to an application) the ability to save its state and tolerate faults by simply restoring an earlier state. In fact, when a checkpoint is executed, a snapshot of the entire system is taken and normally it is saved into some non-volatile medium. If a fault is detected, the recovery mechanism restores the system to the last checkpointed state. There is an abundant literature (see [8, 33] for a quick review) on protocols and techniques on how to build a global (distributed) checkpoint, on what kind of information should represent the global state, and so on. We will address this topic by a programming language point of view, and then not considering libraries, middleware and operating system services.

In [19] a reversible extension of the Scala language supporting channel-based synchronisation is presented. Two are the primitives added to Scala in order to enact reversibility (and rollback): `stable e` and `backtrack e` to menage backtracking over speculative executions. The first one delimits the scope of the backtrack events within e, while the second brings back the process to the dynamically closest nested block with the value of e. Backtracking has also the effect of deleting all the communications the process has done. A backtrack action might force neighbouring processes to also backtrack, possibly resulting in a cascade of backtracking (*domino effect*) for a poorly written program. The backtrack mechanism is shown to be implementable via a point to point algorithm avoiding thus a centralised controller.

Bringing checkpoint-recovery technique in the actor model has been tackled by Transactor [37]: a fault tolerant programming model for composing loosely-coupled distributed components. It extends the actor model with constructs which distributed processes can use to to build globally consistent checkpoints. Basically a transactor can decide to commit its current state to a stable one. When a transactor becomes stable, further communications cannot change its state. It can be seen as a promise, to all the other transactors who communicated with it, that its state will not change. When an unstable transactor

decides to roll-back, or it is asked to do so, it will cause the rollback of all the transactor whose state depended on the state of the unstable transactor. Interestingly the semantics also consider message loss. The language is proved to be sound, that is a trace containing node failures is equivalent to a normal one not containing failures, but possibly message losses. Moreover, checkpointing is possible just under certain conditions, and not in general cases. Indeed, not all the transactor programs can reach global checkpoints. A trivial program with a transactor that sends messages introducing dependencies, but never stabilizes or tries to checkpoint, will eliminate the ability of its dependent parties to reach checkpoints. Hence the authors introduces the *Universal Checkpoint Protocol* (UPC) that assumes a set of preconditions that will entail global checkpointing for a set T of transactors.

Transient faults are unusual conditions that can be remedied by just re-executing the code which raised it. These faults are usually generated by temporally unavailable resources. For example, if a server is rebooting due to an internal error, then all the client requests issued during the rebooting time should be re-executed. In [150] a concurrent ML language, called stabilizers, for transient concurrent fault recovery in concurrent program is presented. The language introduces three new primitives: `stable`, `stabilize` and `cut`, able to deal with program global checkpoints. Primitive stable allows a thread to create a new stable section, that is a new global checkpoint. Primitive stabilize, issued by a single thread, allows the entire program to roll-back to the previous global checkpoint and finally primitive cut discards the current global checkpoint. This ensures that subsequent calls to stabilize will never cause the program to get back to a state that existed logically before the cut. Even if the semantics is proved to be safe, that is stabilization actions can never manufacture new states, the semantics cannot avoid the domino effect, that is a stabilize operation may unduly revert the program to a state beyond the target checkpoint.

3.3 Library support for limited reversibility

The Command Pattern [41] is a design pattern that describes how to execute, undo and redo operations that are initiated by a user. To support the Undo functionality, which reverses the effect of an operation, a Command objects following that pattern carries information on whether it can be reversed or not (e.g., usually saving a document is not reversible), and instructions on how to undo its own execution. Systems built following this pattern can undo and redo commands issued. Many higher-level GUI toolkits offer predefined interfaces or base classes to implement the Command pattern and get such limited reversibility (Undo/Redo) with small implementation effort. Reversible [113] is a library for python whose aim is to provide a simple abstraction for actions that can be reversed or rolled back and provides methods to construct, chain, and consume them in a readable way.

Reversing the state of a system might mean going back to an earlier version of its state. The cost of storing previous versions of large data structures can be prohibitive. Persistent (immutable) data structures [105] offer more efficient storing of multiple versions of a data structure, sharing structure where possible. This makes it possible to simply store previous state information without undue memory pressure. While persistent data types can be implemented in many languages, their usefulness is limited if existing third-party or library code

expects a language's existing, non-persistent data structures. The Clojure language [57] uses persistent data structures throughout and provides support for calling Java libraries; many purely functional languages also have persistency built-in.

3.4 Debugging & Program Slicing

Reversible debugging has been known for the last 40 years [53, 148] and gets all its interest and motivation from assisting the programmer in the search of possible bugs by exploring the computation both forward and backward. Retracing back the steps is very useful when investigating a misbehavior. In a sequential setting it is also very natural: steps are simply undone in the reverse order of execution. Reversibility for debugging of sequential programs has been quite extensively explored [21, 35, 72, 84], and some reversible debugging features are available also in mainstream debuggers. For instance, GDB supports reversible debugging since version 7.0 (released in 2009). A main limitation of sequential reversible debuggers is the huge overhead required to store history information, both in terms of time and in terms of space, which makes difficult the use of reversible debuggers for large programs. State of the art techniques based on incremental storage of information and complex fine tuning allow a consistent reduction of the overhead making the technique applicable at the industrial level [136].

The interplay between reversibility and concurrency makes things more complex: concurrent reversible debugging is a less explored world. All the approaches to concurrent reversible debugging we are aware of fall in the two categories below

Non-deterministic replay debugging [7, 139, 136] : in order to go back to a previous step, the execution is replayed non-deterministically from the start (or from a previous checkpoint) until that step.

Deterministic replay/reverse-execute debugging [70, 24] : a log is kept while executing, and when going back thread activities are either undone in the exact reverse order they were executed, or the execution is replayed from a previous checkpoint following the particular interleaving in the log.

Both approaches present drawbacks. In the first case, actions could get scheduled in a different order at every replay, and the error may not get reproduced. So in this case, error proper to concurrency such as race conditions could not get caught. Even if it does, one may not get any insight on the causes of the error. Following the second approach, if the error was due to one among a million of independent parallel threads, and that one was the first one to execute, one needs to undo all the program execution before finding the bug. Even more, one does not understand which threads are related to a given misbehavior, since there is no information on the relations among them. In [44] a novel approach to concurrent reversible debugging exploiting causality information is presented, were the notion of causally consistent reversibility is brought into a reversible debugger.

Causality in the context of non-reversible concurrent debugging has been addressed in different works [81, 149, 142] , which mainly rely on the Lamport's happens-before causality relation [76]. In all these works causality is used to

support determinism in replaying techniques and to define efficient dynamic slicing. In contrast, the use causality as a support for rollback primitives allowing the programmer to find the causes of a misbehavior has been just explored are Causeway [20] and CaReDeb[25]. Causeway is not a full-fledged debugger, but just a post-mortem traces analyzer. It exploits a causality notion, based on the Lamport's happens-before, more liberal than causal consistency exploited by CaReDeb. CaReDeb is an interpreter and debugger for μOz [85] whose key idea is to use rollback primitives to enact reversible debugging [44]. Rollback is done in a causally consistent fashion, that is in order to revert an action all its causes have to be undone first.

Simple reversibility features exist in programming languages for commercial, industrial robots from major manufacturers such as ABB, Fanuc and KUKA, and have been shown to have significant practical value [79]. In these languages reversibility is used when programming and debugging the robots [102]. Manufacturers however use various approaches and implementations. KUKA offers different options, one of which is backtracking: execution is recorded during forward motion and can be backtracked afterwards [74]. Fanuc uses a simple implementation of program inversion based on backward interpretation [34]. This allows the user to step through programs in reverse order, but only works on the move commands and cannot reverse control flow structures. RAPID adds to this approach, and allows users to specify alternative reverse instructions for backward execution of a subroutine [3]. Overall these implementations are useful for interactively reversing the movements of the robot. They are however not suited for performing conceptually reverse executions of entire tasks.

4 Execution replay

Execution replay is a technique which finds application in several areas such as debugging, fault-tolerant systems, security and simulation. It consists of two phases: first a log of the execution is made (record phase), then the log is used to control the re-execution of the program (replay phase). Checkpoints can be used to optimize replay time (but increase the space requirements): execution is not replayed since the very beginning, but just since the last checkpoint.

Main requirements on the log information are that it should be small enough, so that logs of long computations fit in a reasonable storage, and easy to compute, since log production should not affect too much the performance of the running application. Performance of replay is less critical since replay is actually performed only in case of misbehaviour, frequently only for a fragment of the computation, and on a dedicate machine. Replay is also used for reversible debugging [24, 115]: in order to step back in the program under debugging, its execution is replayed but for its last action.

In a sequential setting a log only has to trace nondeterminstic events (e.g., user input, interrupts) thus ensuring that the replayed computation indeed coincides with the original one. Logging becomes more complex in a concurrent scenario (and even more in a distributed one) since scheduling information needs to be stored. There are several approaches to replay concurrent applications. Some of them [36] aim at perfectly replaying the order of execution of actions, by completely logging the scheduling. Perfect replay allows one to reproduce the logged execution. The information stored during the logging phase depends

also on the kind of inter-process communication mechanism used by the system: shared memory [36, 81, 22] or message passing [43, 103].

5 Quantum Programming

Quantum algorithms and communication protocols are described using a language of quantum circuits, describing reversible logical circuits in the language of unitary operations. While this method is convenient in the case of simple algorithms, it is very hard to operate on compound or abstract data types like arrays or integers using this notation. This lack of abstraction motivated the development in the field of quantum programming languages.

One should note that it is now always possible to identify a direct reference to the reversibility in quantum programming languages. They provide some syntax elements – like reverse invocation using exlamation mark (!) – typical for the reversible languages. However, in most cases this is the only explicit element related to reversibility. Of course quantum programming languages are reversible because they use unitary operations as basic building blocks. Thus it is always possible to execute a block which is a reverse of a given block – for a given unitary operator U this is simply its conjugate transpose, $U^\dagger = (U^T)^*$. As such quantum programs are always logicaly reversible and quantum programming languages must ensure this. However this is usually stated implicitly by allowing unitary operations only.

5.1 Imperative paradigm

The first familly of quantum programming languages consists of languages based on the imperative paradigm. Those languages provide syntax known from programming languages like Pascal and C, and extend it with the elemenets necessary to operate on quantum memeory.

QCL QCL (Quantum Computation Language) [107, 108, 109] is one of the most advanced quantum programming language with working interpreter. Its syntax resembles the syntax of C programming language and classical data types are similar to data types in C or Pascal.

The programmes written in QCL can be executed using the available interpreter [106]. The interpreter can be executed in a batch mode or in as an interactive programme. The interpreter is built on a top of `libqc` simulation library written in C++ and offers an excellent speed of execution of simulated programmes. As the simulation of quantum computing requires a considerable amount of computing resources, there were also some attempts to provide a paralellized version of `libqc` library [45].

The basic built-in quantum data type in QCL is **`qureg`** (quantum register). It can be interpreted as the array of qubits (quantum bits).

```
qureg x1[2]; // 2-qubit quantum register x1
qureg x2[2]; // 2-qubit quantum register x2
H(x1); // Hadamard operation on x1
H(x2[1]); // Hadamard operation on the second qubit of the x2
```

Listing 3: Basic operations on quantum registers and subregisters in QCL.

QCL standard library provides standard quantum operators used in quantum algorithms, such as:

- Hadamard `H` and `Not` operations on many qubits,
- controlled not `CNot` with many target qubits and `Swap` gate,
- rotations: `RotX`, `RotY` and `RotZ`,
- phase `Phase` and controlled phase `CPhase`.

QCL supports user-defined operators and functions known from languages like C or Pascal. Classical subroutines are defined using **`procedure`** keyword. Also standard elements, known from C programming language, like looping (*e.g.* **`for`** `i=1` **`to`** `n { ... }`) and conditional structures (*e.g.* **`if`** `x==0 { ... }`), can be used to control the execution of quantum and classical elements. In addition to this, it provides two types of quantum subroutines, which for the core of the reversible part of the lanbguage.

The first type is used for unitary operators. By using it one can define new operations, which in turn can be used to manipulate quantum register. For example operator `diffuse`, defined in Listing 4, defines *inverse about the mean* operator used in Grover's algorithm. Such syntax enabled the higher level of abstraction and extend the library of functions available for a programmer.

```
operator diffuse(qureg q) {
  H(q);              // Hadamard Transform
  Not(q);            // Invert q
  CPhase(pi,q);      // Rotate if q=1111..
  !Not(q);           // undo inversion
  !H(q);             // undo Hadamard Transform
}
```

Listing 4: The implementation of the inverse about the mean operation in QCL [109]. Constant `pi` represents number π. Exclamation mark `!` is used to indicate that the interpreter should use the inverse of a given operator. Operation `diffuse` is used in the quantum search algorithm.

QCL introduces special syntax – exclamation mark `!` – to indicate the inverse of a given operator. Subprocedures defined using **`operator`** keyword can called in the reveres version using `!diffuse` syntax.

QCL utilize different types of quantum memeory to control the operations on quantum registers and to perform optimization of the quantum circuit generation. Quantum memory can be declared using quantum types **`qureg`**, **`quconst`**, **`quvoid`** and **`quscratch`**. Type **`qureg`** is used as a base type for general quantum registers. Other types allow for the optimisation of a generated quantum circuit. The summary of the types defined in QCL is presented in Table 1.

LanQ The second important example of quantum programming language based on the imperative paradigm is LanQ [97]. LanQ was developed to address the problems arising from the lack of elements supporting quantum communication. Additionally, LanQ is the first quantum programming language with full operation semantics specified [98]. This allows for the formal reasoning

Table 1: Types of quantum registers used for memory management in QCL.

Type	Description	Usage
qureg	general quantum register	basic type
quvoid	register which has to be empty when operator is called	target register
quconst	register which must be invariant for all operators used in quantum conditions	quantum conditions
quscratch	register which has to be empty before and after the operator is called	temporary registers

about the correctness of programmes written in LanQ and for the further development of the language. Semantics is also crucial for the optimization of the programmes written in LanQ.

The programmes written in LanQ can be executed and tested using an available interpreter [96]. The interpreter was developed as a part of a PhD thesis [97], but its development stopped in 2007.

Its main feature is the support for creating multipartite quantum protocols. LanQ, as well as cQPL presented in the next section, are built with quantum communication in mind. Thus, in contrast to QCL, they provide the features for facilitating the simulation of quantum communication.

The syntax of the LanQ programming language is very similar to the syntax of C programming language. In particular it supports:

- Classical data types: **`int`** and **`void`**.
- Conditional statements of the form **`if`** `( cond ) { ... }` **`else`** `{ ... }`
- Looping with **`while`** keyword **`while`** `( cond ) { ... }`
- User defined functions, **`int`** `fun(` **`int`** `i) {` **`int`** `res; ...` **`return`** `res; }`

LanQ is built around the concepts of process and interprocess communication, known for example from UNIX operating system. It provides the support for controlling quantum communication between many parties. The implementation of teleportation protocol presented in Listing 6 provides an example of LanQ features, which can be used to describe quantum communication.

Function `main()` in Listing 6 is responsible for controlling quantum computation. The execution of protocol is divided into the following steps:

1. Creation of the classical channel for communicating the results of measurement:
   `channel``[`**`int`**`] c` **`withends`** `[c0,c1];`.
2. Creation of Bell state used as a quantum channel for teleporting a quantum state
   (`psiEPR` **`aliasfor`** `[psi1, psi2]`); this is accomplished by calling external function `createEPR()` creating an entangled state.
3. Instruction **`fork`** executes `alice()` function, which is used to implement a sender; original process continues to run.

```
void alice(channelEnd[int] c0, qbit auxTeleportState) {
  int i;
  qbit phi;
  // prepare state to be teleported
  phi = computeSomething();
  // Bell measurement
  i = measure (BellBasis, phi, auxTeleportState);
  send (c0, i);
}

void bob(channelEnd[int] c1, qbit stateToTeleportOn) {
  int i;
  i = recv(c1);
  // execute one of the Pauli gates according to the protocol
  if (i == 1) {
    Sigma_z(stateToTeleportOn);
  } else if (i == 2) {
    Sigma_x(stateToTeleportOn);
  } else if (i == 3) {
    Sigma_x(stateToTeleportOn);
    Sigma_z(stateToTeleportOn);
  }
  dump_q(stateToTeleportOn);
}
```

Listing 5: Modules used in the quantum teleportation programme implemented in LanQ (see: Listing 6).

```
void main() {
  channel[int] c withends [c0,c1];
  qbit psi1, psi2;
  psiEPR aliasfor [psi1, psi2];

  psiEPR = createEPR();

  c = new channel[int]();
  fork alice(c0, psi1);
  bob(c1, psi2);
}
```

Listing 6: Teleportation protocol implemented in LanQ [97]. Functions `Sigma_x()`, `Sigma_y()` and `Sigma_z()` are responsible for implementing Pauli matrices. Function `createEPR()` (not defined in the listing) creates maximally entangled state between parties — Alice and Bob. Quantum communication is possible by using the state, which is stored in a global variable `psiEPR`. Function `computeSomething()` (not defined in the listing) is responsible for preparing a state to be teleported by Alice.

4. In the last step function `bob()` implementing a receiver is called.

Types in LanQ are used to control the separation between classical and quantum computation. In particular they are used to prohibit copying of quantum

registers. The language distinguishes two groups of variables [97, Chapter 5]:

- Duplicable or non-linear types for representing classical values, *e.g.* **bit**, **int**, **boolean**. The value of a duplicable type can be exactly copied.
- Non-duplicable or linear types for controlling quantum memory and quantum resources, *e.g.* **qbit**, **qtrit** channels and channel ends (see example in Listing 6). Types from this group do not allow for cloning, *ie.* it is impossible to make a copy of such variable.

One should note that quantum types defined in LanQ are mainly used to check validity of the programme before its run. However, such types do not help to define abstract operations. As a result, even simple arithmetic operations have to implemented using elementary quantum gates.

5.2 Functional paradigm

Second familly of quantum programming languages consists of languages based on the functional paradigm. Most of the laguages from this familly are constructed with the focus on the formal properties of the

cQPL Classical elements of cQPL are very similar to classical elements implemented in imperative programming languages. The syntax resembles that of classical programming languages based on C programming language.

In particular cQPL provides conditional structures using **if ... then ... else** block and loops are introduced with **while** keyword.

Quantum memory can be accessed in cQPL using the variables of type **qbit** or **qint**. Basic operations on quantum registers are presented in Listing 7. In particular, the execution of quantum gates is performed by using `*=` operator.

```
new qbit q1 := 0;
new qbit q2 := 1;
// execute CNOT gate on both qubits
q1, q2 *= CNot;
// execute phase gate on the first qubit
q1 *= Phase 0.5;
```

Listing 7: State initialisation and basic gates in cQPL. Data type `qbit` represents a single qubit.

It should be pointed out that **qint** data type provides only a short-cut for accessing the table of qubits.

Only a few elementary quantum gates are built into the language:

- Single qubit gates `H`, `Phase` and `NOT` implementing elementary one-qubits gates.
- `CNot` operator implementing controlled negation,
- `FT(n)` operator for n-qubit quantum Fourier transform.

This set of operations allows to simulate an arbitrary quantum computation. Besides, it is possible to define new gates by specifying their matrix elements directly.

Measurement is performed using **measure**/**then** keywords and `print` command allows to display the value of a variable.

```
measure a then {
  print "a is |0>";
} else {
  print "a is |1>";
};
```

In similar manner like in QCL, it is also possible to inspect the value of a state vector using `dump` command.

The main feature of cQPL is its ability to build and test quantum communication protocols easily. Communicating parties are described using *modules*. In analogy to LanQ, cQPL introduces channels which can be used to send quantum data.

Once again we stress out that the notion of channels used in cQPL and LanQ is different from that used in quantum theory. In quantum mechanics channels, sometimes referred to as operations, are used to describe allowed physical transformations, while in quantum programming they are used to describe communication links.

Communicating parties are described by modules, introduced using `module` keyword. Modules can exchange quantum data (states). This process is accomplished using **`send`** and `receive` keywords.

To compare cQPL and LanQ one can use the implementation of the teleportation protocol. The implementation of teleportation protocol in cQPL is presented in Listing 8, while the implementation in LanQ is provided in Listing 6.

```
module Alice {
    proc createEPR: a:qbit, b:qbit {
       a *= H;
       b,a *= CNot;  /* b: Control, a: Target */
    } in {
      new qbit teleport := 0;
      new qbit epr1 := 0;
      new qbit epr2 := 0;

      call createEPR(epr1, epr2);
      send epr2 to Bob;

      /* teleport: Control, epr1: Target  */
      teleport, epr1 *= CNot;

      new bit m1 := 0;
      new bit m2 := 0;
      m1 := measure teleport;
      m2 := measure epr1;

      /* Transmit the classical measurement results to Bob */
      send m1, m2 to Bob;
};

module Bob {
   receive q:qbit from Alice;
   receive m1:bit, m2:bit from Bob;

   if (m1 = 1) then { q *= [[ 0,1,1,0 ]];  /* Apply sigma_x */ };
```

```
    if (m2 = 1) then { q *= [[ 1,0,0,-1 ]];  /* Apply sigma_z */};

    /* The state is now teleported */
    dump q;
};
```

Listing 8: Teleportation protocol implemented in cQPL (from [93]). Two parties – Alice and Bob – are described by modules. Modules in cQPL are introduced using **module** keyword and can exchange quantum data using **send**/**receive** structure.

One can note tha cQPL modules resemble to some extent the objects used in object-oriented languages.

QML Another quantum programming language following a functional paradigm is QML developed by Altenkirch and Grattage [5, 47]. The QML compiler was described in [48] and can be downloaded from the project web page [46].

The name suggest that QML was designed as a quantum version of ML language [95]. The language, however, is implemented in Haskell and follows some syntactic conventions used in Haskell.

The QML compiler requires GHC (Glasgow Haskell Compiler) [2] in version 6 in order to run QML programmes. In order to run a program written in QML one needs to load the definitions in `qml.hs` into the interactive environment `ghci` and use one of the functions described in Table 2.

Table 2: Possible methods of evaluation of QML programmes [48].

Function	Evaluation method
runM	Unitary matrix representing a reversible part of the programme.
runI	Isometry providing a full description of the programme for terms that produce no garbage.
runS	Superoperator initializing the heap and tracing-out the garbage.

Programme structure A programme written in QML consists of a sequence of function definitions. Each definition is of the form

```
funName (var1,type1) ... (varN,typeN) |- funBody :: retType
```

For example, the classical not function (Cnot) is defined as

```
Cnot (q,qb) |- if q then qtrue
                    else qfalse :: qb
```

Using the same syntax the user can define constants, which in QML are equivalent to functions. For example, to use a constant representing a superposition $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ one can declare it as

```
-- one-qubit superposition
Qsup |- hF*qtrue + hF*qfalse :: qb
```

Here the term **hF** is defined as $\frac{1}{\sqrt{2}}$. The $\star$ operation allows to associate the probability amplitude with a term.

The state of compound systems can be represented using the `()` operation. For example, the constant representing the EPR pair (*ie.* one of the Bell states) is defined as

```
Epr |- hF * (qtrue,qtrue) + hF * (qfalse,qfalse) :: qb*qb;
```

One should note that in the above example the resulting type is described as `qb*qb` and here $\star$ operator is used to describe a type of two-qubit state.

Subroutines As in any functional programming language, programmes written in QML are composed of small functions. This makes the written code more readable and easier to maintain.

Subroutines in QML can operate on an arbitrary number of arguments. A subroutine is introduced by the following syntax

```
ProcName (arg) -|   code
                    code
```

One should note that procedure names have to start with the upper case letter. Moreover, similarly as in Haskell, the indention is important and denotes the continuation of the code block.

```
Cnot (b,qb) |- if b then qfalse else qtrue :: qb;


CNot (s,qb) (b,qb) |- if s then Cnot (b) else b :: qb;

-- classically-controlled quantum Not
CQnot (s,qb) (t,qb) |- if s then Qnot (t) else t :: qb;
```

The quantum CNot operation can be defined in terms of the above by using a quantum conditional operation

```
QCNot (s,qb) (t,qb) |- ifo s then (qtrue,Qnot (t))
                               else (qfalse,t) :: qb * qb;
```

where Qnot is defined as

```
Qnot (b,qb) |- ifo b then qfalse else qtrue :: qb;
```

5.3 Support in general purpose languages

The last method for implementing high-level quantum programming concepts is to extend standard programming languages using specialized library of functions.

Libquantum and Q Language Along with QCL several other imperative quantum programming languages were proposed. Notably Q Language developed by Betteli [15, 16] and libquantum [141] have the ability to simulate noisy environment. Thus, they can be used to study decoherence and analyze the impact of imperfections in quantum systems on the accuracy of quantum algorithms.

Q Language is implemented as a class library for C++ programming language and libquantum is implemented as a C programming language library. Q Language provides classes for basic quantum operations like QHadamard, QFourier, QNot, QSwap, which are derived from the base class Qop. New operators can be defined using C++ class mechanism. Both Q Language and libquantum share some limitations with QCL, since it is possible to operate on single qubits or quantum registers (*ie.* arrays of qubits) only.

Quantum-Octave In a similar fashion the basic high-level structures used for developing quantum programming languages were developed as a set of functions for the general purpose scientific computing system. The structures introduced in [42] are similar to the elements used in QCL and Q Language and were described in quantum pseudo-code based on QCL quantum programming language. They were implemented in GNU Octave language for scientific computing. The procedures used in the implementation are available as a package quantum-octave providing a library of functions, which facilitates the simulation of quantum computing. This package allows also to incorporate high-level programming concepts into the simulation in GNU Octave.

GNU Octave environment is to the large extent compatible with Matlab, thus the described package can be also used with Matlab. As such it connects the features unique for high-level quantum programming languages, with the full palette of efficient computational routines commonly available in modern scientific computing systems.

Quipper Quipper provides an implementation of programming language for quantum computing based on the functional paradigm. The language hasb been describe in [52]. It also gaines a considerable attnsion from the general scientific community [60, 59, 137]. Moreover, some popular quantum algorithms havce been implemented using Quipper [126].

The interpreter is implemented as library for Haskell programming language and, effectively, Quipper programes are written in Haskell. It can be donwloaded from [1]

6 Bidirectional Transformation

A *bidirectional transformation* (BX) is a "mechanism for maintaining the consistency of two (or more) related sources of information" [27]. To be more precise, given a relation $R \subseteq A \times B$ specifying when two artifacts $a \in A$ and $b \in B$ are consistent, denoted $R(a, b)$, the goal of a BX framework is to derive transformations $\overrightarrow{R} : \Delta_A \times B \to \Delta_B$ and $\overleftarrow{R} : \Delta_B \times A \to \Delta_A$ that can be used to restore consistency when one of the artifacts is updated: given $a \in A$ and $b \in B$ such that $R(a, b)$, if a suffers an update $\delta \in \Delta_A$ that may break the consistency between a and b, then $\overrightarrow{R}$ can be applied to obtain an update $\overrightarrow{R}(\delta, b) \in \Delta_B$ that, when applied to b, restores the desired consistency; likewise for $\overleftarrow{R}$. Notice that a BX framework cannot directly handle a synchronization scenario, where both $a \in A$ and $b \in B$ may be concurrently updated.

BX frameworks differ on the kind of updates they handle: most are *state-based* and only consider the outcome of the update, i.e., Δ_A is just a value of

type A; some are *delta-based* [31] and may handle a more detailed characterization of the updates - Δ_A can, for example, include a correspondence relation between elements of A before and after the update, making clear which where deleted, modified or inserted. BX frameworks also differ on the kind of consistency relation $R \in A \times B$ that can be specified: some allow R to be, in principle, an arbitrary relation - a value $a \in A$ can be consistent with with many different value of type B, and vice-versa; some only allow R to be deterministic (or functional) - for every value $a \in A$ there is at most a value $b \in A$ such that $R(a, b)$; some impose further restructions, for example, forcing R to be a bijection.

The BX frameworks that are more related with reversible computation are precisely those where R must be deterministic and updates are state-based. In this case, R can be specified by a transformation (or function) from A to B, and also fulfills the role of $\overrightarrow{R}$, being used to (trivially) recover consistency when an update is performed to a value $a \in A$[2]. This BX scenario is also known as *view-update*, since R can be seen as a function that extracts a view $b \in B$ from $a \in A$, and $\overleftarrow{R}$ is responsible to propagate updates to the view back to the source. In this scenario it is also common to denote R (and $\overrightarrow{R}$) by $\mathsf{get} : A \to B$ and $\overleftarrow{R}$ by $\mathsf{put} : B \times A \to A$ (known as *putback*).

There are two basic properties that a BX view-update framework is usually required to satisfy in order to be *well-behaved* [38]:

$$\forall a \in A, b \in B \cdot \mathsf{get}(\mathsf{put}(b, a)) = b \qquad \text{(PutGet)}$$

$$\forall a \in A \cdot \mathsf{put}(\mathsf{get}(a), a) = a \qquad \text{(GetPut)}$$

To simplify, here we assume get and put to be total functions, but the laws can be trivially generalized to the partial setting. Law PutGet ensures that put can indeed be used to restore consistency, i.e., $\mathsf{put}(b, a)$ must return an a' of which b is a view. Law GetPut forces put to return a null update when the input values are already consistent, i.e., if $\mathsf{get}(a) = b$ then $\mathsf{put}(b, a)$ must return the original a. Notice that GetPut sets a very lax lower-bound on what constitutes a reasonable behavior for put: when the view is updated there is considerable freedom on how to propagate it back into the source. Stronger alternatives to this law have been proposed, for example variants of the *principle of least change* [94], that requires $\mathsf{put}(b, a)$ to return a consistent $a' \in A$ that is as close as possible to the original $a \in A$.

Several approaches have been proposed to obtain well-behaved view-update frameworks. One of the most popular is the *combinatorial* approach first introduced in the so-called *lenses* [38]: a total well-behaved lens l from A to B, denoted $l \in A \rightleftharpoons B$, comprises both $\mathsf{get}_l : A \to B$ and $\mathsf{put}_l : B \times A \to A$, satisfying PutGet and GetPut. A lens framework consists of a set of simple primitive lenses and a set of *combinators* that build more complex lenses out of simpler ones: these combinators ensure "by-construction" that the resulting lenses are well-behaved whenever their parameter lenses are also well-behaved. A simple example of a primitive lens could be $\mathsf{fst} : A \times B \rightleftharpoons A$, to extract the first component of a pair, defined as follows:

$$\mathsf{get}_{\mathsf{fst}}(a, b) = a$$
$$\mathsf{put}_{\mathsf{fst}}(b', (a, b)) = (a, b')$$

[2]Notice that $\overrightarrow{R}$ no longer needs to receive the original $b \in B$ as input, as the updated consistent $b' \in B$ is now uniquely determined by the updated $a' \in A$.

The most fundamental lens combinator is sequential composition that, given lenses $l \in A \rightleftharpoons B$ and $k \in B \rightleftharpoons C$ yields a lens $l;k : A \rightleftharpoons C$, defined as follows:

$$\mathsf{get}_{l;k}(a) = \mathsf{get}_k(\mathsf{get}_l(a))$$
$$\mathsf{put}_{l;k}(c', a) = \mathsf{put}_l(\mathsf{put}_k(c', \mathsf{get}_l(a)), a)$$

Several domain-specific lens frameworks have been proposed, for example, for manipulating string data [17] or to solve the classic database view-update problem [18]. Popular lens libraries have also been implemented for general purpose languages such as Haskell[3] or Scala[4].

Another popular technique to obtain an well-behaved view-update framework is *bidirectionalization*: given the definition of get somehow bidirectionalize it in order derive a suitable put. The *constant complement* approach [11], first proposed in the database community to solve the classic view-update problem, is a standard way to perform *syntactic bidirectionalization*. This approach builds on the fact that an injective function $f : A \to B$ can easily be inverted on order to obtain a (possibly partial) function $f^{-1} : B \to A$. Given $\mathsf{get} : A \to B$, $\mathsf{get}^c : A \to C$ is a *view complement* function if the tupled function $\mathsf{get} \vartriangle \mathsf{get}^c : A \to B \times C$ is injective[5]. Essentially, the goal of the view complement function get^c is to keep all the information that get discards. Given get^c, we can define put as follows:

$$\mathsf{put}(b, a) = (\mathsf{get} \vartriangle \mathsf{get}^c)^{-1}(b, \mathsf{get}^c(a))$$

Notice that is always possible to obtain a view complement function (the identity function is a possibility) – the key issue is to obtain one that produces a complement C as small as possible, so that the above put is as defined as possible, i.e., it can propagate back more updates. [91] is a good example of applying this approach to a non-database domain, namely to bidirectionalize get functions defined in a functional language with support for algebraic data types. In this concrete framework, an updatability checker is also derived, that can be used to determine if a given update can be propagated back.

For particular view functions, namely polymorphic ones, it is also possible to perform *semantic bidirectionalization* [140], that is, define a generic higher-order put function that receives get as a parameter, and executes it in a kind of "simulation mode" to determine how to propagate back an update. The main advantage of this technique is that, since it does not need to impose any syntactic restrictions on how get is defined, it can easily be deployed in a general purpose language (Haskell was the language of choice in [140]). The main drawback (appart from the restriction to polymorphic functions) is poor updatability, namely when handling updates that change the shape of the view. Several extensions have been proposed to overcome these drawbacks, namely combining it with syntactic bidirectionalization [91].

More information about BX can be found in an (early) state of the art survey [27], in the proceedings of the ongoing workshop series on BX, and in the respective community wiki[6].

[3] `https://github.com/ekmett/lens`
[4] `https://github.com/julien-truffaut/Monocle`
[5] The tupled function $f \vartriangle g$ is defined as $\lambda x.(f(x), g(x))$
[6] `http://bx-community.wikidot.com`

7 Robotics

Robots can be defined as generic mechatronic devices controlled by a computer. This definition naturally raises the question of what it means for the computer to be running a reversible program. Given the massive diversity in robotic machinery, the meaning of robotic reversibility is likely to depend on the type of the robot, and is not necessarily meaningful for all kinds of robots.

Industrial robots are segmented tool-manipulating robots often used in production; they are normally controlled by a single host computer. As described in Section 3.4, limited forms of reversibility for debugging and interactive programming has traditionally been a feature found in many major commercial industrial robot systems. An explicit connection between reversibility in robot programming languages and reversible computing was however first made in the context of modular robotics. We describe the use of reversible languages to control modular self-reconfigurable robots and to provide a more general form of control for industrial robots.

7.1 Modular self-reconfigurable robots

Modular robotics is an approach to the design, construction and operation of robotic devices aiming to achieve flexibility and reliability by using a reconfigurable assembly of simple subsystems [131]. Robots built from modular components can potentially overcome the limitations of traditional fixed-morphology systems because they are able to rearrange modules automatically on a by-need basis, a process known as self-reconfiguration. Although self-reconfiguration tends to be a reversible process, physical constraints such as changes in the environment or motion limitations due to gravity may impact the reversibility of a given self-reconfiguration sequence.

Schultz et al investigated the distributed execution of a pre-specified self-reconfiguration sequence in the ATRON modular robot [121, 122]. A sequence is specified using a simple, centralized scripting language named DynaRole. DynaRole programs could be manually written ro automatically generated by a planner. The distributed controller generated from this language allows for parallel self-reconfiguration steps and is highly robust to communication errors and loss of local state due to software failures. Furthermore, the self-reconfiguration sequence can automatically be reversed: any self-reconfiguration process described in the language is reversible, subject to physical constraints. The distributed scripting facility is however limited to specifying straightforward sequences of actions, there is no support for specifying when sequences should execute, nor is there support for e.g. conditionals and local state.

As a concrete example, consider the DynaRole program shown in Listing 9, which is an excerpt of a 65-line program describing a self-reconfiguration sequence for a 7-module ATRON robot changing its shape between a flat shape resembling the number 8 and a car (the 8-shape is an intermediate shape used when changing from a snake to a car). The sequence consists of an unconditional series of steps performed either in sequentially or in parallel. Each step can manipulate a gripper ("retract" or "extend") or rotate the main joint ("rotateFromToBy"). The language is designed to allow statement-level inversion: the inverse of retracting a given gripper is to extend it (and vice versa), and the inverse of rotating from a given position to a target position is to rotate

```
sequence eight2car {
  M0.Connector[$CONNECTOR_0].retract() &
  M3.Connector[$CONNECTOR_4].retract();
  M3.rotateFromToBy(0,324,false,150) ;
  M4.rotateFromToBy(0,108,true,150);
  M4.Connector[$CONNECTOR_0].extend() &
  // ....
  M3.Connector[$CONNECTOR_2].retract() ;
  M1.rotateFromToBy(108,216,true,150) ;
}

sequence car2eight = reverse eight2car;
```

Listing 9: DynaRole sequence describing the "8 to car" self-reconfiguration sequence

back again. Note that rotation commands often do not carry a "from" argument since this is normally given by the current position of the controller. The special keyword "reverse" shown in the last line of the listing provides direct access to the reverse semantics. In addition to defining and reversing sequences, the language also allows sequences to be composed.

In an effort to provide a more general reversible language for controlling modular robots, the language μrRoCE (micro reversible robust collaborative ensembles) was devised as a minimalistic, general-purpose and reversible language for programming modular robot systems [124]. The language was prototyped as an embedded DSL in Scheme integrated with the USSR simulator for modular robots [23], based on code ideas of the RoCE language [123, 120]. The language integrates a minimal set of operations and control flow operators inspired from Janus with a distributed execution model based on state-machines. Essentially, behaviors can be either continuous or reactive based on events, and behaviors are evaluted using either a forwards or a reverse execution semantics.

```
(define DockingCar
  (@Front (if (and (@LeftWheel (module-nearby 1))
                   (@RightWheel (module-nearby 3)))
              (begin (@LeftWheel (connector-extend 1))
                     (@RightWheel (connector-extend 3)))
              'nop
              (and (@LeftWheel (is-connected 1))
                   (@RightWheel (is-connected 3))))))
```

Listing 10: Docking of cars in μrRoCE (details omitted)

As a concrete example, consider the μRoCE program shown in Listing 10. The program describes a docking sequence between to small robot cars made from a "front" module and two "wheel" modules [132]. The "DockingCar" definition triggers on any module playing the role "Front" (which is the otherwise passive module placed between two wheel modules). The definitions consist of a Janus-style reversible conditional that tests if another module is close to the wheel modules, and if so extends the connectors (and otherwise does nothing).

The post-assertion for the conditional states that the wheels should now be connected. Reversing this behavior provides an undocking behavior.

7.2 Industrial robots

Industrial robots are increasingly being used for complex operations like assembly of products. The likelihood of errors however increases with the complexity of the assembly operation. As an extension to the standard use of reversibility for debugging (see also Section 3.4), it has been proposed that certain classes of errors during assembly operations can be addressed using reverse execution, allowing the robot to temporarily back out of an erroneous situation, after which the assembly operation can be automatically retried [80, 119]. The assembly sequence is programmed in a reversible DSL named RASQ, which makes it is possible to automatically derive a disassembly sequence from a given assembly sequence, or vice versa. If an error is encountered because the assembly becomes stuck, the execution direction changes immediately, trying to undo the operation and by getting the assembly "unstuck."

```
object nut, bolt
sequence attach_nut_bolt {
  moveto (...pos above table...)
  pickup (nut, fixed_gripper, (...pos of nut...))
  moveto (...)
  try 3 (force<1) {
    moveto (...pos on bolt...)
    call apply_and_turn_nut
  }
  release (nut, fixed_gripper, (...))
  moveto (...pos above table...)
}

sequence apply_and_turn_nut { ...commands... }
reverse { ...commands that undo apply_and_turn_nut...  }

grip fixed_gripper (nut) { ...commands for gripping a nut... }
```

Listing 11: Sample RASQ program, vector constants and a few other details are omitted for clarity, only the body of "attach_nut_bolt" is shown.

The RASQ language is based on semantic modeling of the actions that the robot makes during the assembly, reversing an assembly sequence is done by reversing the actions using program inversion. A key contribution of RASQ is however that it allows the programmer to explicitly override the definition of the reverse of any given action. For example, the reverse of a simple push to an object is not normally achieved by moving the gripper in the reverse direction. Rather, "unpushing" could be achieved by pulling the object after gripping it, or perhaps pushing it from a different angle.

As a concrete example, consider the RASQ program in Listing 11. A RASQ program consists of declarations of *objects*, *sequences* and *grips*. Objects are to be manipulated by sequences of commands, using grips to pick up and release objects (an object is moved by moving the robot while the object is gripped.)

Key to reversibility is the declaration of a "sequence" as a number of series of steps defined in terms of operations (such as moving the gripper), calls to other sequences, and try-blocks that cause the body to be evaluated a number of times.

8 Control Theory

Control theory and the foundations of reversibility are closely related, beginning from Maxwell's demon and Landauer's principle [13], since Maxwell's demon is a control-based concept and control theory can be interpreted through information theory [134]. This essential relationship is utilised in feedback thermodynamic engines based on reversibility of feedback [63]. Reversibility helps in determining energy cost in control as well [62].

Relationship of control theory and reversibility is twofold. On one hand, systems with the reversibility property (specifically time reversibility) both in classical and quantum setting are well-known and their properties are utilised in control [129]. Another important notion of reversibility in control theory is the reversibility of chemical reactions, so reversible control has been introduced there as well [61].

On the other hand, application of reversible computing in control theory takes several different forms as well. Time-reversible integration, applicable in classical feedback control algorithms, is a notable example [55] . Finally, the question of classical reversible computing, reversible gates and reversible programming in control theoretical applications needs to be addressed. While there have been straightforward conversions of controllers (such as PID) from classical building blocks to reversible logic gates, interesting attempts have been made to provide framework for modern control tools such as fuzzy logic, starting from theoretical foundations [116], proposed real implementations [6] to applications in robotics [112]. Control based on reversible gates for robots has been subject to research in light of quantum computing applications [68, 111].

References

[1] Software available from `http://www.mathstat.dal.ca/~selinger/quipper/`.

[2] The Glasgow Haskell Compiler, 1989-. Software available from the web page `http://www.haskell.org/ghc/`.

[3] ABB Robotics. Rapid overview - technical reference manual, 2004.

[4] Samson Abramsky. A structural approach to reversible computation. *Theor. Comput. Sci.*, 347(3):441–464, 2005.

[5] T. Altenkirch and J. Grattage. A functional quantum programming language. In *Proceedings. 20th Annual IEEE Symposium on Logic in Computer Science*, pages 249–258. IEEE, 2005.

[6] Tomoyuki Araki. On complex vector fuzzy logic and reversible computing. *Journal of Japan Society for Fuzzy Theory and Intelligent Informatics*, 26(1):529–537, 2014.

[7] Kapil Arya, Tyler Denniston, Ana Maria Visan, and Gene Cooperman. Semi-automated debugging via binary search through a process lifetime. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems, PLOS 2013, Farmington, Pennsylvania, USA, November 3-6, 2013*, pages 9:1–9:7. ACM, 2013.

[8] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[9] Holger Bock Axelsen. Clean translation of an imperative reversible programming language. In Jens Knoop, editor, *Compiler Construction. CC '11*, volume 6601 of *LNCS*, pages 142–161. Springer-Verlag, 2011.

[10] Holger Bock Axelsen and Robert Glück. Reversible representation and manipulation of constructor terms in the heap. In Gerhard W. Dueck and D. Michael Miller, editors, *Reversible Computation*, volume 7948 of *LNCS*, pages 96–109, 2013.

[11] François Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981.

[12] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17(6):525–532, November 1973.

[13] Charles H Bennett. Notes on landauer's principle, reversible computation, and maxwell's demon. *Studies In History and Philosophy of Science Part B: Studies In History and Philosophy of Modern Physics*, 34(3):501–510, 2003.

[14] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[15] S. Bettelli. *Toward an architecture for quantum programming.* PhD thesis, Università di Trento, February 2002.

[16] S. Bettelli, L. Serafini, and T. Calarco. Toward an architecture for quantum programming. *Eur. Phys. J. D*, 25(2):181–200, 2003.

[17] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 407–419, New York, NY, USA, 2008. ACM.

[18] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updatable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. ACM.

[19] Geoffrey Brown and Amr Sabry. Reversible communicating processes. In Simon Gay and Jade Alglave, editors, *Proceedings Eighth International Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015, London, UK, 18th April 2015.*, volume 203 of *EPTCS*, pages 45–59, 2016.

[20] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proceedings of HotOS'05: 10th Workshop on Hot Topics in Operating Systems, June 12-15, 2005, Santa Fe, New Mexico, USA*. USENIX Association, 2005.

[21] Shyh-Kwei Chen, W. Kent Fuchs, and Jen-Yao Chung. Reversible debugging using program instrumentation. *IEEE Trans. Software Eng.*, 27(8):715–727, 2001.

[22] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, SPDT '98, pages 48–59, New York, NY, USA, 1998. ACM.

[23] David Johan Christensen, David Brandt, Kasper Stoy, and Ulrik Pagh Schultz. A unified simulator for self-reconfigurable robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'08)*, pages 870–876, France, 2008.

[24] Commercial reversible debugger. `.http://chrononsystems.com/`.

[25] Ivan Lanese Claudio Antares Mezzina, Elena Giachino. Caredeb web site. `http://www.cs.unibo.it/caredeb`.

[26] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. *Bidirectional Transformations: A Cross-Discipline Perspective*, pages 260–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[27] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[28] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR*, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[29] Edsko de Vries, Vasileios Koutavas, and Matthew Hennessy. *Communicating Transactions*, pages 569–583. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[30] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Reversible combinatory logic. *Mathematical Structures in Computer Science*, 16(4):621–637, 2006.

[31] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: The symmetric case. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*, volume 6981 of *LNCS*, pages 304–318. Springer, 2011.

[32] Kevin Donnelly and Matthew Fluet. Transactional events. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 124–135, New York, NY, USA, 2006. ACM.

[33] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[34] FANUC. Operators manual - r-j3ic controller arc tool, 2006.

[35] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, University of Wisconsin, Madison, Wisconsin, USA, May 5-6, 1988*, pages 112–123. ACM, 1988.

[36] Henrique Ferreiro, Vladimir Janjic, Laura M. Castro, and Kevin Hammond. Repeating history: Execution replay for parallel haskell programs. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *Trends in Functional Programming - 13th International Symposium, TFP 2012, St. Andrews, UK, June 12-14, 2012, Revised Selected Papers*, volume 7829 of *Lecture Notes in Computer Science*, pages 231–246. Springer, 2012.

[37] John Field and Carlos A. Varela. Transactors: A programming model for maintaining globally consistent distributed state in unreliable environments. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 195–208, New York, NY, USA, 2005. ACM.

[38] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3), May 2007.

[39] Michael P. Frank. *Reversibility for Efficient Computing*. PhD thesis, MIT, EECS, 1999.

[40] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[42] P. Gawron, J. Klamka, J.A. Miszczak, and R. Winiarczyk. Extending scientifc computing system with structural quantum programming capabilities. *Bull. Pol. Acad. Sci.-Tech. Sci.*, 58(1):77–88, 2010.

[43] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications (awarded best paper!). In Atul Adya and Erich M. Nahum, editors, *Proceedings of the 2006 USENIX Annual Technical Conference, Boston, MA, USA, May 30 - June 3, 2006*, pages 289–300. USENIX, 2006.

[44] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In *FASE*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.

[45] I. Glendinning and B. Ömer. Parallelization of the QC-lib quantum computer simulator library. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 3019 of *Lecture Notes in Computer Science*, pages 461–468. Springer, 2004.

[46] J. Grattage. QML@CS.Nott. Software available from `http://sneezy.cs.nott.ac.uk/QML/compiler/`.

[47] J. Grattage. *QML: A functional quantum programming language*. PhD thesis, School of Computer Science and School of Mathematical Sciences, The University of Nottingham, 2006.

[48] J. Grattage. An overview of QML with a concrete implementation in Haskell. *Electronic Notes in Theoretical Computer Science*, 270(1):157–165, 2011. Proceedings of the Joint 5th International Workshop on Quantum Physics and Logic and 4th Workshop on Developments in Computational Models (QPL/DCM 2008).

[49] Jim Gray. A transaction model. In J. W. de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 1980.

[50] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[51] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[52] A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in quipper. In *Reversible Computation*, pages 110–124. Springer, 2013.

[53] Ralph Grishman. The debugging system AIDS. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1970 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 5-7, 1970*, volume 36 of *AFIPS Conference Proceedings*, pages 59–64. AFIPS Press, 1970.

[54] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Composing first-class transactions. *ACM Trans. Program. Lang. Syst.*, 16(6):1719–1736, November 1994.

[55] Ernst Hairer and Gustaf Söderlind. Explicit, time reversible, adaptive step size control. *SIAM Journal on Scientific Computing*, 26(6):1838–1851, 2005.

[56] Stuart Halloway and Aaron Bedra. *Programming Clojure*. The Pragmatic Programmers, 2 edition, 2012.

[57] Stuart Halloway and Aaron Bedra. *Programming Clojure*. The Pragmatic Programmers, second edition, 2012.

[58] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[59] B. Hayes. Programming your quantum computer. *American Scientist*, 102(1):22–25, 2014.

[60] S. Hebden. New language helps quantum coders build killer apps. *New Scientist*, (5 July), 2013.

[61] Axel A Hoff, Hans H Diebner, and Gerold Baierc. Reversible control of chemical reaction systems. *Zeitschrift für Naturforschung A*, 50(12):1141–1146, 1995.

[62] Jordan M Horowitz and Kurt Jacobs. Energy Cost of Controlling Mesoscopic Quantum Systems. *Physical review letters*, 115(13):130501, 2015.

[63] Jordan M Horowitz and Juan MR Parrondo. Designing optimal discrete-feedback thermodynamic engines. *New Journal of Physics*, 13(12):123019, 2011.

[64] Lorenz Huelsbergen. A logically reversible evaluator for call-by-name lambda calculus. In Tommaso Toffoli, M. Biafore, and Leao J., editors, *Workshop on Physics and Computation, PhysComp '96, Proceedings*. IEEE, 1996.

[65] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 470–494. Springer, 2009.

[66] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony L. Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.

[67] Roshan P. James and Amr Sabry. Theseus: A high level language for reversible computing. 2014.

[68] Eugene Kagan and Irad Ben-Gal. *Navigation of quantum-controlled mobile robots*. INTECH Open Access Publisher, 2011.

[69] Aaron Kimball and Dan Grossman. Software transactions meet first-class continuations. In *8th Annual Workshop on Scheme and Functional Programming*. Citeseer, 2007.

[70] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines (awarded general track best paper award!). In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 1–15. USENIX, 2005.

[71] Werner Kluge. A reversible SE(M)CD machine. In Pieter Koopman and Tommaso Clack, Chrisffoli, editors, *Implementation of Functional Languages*, volume 1868 of *LNCS*, pages 95–113. Springer-Verlag, 2000.

[72] Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. In *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005*, pages 79–88. ACM, 2005.

[73] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with java stm. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[74] Kuka Roboter GmbH. Kuka system software 5.5, 2010.

[75] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In Vishal Misra, Paul Barford, and Mark S. Squillante, editors, *SIGMETRICS 2010, Proceedings of the 2010 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, New York, USA, 14-18 June 2010*, pages 155–166. ACM, 2010.

[76] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[77] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[78] Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent reversibility. *Bulletin of the EATCS*, 114, 2014.

[79] Johan Sund Laursen, Ulrik Pagh Schultz, and Lars-Peter Ellekilde. Automatic error recovery in robot assembly operations using reverse execution. In *Intelligent Robots and Systems (IROS 2015), 2015 IEEE/RSJ International Conference on*. IEEE, 2015.

[80] Johan Sund Laursen, Ulrik Pagh Schultz, and Lars-Peter Ellekilde. Automatic error recovery in robot assembly operations using reverse execution. In *Intelligent Robots and Systems (IROS 2015), 2015 IEEE/RSJ International Conference on*, 2015.

[81] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Computers*, 36(4):471–482, 1987.

[82] George B. Leeman, Jr. A formal approach to undo operations in programming languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(1):50–87, January 1986.

[83] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 157–168, New York, NY, USA, 2011. ACM.

[84] Bil Lewis and Mireille Ducassé. Using events to debug java programs backwards in time. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 96–97. ACM, 2003.

[85] Michael Lienhardt, Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. A reversible abstract machine and its space overhead. In Holger Giese and Grigore Rosu, editors, *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, volume 7273 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2012.

[86] Victor Luchangco and Virendra J. Marathe. Transaction communicators: Enabling cooperation among concurrent transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 169–178, New York, NY, USA, 2011. ACM.

[87] Christopher Lutz and Howard Derby. Janus: A time-reversible language. A letter to R. Landauer. `http://tetsuo.jp/ref/janus.pdf`, 1986.

[88] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. O'Reilly Media, 2013.

[89] Armando B. Matos. Linear programs in a simple reversible language. *Theoretical Computer Science*, 290(3):2063–2074, 2003.

[90] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In Ralf Hinze and Norman Ramsey, editors, *Proc. of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, pages 47–58. ACM, 2007.

[91] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 47–58, New York, NY, USA, 2007. ACM.

[92] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalizing programs with duplication through complementary function derivation. *Computer Software*, 26(2):56–75, 2009. In Japanese.

[93] W. Mauerer. Semantics and simulation of communication in quantum programming. Master's thesis, University Erlangen-Nuremberg, 2005.

[94] Lambert Meertens. Designing constraint maintainers for user interaction. Technical report, 1998. Manuscript available at `www.kestrel.edu/home/people/meertens`.

[95] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML, Revised Edition*. The MIT Press, 1997.

[96] H. Mlnařík. LanQ – a quantum imperative programming language. Software available on-line at `http://lanq.sourceforge.net`.

[97] H. Mlnařík. *Quantum Programming Language LanQ*. PhD thesis, Masaryk University, 2007.

[98] H. Mlnařík. Semantics of quantum programming language LanQ. *Int. J. Quant. Inf.*, 6(1, Supp.):733–738, 2008.

[99] Torben Ægidius Mogensen. Reference counting for reversible languages. In Shigeru Yamashita and Shin-ichi Minato, editors, *Reversible Computation*, volume 8507 of *Lecture Notes in Computer Science*, pages 82–94. Springer International Publishing, 2014.

[100] Torben Ægidius Mogensen. Garbage collection for reversible functional languages. In Jean Krivine and Jean-Bernard Stefani, editors, *Reversible Computation*, volume 9138 of *Lecture Notes in Computer Science*, pages 79–94. Springer International Publishing, 2015.

[101] Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. An injective language for reversible computation. In *Mathematics of Program Construction*, LNCS, pages 289–313. Springer-Verlag, 2004.

[102] Henrik Mühe, Andreas Angerer, Alwin Hoffmann, and Wolfgang Reif. On reverse-engineering the KUKA robot language. *CoRR*, abs/1009.5004, 2010.

[103] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *The Journal of Supercomputing*, 8(4):371–388, 1995.

[104] Naoki Nishida, Adrián Palacios, and Germán Vidal. Reversible Term Rewriting. In *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Leibniz International Proceedings in Informatics (LIPIcs), 2016. *To appear*. Extended version available from the URL: `http://users.dsic.upv.es/~gvidal/german/rr16/`.

[105] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[106] B. Ömer. QCL – a programming language for quantum computers. Software available on-line at `http://tph.tuwien.ac.at/~oemer/qcl.html`.

[107] B. Ömer. A procedural formalism for quantum computing. Master's thesis, Vienna University of Technology, 1998.

[108] B. Ömer. Quantum programming in QCL. Master's thesis, Vienna University of Technology, 2000.

[109] B. Ömer. *Structured Quantum Programming.* PhD thesis, Vienna University of Technology, 2003.

[110] Christos Papadimitriou. *The Theory of Database Concurrency Control.* Computer Science Press, Inc., New York, NY, USA, 1986.

[111] Arushi Raghuvanshi, Yale Fan, Michal Woyke, and Marek Perkowski. Quantum robots for teenagers. In *Multiple-Valued Logic, 2007. ISMVL 2007. 37th International Symposium on*, pages 18–18. IEEE, 2007.

[112] Arushi Raghuvanshi and Marek Perkowski. Fuzzy quantum circuits to model emotional behaviors of humanoid robots. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, 2010.

[113] reversible. `https://reversible.readthedocs.io/en/latest/`.

[114] Michael F. Ringenburg and Dan Grossman. Atomcaml: First-class atomicity via rollback. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 92–104, New York, NY, USA, 2005. ACM.

[115] Michiel Ronsse, Koenraad De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *AADEBUG*, 2000.

[116] R. Rovatti and G. Baccarani. Fuzzy reversible logic. In *1998 IEEE International Conference on Fuzzy Systems Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98CH36228)*, pages 761–766. Institute of Electrical & Electronics Engineers (IEEE), 1998.

[117] Markus Schordan, David R. Jefferson, Peter D. Barnes Jr., Tomas Oppelstrup, and Daniel J. Quinlan. Reverse code generation for parallel discrete event simulation. In Jean Krivine and Jean-Bernard Stefani, editors, *Reversible Computation - 7th International Conference, RC 2015, Grenoble, France, July 16-17, 2015, Proceedings*, volume 9138 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2015.

[118] Markus Schordan, Tomas Oppelstrup, David Jefferson, Peter D. Barnes Jr., and Daniel J. Quinlan. Automatic generation of reversible C++ code and its performance in a scalable kinetic monte-carlo application. In Richard Fujimoto, Brian W. Unger, and Christopher D. Carothers, editors, *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, SIGSIM-PADS 2016, Banff, Alberta, Canada, May 15-18, 2016*, pages 111–122. ACM, 2016.

[119] Ulrik P. Schultz, Johan S. Laursen, Lars-Peter Ellekilde, and Holger B. Axelsen. Towards a domain-specific language for reversible assembly sequence. In *Reversible Computation*, 2015.

[120] Ulrik Pagh Schultz. Towards a robust spatial computing language for modular robots. In *Proceedings of the 2012 Workshop on Spatial Computing*, Spain, June 2012.

[121] Ulrik Pagh Schultz, Mirko Bordignon, and Kasper Stoy. Robust and Reversible Self-Reconfiguration. In *Proc. 2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'09)*, pages 5287–5294, St. Louis MO, USA, October 11-15 2009.

[122] Ulrik Pagh Schultz, Mirko Bordignon, and Kasper Stoy. Robust and reversible execution of self-reconfiguration sequences. *Robotica*, 29:35–57, 2011.

[123] U.P. Schultz. Poster: Programming language abstractions for self-reconfigurable robots. In *Systems, Programming, and Applications: Software for Humanity (SPLASH 2012)*, pages 69–70, New York, NY, USA, 2012. ACM.

[124] U.P. Schultz. Towards a general-purpose, reversible language for controlling self-reconfigurable robots. In *RC 2013*, volume 7581 of *LNCS*, pages 97–111. Springer, 2013.

[125] Nir Shavit and Alexander Matveev. *Encyclopedia of Algorithms*, chapter Transactional Memory, pages 1–4. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[126] S. Siddiqui, M.J. Islam, and O. Shehab. Five quantum algorithms using quipper. arXiv:1406.4481, 2014.

[127] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 191–210, New York, NY, USA, 2007. ACM.

[128] Carlo Spaccasassi and Vasileios Koutavas. *Trends in Functional Programming: 14th International Symposium, TFP 2013, Provo, UT, USA, May 14-16, 2013, Revised Selected Papers*, chapter Towards Efficient Abstractions for Concurrent Consensus, pages 76–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[129] Julien Clinton Sprott, William Graham Hoover, and Carol Griswold Hoover. Heat conduction, and the lack thereof, in time-reversible dynamical systems: Generalized Nosé-Hoover oscillators with a temperature gradient. *Physical Review E*, 89(4):042914, 2014.

[130] Bill Stoddart, Robert Lynas, and Frank Zeyda. A virtual machine for supporting reversible probabilistic guarded command languages. *Electr. Notes Theor. Comput. Sci.*, 253(6):33–56, 2010.

[131] Kasper Stoy, David Brandt, and David Johan Christensen. *An Introduction to Self-Reconfigurable Robots*. MIT Press, 2010.

[132] Kasper Stoy, David Johan Christensen, David Brandt, Mirko Bordignon, and Ulrik Pagh Schultz. Exploit morphology to simplify docking of self-reconfigurable robots. In *Proc. Int. Symp. on Distributed Autonomous Robotic Systems (DARS'08)*, pages 441–452, Tsukuba, Japan, 2008.

[133] Michael Kirkedal Thomsen and Holger Bock Axelsen. Interpretation and programming of the reversible functional language. In *Implementation and Application of Functional Programming Languages (IFL)*, 2015.

[134] Hugo Touchette and Seth Lloyd. Information-theoretic approach to the study of control systems. *Physica A: Statistical Mechanics and its Applications*, 331(1):140–172, 2004.

[135] Kishor S. Trivedi. *Probability and statistics with reliability, queuing, and computer science applications*. Prentice Hall, 1982.

[136] Commercial reversible debugger. `http://undo-sotware.com`.

[137] B. Valiron, N.J. Ross, P. Selinger, D.S. Alexander, and J.M. Smith. Programming the quantum future. *Communications of the ACM*, 58(8):52–61, 2015.

[138] P. Van Roy and S. Haridi. *Concepts, Techniques and Models of Computer Programming*. MIT Press, 2004.

[139] Ana Maria Visan, Artem Polyakov, Praveen S. Solanki, Kapil Arya, Tyler Denniston, and Gene Cooperman. Temporal debugging using URDB. *CoRR*, abs/0910.5046, 2009.

[140] Janis Voigtländer. Bidirectionalization for free! (pearl). In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 165–176, New York, NY, USA, 2009. ACM.

[141] H. Weimer. The C library for quantum computing and quantum simulation. `http://www.libquantum.de/`.

[142] Guoqing (Harry) Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 85–94. ACM, 2007.

[143] Tetsuo Yokoyama. Reversible computation and reversible programming languages. *Electr. Notes Theor. Comput. Sci.*, 253(6):71–81, 2010.

[144] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Reversible flowchart languages and the structured reversible program theorem. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming. Proceedings*, volume 5126 of *LNCS*, pages 258–270. Springer-Verlag, 2008.

[145] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Towards a reversible functional language. In Alexis De Vos and Robert Wille, editors, *Reversible Computation, RC '11*, volume 7165 of *LNCS*, pages 14–29. Springer-Verlag, 2012.

[146] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Fundamentals of reversible flowchart languages. *Theoretical Computer Science*, 2015. dx.doi.org/10.1016/j.tcs.2015.07.046. Article in press.

[147] Tetsuo Yokoyama and Robert Glück. A reversible programming language and its invertible self-interpreter. In *Partial Evaluation and Program Manipulation. PEPM '07*, pages 144–153. ACM, 2007.

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

[149] Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, pages 81–91. ACM, 2006.

[150] Lukasz Ziarek and Suresh Jagannathan. Lightweight checkpointing for concurrent ML. *J. Funct. Program.*, 20(2):137–173, 2010.

[151] Paolo Zuliani. Logical reversibility. *IBM Journal of Research and Development*, 45(6):807–818, 2001.