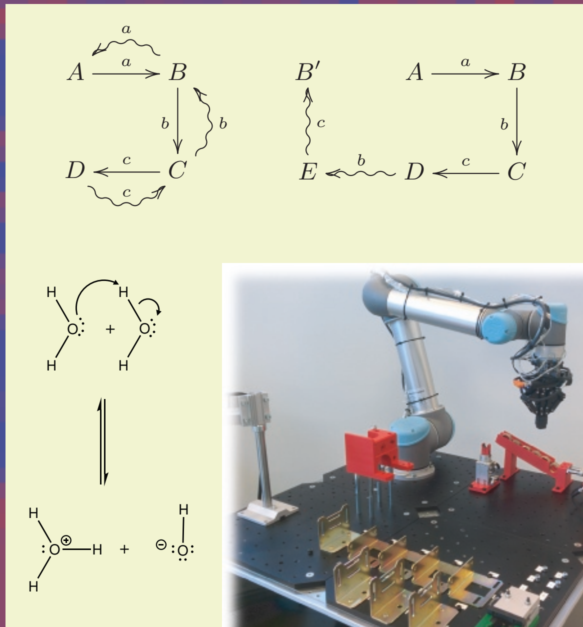
State-of-the-Art Survey

LNCS 12070

Irek Ulidowski
Ivan Lanese
Ulrik Pagh Schultz
Carla Ferreira (Eds.)

Reversible Computation: Extending Horizons of Computing

Selected Results of the COST Action IC1405



Springer Open

COST
EUROPEAN COOPERATION
IN SCIENCE & TECHNOLOGY

Lecture Notes in Computer Science 12070

Founding Editors

Gerhard Goos
Karlsruhe Institute of Technology, Karlsruhe, Germany

Juris Hartmanis
Cornell University, Ithaca, NY, USA

Editorial Board Members

Elisa Bertino
Purdue University, West Lafayette, IN, USA

Wen Gao
Peking University, Beijing, China

Bernhard Steffen
TU Dortmund University, Dortmund, Germany

Gerhard Woeginger
RWTH Aachen, Aachen, Germany

Moti Yung
Columbia University, New York, NY, USA

More information about this series at http://www.springer.com/series/7407

Irek Ulidowski · Ivan Lanese ·
Ulrik Pagh Schultz · Carla Ferreira (Eds.)

Reversible Computation: Extending Horizons of Computing

Selected Results of the COST Action IC1405

Springer Open

cost
EUROPEAN COOPERATION
IN SCIENCE & TECHNOLOGY

Editors

Irek Ulidowski
University of Leicester
Leicester, UK

Ivan Lanese
University of Bologna
Bologna, Italy

Ulrik Pagh Schultz
University of Southern Denmark
Odense, Denmark

Carla Ferreira
NOVA University Lisbon
Caparica, Portugal

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-47360-0 ISBN 978-3-030-47361-7 (eBook)
https://doi.org/10.1007/978-3-030-47361-7

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Acknowledgement and Disclaimer
This publication is based upon work from COST Action IC1405 Reversible Computation: Extending Horizons of Computing, supported by COST (European Cooperation in Science and Technology).
The book reflects only the authors' views. Neither the COST Association nor any person acting on its behalf is responsible for the use, which might be made of the information contained in this publication. The COST Association is not responsible for external websites or sources referred to in this publication.

© The Editor(s) (if applicable) and The Author(s) 2020. This book is an open access publication.
Open Access This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.
The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

European Cooperation in Science and Technology (COST)

This publication is based upon work from COST Action IC1405 Reversible Computation - Extending Horizons of Computing, supported by COST (European Cooperation in Science and Technology).

COST is a funding agency for research and innovation networks. Our Actions help connect research initiatives across Europe and enable scientists to grow their ideas by sharing them with their peers. This boosts their research, career, and innovation.

www.cost.eu

Funded by the Horizon 2020 Framework Programme
of the European Union

Preface

Reversible Computation (RC) is a new paradigm that extends the traditional forwards-only mode of computation with the ability to execute in reverse, so that computation can run backwards as easily as forwards. It aims to deliver novel computing devices and software, and to enhance existing systems by equipping them with reversibility. There are many potential applications of RC, including languages and software tools for reliable and recovery-oriented distributed systems and revolutionary reversible logic gates and circuits, but they can only be realised and have lasting effect if conceptual and firm theoretical foundations are established first. This state-of-the-art survey presents the main recent scientific outcomes in the area of RC, focusing on those that have emerged during COST Action IC1405 Reversible Computation - Extending Horizons of Computing, a European research network that operated from May 2015 to April 2019.

Action IC1405 was organised into four Working Groups. The members of Working Group 1 concentrated their efforts on establishing Foundations of RC. Working Groups 2 and 3 focused on specific technical challenges and potential application areas of reversibility in Software and Systems and in Reversible Circuit Design respectively. The purpose of Working Group 4 was to validate and explore application of Action's research results via practical case studies.

Working Groups 1–3 produced yearly scientific reports during the life of the Action, and these reports have been developed further into four comprehensive chapters surveying the main conceptual, theoretical, and technical achievements of the RC Action. Seven of the case studies from Working Group 4 were selected for presentation in this book. They show that RC techniques can form essential parts of solutions to many difficult practical problems as can be seen, for example, in the success of reversible debugging software tools. Overall, there are 40 co-authors of the book, which represents a substantial proportion of around 110 active members of the RC Action. This survey is a result of collaborative work that was carried out in part during regular Action meetings and Short-Term Scientific Missions (STSMs) supported by COST.

The content of the survey is structured as follows:

- Chapter 1 presents many new theoretical developments in the foundations of RC. It is worth noting the work on reversing Petri nets and on categorical characterisation of reversibility which was carried out as a direct result of the members of the respective communities participation in IC1405. Results obtained by Working Group 1 on reversibility in programming languages, term rewriting, membrane systems, process calculi, automata, and quantum formal verification are also given here.
- The main results obtained in the area of reversible software and systems are described in Chapter 2. They span from the definition of imperative and reversible object-oriented languages to the impact of reversibility on analysis techniques based on behavioural types, and to the application of reversibility for recovery, efficient

simulation, and wireless communications. The outcomes of Working Group 2 have been mostly of practical nature, hence some of the topics above are further discussed in the chapters of the book devoted to case studies.

- Chapter 3 covers simulation and design techniques for quantum circuits. Quantum circuits are inherently reversible and have received significant attention in the recent years. Simulating and designing them in a proper fashion is however a non-trivial task. The chapter provides an overview of solutions for these tasks which utilise expertise on efficient data structures and algorithms gained in the design of conventional circuits and systems.
- An overview of recent results towards a new classification of reversible functions, which would be useful in the synthesis of reversible circuits, is presented in Chapter 4. Firstly, theoretical results on properties of component functions of reversible functions are given. Then, the results of recent research on the existence of Boolean reversible functions of any number of variables (with all component functions belonging to different equivalence classes) are described. Finally, results on the existence of Boolean reversible functions with specified properties of all component functions are reported.
- Chapter 5 focuses on the application of reversibility to debugging. This is a quite natural application, since debugging aims at finding bugs (that is, wrong lines of code) causing visible misbehaviours, and to do that it is quite natural to execute backward from the misbehaviour. The chapter focuses on debugging of concurrent systems, where the use of reversibility is more recent, and considers both a standard imperative language and a subset of the functional language Erlang. Notably, the results described in this section are practical, but obtained as a direct application of theoretical investigations in the area of process calculi and semantics.
- The combination of reversibility and run-time monitoring of distributed systems is advocated in Chapter 6. It considers Erlang programs as an instance of the implementation of a model-driven methodology which can also be applied to other message-passing frameworks. Reversible choreographies are introduced to abstractly represent message-passing software and are used to specify adaptation and recovery strategies. These specifications are then used to generate monitors that govern the recovery and run-time adaptation of the execution according to the specified recovery policies.
- Chapter 7 give an overview of process calculi and Petri nets techniques for the modeling and reasoning about reversibility of systems, including out-of-causal-order reversibility as in chemical reactions. As an example, the autoprotolysis of water reaction is modeled in the Calculus of Covalent Bonding, the Bonding Calculus, and in Reversing Petri Nets.
- A robotic assembly case study is presented in Chapter 8. It investigates to what extent program inversion of a robotic assembly sequence can be considered to derive a reverse behaviour, and to what extent changing the execution direction at runtime (namely backtracking and retrying) using program inversion can be used as an automatic error handling procedure. The programming model is used to reversibly control industrial robots and demonstrates reversible control of industrial robots in real-world scenarios.

- Chapter 9 presents practical results in the field of optimistic parallel discrete event simulation (PDES). Optimistic PDES requires reversibility to perform a distributed roll-back in case conflicts are detected due to the optimistic execution approach. Two approaches to reversibility are compared: one based on the reversible programming language Janus, the other based on a variant of checkpointing, also called incremental state saving. For the purpose of comparing the performance of the two approaches, a benchmark simulation model is presented which is specifically designed for evaluating the performance of approaches to reversibility in PDES.
- A case study on applications of RC in wireless communications is given in Chapter 10. A communication system has an inherent link with RC. It is demonstrated that the communication channel can be modeled using reversible paradigms such as reversible cellular automata, that the hardware conducting communications based on wave time reversal has a natural, simple implementation in terms of reversible gates, and, lastly, that optimisation for large antenna arrays can be efficiently done in real time using reversible computational models such as Reversing Petri Nets.
- Finally, Chapter 11 provides an overview of key reconciliation techniques in quantum key distribution protocols with a focus on communication and computing performance. Different ways to identify errors in establishing symmetric cryptographic keys are investigated, with a focus on recursivity and reversibility. This is particularly noticeable with the Cascade Protocol, while other protocols focus on achieving one-sided processing which is of great importance for satellite quantum communications. Also, a new approach to key reconciliation techniques based on artificial neural networks is introduced.

We are grateful to all the contributors of this book, who worked tirelessly preparing the chapters and improving them greatly following a review process. Our thanks are due to many reviewers who helped to improve the scientific quality of the book. We would like to thank Veroniva Gaspes, the STSM Coordinator of Action IC1405, for dealing efficiently with over 80 STSM visits. We also thank Jovanka Pantović for taking care of ICT conference grants.

We would like to express our appreciation to Ralph Stübner, the Scientific Officer of the Action, for the support and advice received over the four years of the Action. Our administrative and financial affairs were looked after very effectively by Olga Gorczyca from COST. Our special thanks also go to Alfred Hofmann, Anna Kramer and Elke Werner, and other members of the editorial team at Springer, for their efficient and patient editorial assistance.

March 2020

Irek Ulidowski
Ivan Lanese
Ulrik Pagh Schultz
Carla Ferreira

Organization

Action IC1405 Committee

Action Scientific Officer

Ralph Stübner	COST Association, Belgium

Action Chair

Irek Ulidowski	University of Leicester, UK

Action Vice-chair

Ivan Lanese	Focus Team, University of Bologna/Inria, Italy

Working Group (WG) Leaders and Co-leaders

WG1 Leader

Iain Phillips	Imperial College London, UK

WG1 Co-leader

Michael Kirkedal Thomsen	University of Copenhagen, Denmark

WG2 Leader

Claudio Antares Mezzina	University of Urbino, Italy

WG2 Co-leader

Rudolf Schlatte	University of Oslo, Norway

WG3 Leader

Robert Wille	Johannes Kepler University, Austria

WG3 Co-leader

Paweł Kerntopf	Warsaw University of Technology, Poland

WG4 Leader

Ulrik Pugh Schultz	University of Southern Denmark, Denmark

WG4 Co-leader

Carla Ferreira	University of Lisbon, Portugal

STSM Coordinator

Veronica Gaspes	University of Halmstad, Sweden

ITC Conference Grants Coordinator

Jovanka Pantović	University of Novi Sad, Serbia

COST Action Equality Chair

Anna Philippou	University of Cyprus, Cyprus

COST Action Website Chair

Michael Kirkedal Thomsen	University of Copenhagen, Denmark

Additional Reviewers

Aman, Bogdan
Ciobanu, Gabriel
Di Giusto, Cinzia
Francalanza, Adrian
Giunti, Marco
Glück, Robert
Hoey, James
Kerntopf, Pawel
Krivine, Jean
Mehic, Miralem
Mezzina, Claudio
Niemann, Philipp
Philippou, Anna
Podlaski, Krzysztof
Schlatte, Rudolf
Schordan, Markus
Tuosto, Emilio
Vidal, German
Wille, Robert
Worsch, Thomas
Yokoyama, Tetsuo

Contents

Foundations of Reversible Computation 1
Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan, Iain Phillips, G. Michele Pinna, Luca Prigioniero, Irek Ulidowski, and Germán Vidal

Software and Reversible Systems: A Survey of Recent Activities 41
Claudio Antares Mezzina, Rudolf Schlatte, Robert Glück, Tue Haulund, James Hoey, Martin Holm Cservenka, Ivan Lanese, Torben Æ. Mogensen, Harun Siljak, Ulrik P. Schultz, and Irek Ulidowski

Simulation and Design of Quantum Circuits 60
Alwin Zulehner and Robert Wille

Research on Reversible Functions Having Component Functions with Specified Properties: An Overview 83
Paweł Kerntopf, Claudio Moraga, Krzysztof Podlaski, and Radomir Stanković

A Case Study for Reversible Computing: Reversible Debugging of Concurrent Programs 108
James Hoey, Ivan Lanese, Naoki Nishida, Irek Ulidowski, and Germán Vidal

Towards Choreographic-Based Monitoring 128
Adrian Francalanza, Claudio Antares Mezzina, and Emilio Tuosto

Reversibility in Chemical Reactions 151
Stefan Kuhn, Bogdan Aman, Gabriel Ciobanu, Anna Philippou, Kyriaki Psara, and Irek Ulidowski

Reversible Control of Robots 177
Ulrik Pagh Schultz

Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event Simulation 187
Markus Schordan, Tomas Oppelstrup, Michael Kirkedal Thomsen, and Robert Glück

Reversible Computation in Wireless Communications 208
Harun Siljak

Error Reconciliation in Quantum Key Distribution Protocols 222
Miralem Mehic, Marcin Niemiec, Harun Siljak, and Miroslav Voznak

Author Index 237

Foundations of Reversible Computation

Bogdan Aman[1,2], Gabriel Ciobanu[1,2], Robert Glück[3], Robin Kaarsgaard[3], Jarkko Kari[4], Martin Kutrib[5], Ivan Lanese[6], Claudio Antares Mezzina[7], Łukasz Mikulski[8], Rajagopal Nagarajan[9], Iain Phillips[10](✉), G. Michele Pinna[11], Luca Prigioniero[5,12], Irek Ulidowski[13], and Germán Vidal[14]

[1] Romanian Academy, Institute of Computer Science, Iaşi, Romania
baman@iit.tuiasi.ro
[2] A.I. Cuza University, Iaşi, Romania
gabriel@info.uaic.ro
[3] University of Copenhagen, Copenhagen, Denmark
{glueck,robin}@di.ku.dk
[4] University of Turku, Turku, Finland
jkari@utu.fi
[5] University of Giessen, Giessen, Germany
kutrib@informatik.uni-giessen.de
[6] Focus Team, University of Bologna/Inria, Bologna, Italy
ivan.lanese@gmail.com
[7] Università di Urbino, Urbino, Italy
claudio.mezzina@uniurb.it
[8] Folco Team, Nicolaus Copernicus University, Toruń, Poland
mikulskilukasz@gmail.com
[9] Middlesex University, London, England
R.Nagarajan@mdx.ac.uk
[10] Imperial College London, London, England
i.phillips@imperial.ac.uk
[11] Università di Cagliari, Cagliari, Italy
gmpinna@unica.it
[12] Università degli Studi di Milano, Milan, Italy
prigioniero@di.unimi.it
[13] University of Leicester, Leicester, England
iu3@leicester.ac.uk
[14] MiST, VRAIN, Universitat Politècnica de València, Valencia, Spain
gvidal@dsic.upv.es

Abstract. Reversible computation allows computation to proceed not only in the standard, forward direction, but also backward, recovering past states. While reversible computation has attracted interest for its multiple applications, covering areas as different as low-power computing, simulation, robotics and debugging, such applications need to be supported by a clear understanding of the foundations of reversible computation. We report below on many threads of research in the area of foundations of reversible computing, giving particular emphasis to the results obtained in the framework of the European COST Action IC1405, entitled "Reversible Computation - Extending Horizons of Computing", which took place in the years 2015–2019.

© The Author(s) 2020
I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 1–40, 2020.
https://doi.org/10.1007/978-3-030-47361-7_1

1 Introduction

Reversible computation allows computation to proceed not only in the standard, forward direction, but also backward, recovering past states, and computing inputs from outputs. Reversible computation has attracted interest for multiple applications, covering areas as different as low-power computing [113], simulation [37], robotics [122] and debugging [129]. However, such applications need to be supported by a clear understanding of the foundations of reversible computation. Over the years, a number of theoretical aspects of reversible computing have been studied, dealing with categorical foundations of reversibility, foundations of programming languages and term rewriting, considering various models of sequential (automata, Turing machines) and concurrent (cellular automata, process calculi, Petri nets and membrane computing) computations, and tackling also the challenges posed by quantum computation, which is in a large part naturally reversible. We report below on those threads of research, giving particular emphasis to the results obtained in the framework of the European COST Action IC1405 [78], titled "Reversible Computation - Extending Horizons of Computing", which took place in the years 2015–2019 and involved researchers from 34 different countries.

The contents of this chapter are as follows. Section 2 covers category theory, Sect. 3 reversible programming languages, Sect. 4 term rewriting, and Sect. 5 membrane computing. We then discuss process calculi (Sect. 6), Petri nets (Sect. 7), automata (Sect. 8), and quantum verification and machine learning (Sect. 9). The chapter ends with a brief conclusion (Sect. 10).

2 Category Theory

Category theory is a framework for the description and development of mathematical structures. In category theory mathematical objects and their relationships within mathematical theories are abstracted into primal notions of *object* and *morphism*. Despite being a staple of the related field of quantum computer science for years (see, *e.g.*, [3,79,174]), category theory has seen comparatively little use in modelling reversible computation, where operational methods remain the standard. While the present section aims to give an overview of the use of categorical models in providing categorical semantics for reversible programming languages, categorical models have also been studied for other reversible computing phenomena, notably reversible event structures [65].

2.1 Dagger Categories

One approach to categorical models of reversible computation is given by dagger categories, *i.e.*, categories with an abstract notion of inverse given by assigning to each morphism $X \xrightarrow{f} Y$ an *adjoint* morphism $Y \xrightarrow{f^\dagger} X$, such that $(g \circ f)^\dagger = f^\dagger \circ g^\dagger$ and $\mathrm{id}_X^\dagger = \mathrm{id}_X$ (that is, composition is respected) and $f^{\dagger\dagger} = f$ for all compatible morphisms f and g. Note that this definition says nothing about how f and $f^\dagger$

ought to interact. As such, $f^\dagger$ is not required to "undo" the behaviour of f in any way, but can be *any* morphism with the appropriate signature, so long as the above constraints are met.

A useful specialisation of dagger categories, in connection with reversible computation, is dagger traced symmetric bimonoidal (or *rig*) categories, *i.e.*, dagger categories equipped with two symmetric monoidal tensors (usually denoted $- \oplus -$ and $- \otimes -$), interacting through a distributor and an annihilator, yielding the structure of a *rig* (*i.e.*, a ring without additive inverses). Iteration is modelled by means of a trace operator Tr (see [1,85,175]) such that $(\mathrm{Tr} f)^\dagger = \mathrm{Tr}(f^\dagger)$. These categories are strongly related to the dagger compact closed categories [3,174] that serve as the model of choice for the Oxford school of quantum computing.

The use of dagger traced symmetric bimonoidal categories to model reversible computations goes back at least as far as to the works by Abramsky, Haghverdi and Scott (see, *e.g.*, [2,4]) on (reversible) combinatory algebras, though its applications in reversible programming were perhaps best highlighted by the development of the Π and Π^0 calculi [34,83]. In addition, the reversible functional programming language Theseus [82] exhibits a correspondence with the Π^0 calculus. However, dagger traced symmetric bimonoidal categories are not strictly enough to model Π^0, as such categories fail to account for the recursive data types formed using $- \oplus -$, $- \otimes -$, and their units. In his recent thesis, Karvonen [94] describes precisely the categorical features necessary for such a correspondence, which he calls traced ω-continuous dagger rig categories.

Another notable application of this line of research is found in [167], where a reversible Π^0-like language is extended to describe quantum computations without measurement, but with support for (necessarily terminating) primitively recursive functions.

2.2 Inverse Categories

Another approach to model reversible computation is inverse categories [95] (see [40] for a more modern presentation), a specialisation of dagger categories in which morphisms are required to be partial isomorphisms. More precisely, each morphism $X \xrightarrow{f} Y$ may be *uniquely* assigned a *partial inverse* $Y \xrightarrow{f^\dagger} X$ satisfying $f \circ f^\dagger \circ f = f$.

The development of inverse categories as models of reversible computation was pioneered in the thesis of B.G. Giles [58], though a concrete correspondence was never provided. This work, combined with the comprehensive account of inverse categories with joins given in the thesis of Guo [67], was exploited in [86] to give an account of reversible recursion in inverse categories with joins.

Much of this theory was then put to use in [87], where the authors managed to show soundness, adequacy, and (under certain conditions) full abstraction for reversible flowchart languages [185] in a class of inverse categories with joins.

2.3 Monads and Arrows for Reversible Effects

The first account of monads pertaining to reversible computing was given in [71] as *dagger Frobenius monads*. Though these arise naturally in quantum computation in the context of measurement, it turns out that they are exceedingly rare in the case of classical reversible computing. A better concept for modelling and programming with reversible effects turns out to be that of *dagger* and *inverse arrows* [70], with examples such as reversible computation with mutable memory, errors and error handling, and more.

3 Foundations of Reversible Programming Languages

Reversible programming languages bridge the gap between the hardware and the specific application, and therefore play a central role in the development of reversible computing. Reversible languages must be expressive and usable in a variety of application domains. Their semantics must be precise and their programs accessible to program inversion, analysis and verification. Additionally, they must have efficient realisations on reversible devices and on standard ones. Recent programming language studies have advanced the foundations and theory of reversible languages in several interrelated directions.

3.1 Language Cores

Reversible languages have been reduced to their computational cores:

R-Core [63] is a structured reversible language consisting of a single command for *reversible store updates*, a single control-flow operator for *reversible iteration*, and data structures built from a single binary constructor and a single symbol. Despite its extreme simplicity, the language is *reversibly universal*, which means it is as computationally powerful as any reversible language can be. Its four-line program inverter is as concise as the one for Bennett's reversible Turing machines. The core language and a recent extension with *reversible recursion* were equipped with a denotational semantics [61,63,64].

R-While [62] adds reversible *rewrite rules* and *pattern matching* as syntactic sugar to R-Core, which makes the family of structured reversible languages more accessible to foundational studies and educational purposes than do reversible Turing machines and other reversible devices. The *procedural extension* [64] draws a distinction between tail-recursion by iteration and general recursion by reversible procedures, a notoriously difficult transformation problem in program inversion [96,151]. The *linear-time self-interpretability* makes the language also suitable for foundational studies of computability and complexity from a programming language perspective [84].

CoreFun [80] is a typed reversible functional language that seeks to reduce reversible functional programming [184] to its essentials so that it can serve as a foundation for modern functional language concepts. The language has a formal semantics and a type system to statically check for reversibility of programs.

3.2 Formal Semantics

Precise semantics is the foundation of every programming language, and formality is from where programming languages derive their usefulness and power.

A program is regarded as reversible if each of its meaningful subprograms is partially invertible. Thus, *reversible programs have reversible semantics* [61]. A foundation of the semantics has been established for structured reversible languages built on inverse categories [59,60]. This class of languages includes Janus, a reversible language that was originally formalised by conventional (irreversible) operational semantics, and the R-Core and R-While languages. For example, predicates and assertions occurring in reversible alternatives and reversible iterations are modelled by decision maps, in contrast to conventional semantics. A benefit of the reversible semantic approach is that program inverters and equivalences of reversible programs can be derived directly from the semantics.

The assumption of countable joins in inverse categories is suitable in a categorical account of reversible recursion [86], which enables modelling of procedures in reversible structured and functional languages. Reversibility of Janus was proved with a proof assistant [153].

3.3 Compilation Principles

High-level languages are more productive in most application domains, but high levels of computational abstractions do not come for free. A clean and effective translation to lower abstraction levels is required and sophisticated optimisations may be necessary to generate high quality implementations.

Dynamic memory management is a central runtime mechanism to support dynamic data structures in reversible machines. Its purpose is to support reversible object-oriented languages as well as the core languages described above. Garbage collectors that use multiple references [142] to overcome linearity requirements and heap manager algorithms have been developed and experimentally evaluated. To ease the analysis and optimisation when translating from a high-level reversible language to the underlying reversible machine, the *reversible single static assignment* (RSSA) form can be a suitable intermediate representation in optimising compilers [141]. Its aim is to allow for advanced optimisations such as register allocation on reversible Von Neumann machines.

The recent languages Joule [173] and ROOPL [68] demonstrated that well-known *object-oriented concepts* can be captured reversibly by extending a Janus-like imperative language. *Reversible data types* [43], that is data structures with all of its associated operations implemented reversibly, are enabled by dynamic allocation of constructor terms on the heap [11]. A reversible dynamic memory management based on the Buddy Memory system [99] has been developed and tested in a compiler targeting the assembly language of a reversible computer [43].

3.4 Reversibilisation Techniques

A separate approach to reversibility is *reversibilisation*, which turns irreversible computations into reversible computations. This can be achieved by extending the semantics of an irreversible language or by instrumenting an irreversible program to continually produce information that ensures reversibility.

Some reversibilisation techniques work without user interaction, while others require annotation of programs. Techniques have been developed in recent years that add tracing to term rewriting systems [150] and instrument C++ programs with incremental state saving [171]. Other investigations have focused on techniques for debugging concurrent programs [121,149] and on extending the operational semantics of an irreversible language with tracing [72], thereby defining the inverse semantics of the language. Hybrid approaches aim to combine reversibilisation and reversible sublanguages [172]. In general, the minimisation of the additional computational resources required for sealing information leaks by reversibilisation remains a central challenge.

4 Term Rewriting

Term rewriting [17,98,178] is a foundational theory of computing that underlies most rule-based programming languages. A *term rewriting system* (TRS) is specified as a set of rewrite rules of the form $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l. Positions are used to address the nodes of a term viewed as a tree. A *position* p in a term t is represented by a finite sequence of natural numbers, where $t|_p$ denotes the *subterm* of t at position p and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term s. *Substitutions* are mappings from variables to terms.

Given a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation satisfying the following: given terms s, t, we have $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \in \mathcal{R}$, and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. Given a binary relation $\to$, we denote by $\to^*$ its reflexive and transitive closure, i.e., $s \to^*_\mathcal{R} t$ means that s can be reduced to t in $\mathcal{R}$ in zero or more steps. The goal of term rewriting is reducing terms to so-called *normal forms*, where a term t is called *irreducible* or in *normal form* w.r.t. a TRS $\mathcal{R}$ if there is no term s with $t \to_\mathcal{R} s$. Computing normal forms can be seen as the counterpart of computing *values* in functional programming.

We also consider Conditional TRSs (CTRSs) of the form $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, with $\twoheadrightarrow$ interpreted as *reachability* ($\to^*_\mathcal{R}$). Roughly speaking, $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n \in \mathcal{R}$, and a substitution σ such that $s|_p = l\sigma$, $s_i\sigma \to^*_\mathcal{R} t_i\sigma$ for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$. Consider, e.g., the following CTRS $\mathcal{R}^{\mathsf{fn}}$:

$$\begin{aligned}
&\beta_1 : \mathsf{fn}([\,]) \to [\,] \\
&\beta_2 : \mathsf{fn}(\mathsf{person}(n,l) : xs) \to n : ys \Leftarrow \mathsf{fn}(xs) \twoheadrightarrow ys \\
&\beta_3 : \mathsf{fn}(\mathsf{city}(c) : xs) \to ys \Leftarrow \mathsf{fn}(xs) \twoheadrightarrow ys
\end{aligned}$$

where we use "$:$" and $[\,]$ as list constructors. Here, β_1, β_2 and β_3 denote labels that uniquely identify each rewrite rule. Function fn takes a list of persons of the form $\mathsf{person}(\mathit{first_name}, \mathit{last_name})$ and cities of the form $\mathsf{city}(\mathit{city_name})$ and returns a list of first names. Note that it could be specified in a typical functional language (say, Haskell) as follows:

$$\begin{array}{l} \mathsf{fn}\ [\,] = [\,] \\ \mathsf{fn}\ ((\mathsf{Person}\ n\ l) : xs) = n : ys \ \ \mathsf{where}\ ys = \mathsf{fn}\ xs \\ \mathsf{fn}\ ((\mathsf{City}\ c) : xs) = ys \ \ \mathsf{where}\ ys = \mathsf{fn}\ xs \end{array}$$

4.1 Reversible Term Rewriting

In general, term rewriting is not reversible, even for injective functions; namely, given a rewrite step $t_1 \to t_2$, we do not always have a decidable method to get t_1 from t_2. One of the first approaches to reversibility in term rewriting is due to Abramsky [2], who considered reversibility in the context of *pattern matching automata.*[1] Abramsky's approach requires a condition called *biorthogonality* (which, in particular, implies injectivity), so that the considered automata are reversible. This work can be seen as a rather fundamental delineation of the boundary between reversible and irreversible computation in logical terms. However, biorthogonality is overly restrictive in the context of term rewriting, since almost no term rewrite system is biorthogonal. Another example of a term rewrite system with both forward and reverse rewrite relations is the *reaction systems for bonding* in [159]. It has been used to model a simple catalytic reaction, polymer construction, by a scaffolding protein and a long-running transaction with a compensation.

In the context of the COST action IC1405, Nishida *et al.* [148,150] introduced the first generic notion of *reversible rewriting*, a conservative extension of term rewriting based on a so-called *Landauer embedding*. In this approach, for every rewrite step $s \to_{\mathcal{R}} t$, one should store the applied rule β, the selected position p, and a substitution σ with the values of some variables (e.g., the variables that occur in the left-hand side of a rule but not in its right-hand side). Therefore, reversible rewrite steps have now the form $\langle s, \pi\rangle \rightharpoonup \langle t, \beta(p,\sigma) : \pi\rangle$, where $\rightharpoonup$ is a reversible (forward) rewrite relation and π is a *trace* that stores the sequence of terms of the form $\beta(p,\sigma)$. The dual, inverse relation $\leftharpoondown$ is also introduced, so that its union $\rightleftharpoons$ can be used to perform both forward and backward reductions.

Moreover, [148] also introduces a scheme to *compile* the reversible extension of rewriting into the system rules. Essentially, given a system $\mathcal{R}$, new systems $\mathcal{R}_f$ and $\mathcal{R}_b$ are produced, so that standard rewriting in $\mathcal{R}_f$, i.e., $\to_{\mathcal{R}_f}$, coincides with the forward reversible extension $\rightharpoonup_{\mathcal{R}}$ in the original system, and analogously $\to_{\mathcal{R}_b}$ is equivalent to $\leftharpoondown_{\mathcal{R}}$. Therefore, $\mathcal{R}_f$ can be seen as an *injectivisation* of $\mathcal{R}$, and $\mathcal{R}_b$ can be seen as the *inversion* of $\mathcal{R}_f$.

[1] Although he did not consider rewriting explicitly, pattern matching automata can also be represented in terms of standard notions of term rewriting.

For instance, the injectivisation $\mathcal{R}_f^{\mathsf{fn}}$ of the previous CTRS $\mathcal{R}^{\mathsf{fn}}$ is as follows:

$$\begin{array}{l}\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1 \rangle \\ \mathsf{fn}^{\mathsf{i}}(\mathsf{person}(n,l)\!:\!xs) \to \langle n\!:\!ys, \beta_2(l, ws)\rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, ws\rangle \\ \mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c)\!:\!xs) \to \langle ys, \beta_3(c, ws)\rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, ws\rangle\end{array}$$

together with the corresponding inversion $\mathcal{R}_b^{\mathsf{fn}}$:

$$\begin{array}{l}\mathsf{fn}^{-1}([\,], \beta_1) \to [\,] \\ \mathsf{fn}^{-1}(n\!:\!ys, \beta_2(l, ws)) \to \mathsf{person}(n,l)\!:\!xs \Leftarrow \mathsf{fn}^{-1}(ys, ws) \twoheadrightarrow xs \\ \mathsf{fn}^{-1}(ys, \beta_3(c, ws)) \to \mathsf{city}(c)\!:\!xs \Leftarrow \mathsf{fn}^{-1}(ys, ws) \twoheadrightarrow xs\end{array}$$

For example, the following rewrite derivation in $\mathcal{R}^{\mathsf{fn}}$:

$$\mathsf{fn}([\mathsf{person}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{person}(\mathsf{ada}, \mathsf{lovelace})]) \to^* \; [\mathsf{john}, \mathsf{ada}]$$

is now as follows in $\mathcal{R}_f^{\mathsf{fn}}$:

$$\begin{array}{l}\mathsf{fn}^{\mathsf{i}}([\mathsf{person}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{person}(\mathsf{ada}, \mathsf{lovelace})]) \\ \qquad\qquad \to^* \langle [\mathsf{john}, \mathsf{ada}], \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1)))\rangle\end{array}$$

where $\beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1)))$ is the trace of the computation. Besides proving some fundamental properties of reversible rewriting, Nishida et al. [150] have developed a prototype implementation of the reversibilisation transformations (injectivisation and inversion), which is publicly available through a web interface from http://kaz.dsic.upv.es/rev-rewriting.html.

4.2 Application to Bidirectional Transformations

The framework of *bidirectional transformations* considers two representations of some data and the functions that convert one representation into the other and vice versa (see, e.g., [75] for an overview). Typically, we have a function called "get" that takes a *source* and returns a *view*. In turn, the function "put" takes a possibly updated view (together with the original source) and returns the corresponding, updated source. In this context, *bidirectionalisation* [128] aims at automatically producing one of the functions, typically producing a function put from the corresponding function get. For this purpose, a so-called *complement* function is often introduced so that get becomes injective (see, e.g., [55]).

In [152], Nishida and Vidal present a bidirectionalisation technique based on the injectivisation and inversion transformations of CTRSs from [150]. They also prove a number of relevant properties which ensure that changes in both the source and the view are correctly propagated and that no undesirable side-effects are introduced.

To be precise, given a get function f, the corresponding put can be automatically defined as follows:

$$\mathsf{put}_{\mathsf{f}}(v, s) \to s' \Leftarrow \mathsf{f}^{\mathsf{i}}(s) \twoheadrightarrow \langle v', \pi\rangle, \mathsf{f}^{-1}(v, \pi) \twoheadrightarrow s'$$

Note that the trace of a computation, π, plays the role of a *complement* (following the terminology in the literature of bidirectional transformations).

For instance, given the previous function fn, the corresponding put function is defined as follows:

$$\mathsf{put_{fn}}(v, s) \rightarrow s' \Leftarrow \mathsf{fn^i}(s) \twoheadrightarrow \langle v', \pi \rangle, \mathsf{fn}^{-1}(v, \pi) \twoheadrightarrow s'$$

so that, e.g., $\mathsf{put_{fn}}([\mathsf{peter}, \mathsf{ada}], \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1))))$ reduces to $[\mathsf{person}(\mathsf{peter}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{person}(\mathsf{ada}, \mathsf{lovelace})]$. Note that the first element has been updated from $\mathsf{person}(\mathsf{john}, \mathsf{smith})$ to $\mathsf{person}(\mathsf{peter}, \mathsf{smith})$.

However, $\mathsf{put_f}$ is only defined for "compatible" view updates. E.g., the function $\mathsf{put_{fn}}([\mathsf{ada}], \beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1))))$ cannot be reduced to a value. In [152], the use of *narrowing* [76,176]—an extension of rewriting that replaces matching with unification—is introduced to precisely characterise *compatible* (also called *in-place*) view updates.

For example, given the trace $\beta_2(\mathsf{smith}, \beta_3(\mathsf{london}, \beta_2(\mathsf{lovelace}, \beta_1)))$, narrowing allows us to compute the *view skeleton* $[x_1, x_2]$. This means that any view update that can be obtained as an instance of $[x_1, x_2]$ is compatible with the trace (and, thus, the put function is well defined).

Finally, [152] also discusses some directions for dealing with view updates that are not compatible.

5 Membrane Computing

Natural computing is a complex field of research dealing with models and computational techniques inspired by nature that helps us in understanding the biochemical world in terms of information processing. Membrane computing [154] and reaction systems [53] are two important theories of natural computing inspired by the functioning of living cells.

Membrane computing deals with multisets of symbols processed in the compartments of a membrane structure according to some multiset rewriting rules; some of the symbols (presented with their multiplicity within the regions delimited by membranes) evolve in parallel according to the rules associated with their membranes, while the others remain unchanged and can be used in the subsequent steps. It is also possible to send multisets of symbols in the neighbouring membranes, the systems being organised in a tree-like fashion. The evolution takes place in a maximal parallel manner: all the instances of the applicable rules have to be applied in order to reach the next state.

The situation is different in reaction systems. These systems represent a qualitative model: they deal with sets rather than multisets. Two major assumptions distinguish the reaction systems from the membrane systems: (i) threshold assumption: reaction systems have actually an infinite multiplicity for their resources; (ii) no permanency assumption: only entities produced at one step will be present in the system at the next step.

The issue of reversibility in various computational paradigms has gained interest in recent years. In one of the earliest papers on reversibility in membrane systems [5], the authors (under the influence of category theory) presented

reversibility as a form of duality. A full description of this kind of reversibility in membrane systems is given by Agrigoroaiei and Ciobanu in [6].

In [7], Aman and Ciobanu investigated the reversibility of biochemical reactions in parallel rewriting systems; these systems can easily represent some classes of membrane systems and Petri nets. Formally, a parallel rewriting system is a tuple $(O, \mathcal{R}, w_0)$, where O is a finite alphabet of objects, $\mathcal{R}$ is a set of rewriting rules and w_0 is a multiset of objects over O. For each rule $r \in \mathcal{R}$ there exist the non-empty multisets $lhs(r), rhs(r) \in O^+$ standing for the left-hand side and right-hand side of the rule, respectively, such that $r : lhs(r) \to rhs(r)$. Given a multiset of rules F, then the left-hand side and right-hand side of it can be defined as: $lhs(F) = \sum_{r \in \mathcal{R}} F(r) \cdot lhs(r)$ and $rhs(F) = \sum_{r \in \mathcal{R}} F(r) \cdot rhs(r)$.

A parallel rewriting system $(O, \mathcal{R}, w_0)$ evolves in a maximal parallel manner. This means that a non-empty multiset R of rules is applicable to a multiset w of objects if $lhs(R) \leq w$ and there does not exist $r \in \mathcal{R}$ such that $lhs(r) \leq w - lhs(R)$. By applying a multiset R of rules, a multiset w of objects is transformed into another multiset $w' = w - lhs(R) + rhs(R)$ of objects. If no multiset of rules is applicable, then the computation stops.

The new features of this approach are given by adding an external control specified by using a special symbol $\rho \notin O$ that informs the system that a rollback will be executed, and by constructing two new sets of rules $\overrightarrow{\mathcal{R}} = \{u \to v|_{\neg\rho} \mid u \to v \in \mathcal{R}\}$ and $\overleftarrow{\mathcal{R}}_\rho = \{v \to u|_\rho \mid u \to v \in \mathcal{R}\} \cup \rho \to \lambda$ to mark the rules that will be applied in forward and backward steps, respectively.

Several theoretical results are obtained, including the so-called loop results and the connections between the evolutions of these systems and their reversible extensions. If there exist multisets of rules not competing for the same resources, then the following results hold.

A first result presents the **forward diamond** property:
If $w \xrightarrow{\overrightarrow{\mathcal{R}}} w'$ *and* $w \xrightarrow{\overrightarrow{\mathcal{R}'}} w''$, *where* $\overrightarrow{\mathcal{R}}$ *and* $\overrightarrow{\mathcal{R}'}$ *are two valid multisets of rules such that* $lhs(\overrightarrow{\mathcal{R}}) \cap lhs(\overrightarrow{\mathcal{R}'}) = \emptyset$, *then there exists a multiset* w_1 *such that* $w' \xrightarrow{\overrightarrow{\mathcal{R}'}} w_1$ *and* $w'' \xrightarrow{\overrightarrow{\mathcal{R}}} w_1$.

The second result presents the **reverse diamond** property:
If $w \xrightarrow{\overleftarrow{\mathcal{R}}_\rho} w'$ *and* $w \xrightarrow{\overleftarrow{\mathcal{R}'}_\rho} w''$, *where* $\overleftarrow{\mathcal{R}}_\rho$ *and* $\overleftarrow{\mathcal{R}'}_\rho$ *are two valid multisets of rules such that* $lhs(\overleftarrow{\mathcal{R}}_\rho) \cap lhs(\overleftarrow{\mathcal{R}'}_\rho) = \emptyset$, *then there exists a multiset* w_1 *such that* $w' \xrightarrow{\overleftarrow{\mathcal{R}'}_\rho} w_1$ *and* $w'' \xrightarrow{\overleftarrow{\mathcal{R}}_\rho} w_1$.

A forward step performed using the multiset $\overrightarrow{\mathcal{R}}$ of rules can be matched by a backward step performed using the multiset $\overleftarrow{\mathcal{R}}_\rho$ of rules, and vice-versa (**loop**):

$$w \xrightarrow{\overrightarrow{\mathcal{R}}} w' \quad \text{if and only if} \quad \rho w' \xrightarrow{\overleftarrow{\mathcal{R}}_\rho} w.$$

In [8], Aman and Ciobanu investigated reversibility in reaction systems. Reaction systems [53] deal with sets rather than multisets, assuming that each resource is present in the system in a sufficient amount to ensure that several reactions needing such a resource are not in conflict. Formally, a reaction system $\mathcal{A}$ is a tuple (S, A), where S is a finite alphabet and $A \subseteq rac(S)$. The set $rac(S) = \{(R, I, P) \mid R, I, P \subseteq S, R \cap I = \emptyset\}$ is the set of all reactions over S. Given a reaction $a = (R_a, I_a, P_a)$, the sets R_a, I_a and P_a contain the reactants, inhibitors and products of a, respectively. For a set $C \subseteq S$ and a set of reactions $A \subseteq rac(S)$, the result of applying A on C is defined by $res(A, C) = \bigcup_{a \in A} P_a$, and the evolution can be written as $C \xrightarrow{A} res(A, C)$. The set of all reactions from A that are enabled by C is $en(A, C) = \{a \in A \mid R_a \subseteq C, I_a \cap C = \emptyset\}$.

An interactive process is a pair $\pi = (\gamma, \delta)$ such that $\gamma = C_0, \ldots, C_{n-1}$, $\delta = D_1, \ldots, D_n$ with $n \geq 1$, where C_{j-1}, $D_j \subseteq S$ for $1 \leq j \leq n$ are the context and result sets, respectively. The sets D_j are computed using the equalities $D_1 = res(A, W_0)$ and $D_i = res(A, W_{i-1})$, where the sets $W_0 = C_0$ and $W_i = D_i \cup C_i$ for each $2 \leq i \leq n$ represent the states.

In order to have backward computations, we add to each state W_i a register T_i to remember objects no longer available after step i. The reverse of a set A of reactions is the set $\tilde{A} = \{(P_a, I_a, R_a) \mid (R_a, I_a, P_a) \in A\}$. If $\rho \notin W_i$ and $E_i \neq \emptyset$, then a forward computation $(W_i, T_i) \xrightarrow{E_i} (W_{i+1}, T_{i+1})$ takes place, where $T_{i+1} = inc(T_i) \cup \bigcup_{t \in W_i \setminus lhs(E_i)} (t, 0)$, $inc(T) = \bigcup_{(t,i) \in T} (t, i{+}1)$ and $W_{i+1} = res(E_i, W_i)$. However, if $\rho \in W_i$ and $\tilde{E}_i \neq \emptyset$, then a backward computation $(W_{i+1}, T_{i+1}) \overset{\tilde{E}_i}{\rightsquigarrow} (W_i, T_i)$ takes place, where $T_i = dec(T_{i+1})$, $dec(T_d) = \bigcup_{(t,i) \in T_d; i > 0} (t, i-1)$ and $W_i = res(\tilde{E}_i, W_{i+1}) \cup zero(T_{i+1})$ with $zero(T) = \bigcup_{(t,0) \in T} t$.

If the states satisfy some preconditions, then backward reductions are the inverse of the forward ones, and vice-versa:

- If $W = res(\tilde{E}, W') \cup zero(T')$ and $\rho \in W'$, then

$$(W, T) \xrightarrow{E} (W', T') \text{ implies } (W', T') \overset{\tilde{E}}{\rightsquigarrow} (W, T).$$

- If $W' = res(E, W)$ and $\rho \notin W$, then

$$(W', T') \overset{\tilde{E}}{\rightsquigarrow} (W, T) \text{ implies } (W, T) \xrightarrow{E} (W', T').$$

An operational correspondence between reaction systems and rewriting theory is also proved. It allows a translation of the reversible reaction systems into some rewriting systems executable in the rewriting engine Maude [39].

In [163] Pinna pursues reversibility in membrane systems from a different perspective. The paper focuses on how to reverse steps in computations of membrane systems, without adding rules to represent the reverse application of the original rules. Just one assumption on rules is made, namely that rules are not allowed to rewrite a multiset of objects into an empty multiset: the application of a rule must have an effect, though this could be not observable. This requirement is driven by the necessity that, in order to reversely apply a rule, this one

must produce something. Furthermore, as in most rewriting systems, also in the considered membrane systems a computation step does not register the (multiset of) rules applied. Since this information may be crucial to reversely apply the same (multiset of) rules, one needs some strategies to solve the issue and obtain reversibility.

A solution can be to enrich each object with the information on how the particular object has been produced, namely each object now may carry the name of the rule r used to produce it. Objects are then $O \times R \cup \{\bot\}$ where R is the set of rules $\bigcup_i R_i$, with i ranging over the membranes, and $\bot$ denotes that the object is present in the initial configuration. The unique assumption is that rule names are unique. The drawback of this solution is that once an object is used the information on how it has been produced is lost.

To overcome this problem, the proposed solution is to add to the notion of configuration, previously a vector of multiset of objects, with one element for each membrane, a memory organised as a labelled partial order. Each element of the partial order corresponds to an object and carries also the information on which rule produced it. According to this a memory m is a triple $(X, \preceq, l)$ where $\preceq$ is a partial order and $l : X \rightarrow O \times R \cup \{\bot\} \times \{1, \ldots, n\}$ is the labelling associating the object, the name of rule that produced it and the membrane where the object is allocated. A configuration of a membrane system with n membranes is then the pair $\mathcal{C} = (C, \mathsf{m})$, where $C = (w_1, \ldots, w_n)$ is the tuple of multisets over objects O and $\mathsf{m} = (X, \preceq, l)$ is a memory such that for each $i \in \{1, \ldots, n\}$ it holds that $w_i = \mathsf{obj}_i(max(\mathsf{m}))$, where max gives the multiset of maximal elements of the memory and obj_i forgets the information about the rule.

The effect of applying a vector of multisets of rules $\mathcal{R}$ does not consist only in updating suitably the multisets of objects forming a configuration in the classical sense, but also in adding the information on which rule produced a specific object in the memory. This will be denoted with $(C, \mathsf{m})\{\![\mathcal{R}> (C', \mathsf{m}')$ where $C \stackrel{\mathcal{R}}{\Longrightarrow} C'$ is the usual step in membrane systems computation and the new memory m' is obtained adding to m the objects produced by the rules in $\mathcal{R}$ and by updating the partial order so that the produced elements are greater than the ones consumed by these rules.

Then the reverse application of a vector of multisets of rules can be obtained by looking in this memory for the maximal elements, which correspond to the right-hand sides of the rules to be reversely applied. The proper configuration is then computed from the new memory obtained by removing the maximal elements. The reverse application of a vector of multisets of rules $\mathcal{R}$ is denoted with $(C, \mathsf{m}) <\mathcal{R}]\!\}(C', \mathsf{m}')$, where the maximal elements of m' corresponding to the right-hand sides of rules in $\mathcal{R}$ are removed obtaining a memory m and a configuration C where each element $w_i = \mathsf{obj}_i(max(\mathsf{m}))$.

The following result has been proved:

Let Π_m be a membrane system with memory, (C, m) a configuration, and $\mathcal{R}$ be a vector of multisets of rules such that $(C, \mathsf{m})\{\![\mathcal{R} > (C', \mathsf{m}')$. Then, for all multi-rule vectors $\mathcal{R}'$ such that $(C', \mathsf{m}') <\mathcal{R}]\!\}(C, \mathsf{m})$, it holds that $\mathcal{R}' = \mathcal{R}$.

This simple implementation has the advantage of properly realising the causal reversibility. Furthermore the memory allows also to capture the dependencies among objects in a membrane system computation.

6 Process Calculi

Process calculi are a class of algebraic models for concurrent and distributed systems. Process calculi allow one to express the behaviour of a concurrent system in a concise way, abstracting away from implementation details, and focusing on the interaction patterns among the components of the system. Thus, it is possible to express the behaviour of a system in a mathematically precise way and verification techniques can be easily developed on top of it.

Research on reversing process calculi can be perhaps tracked back to the Chemical Abstract Machine [30], a calculus inspired by chemical reactions whose operational semantics defines both forward and reverse reduction relations. The first attempts to reverse existing process calculi can be found in [44,46], where a reversible extension of CCS [140] was presented. A main contribution of [44] was the definition of the notion of *causal-consistent* reversibility: any action can be undone, provided that its consequences, if any, are undone first. This definition is tailored to concurrent systems, where actions may overlap in time, hence saying "undo the last action" is not meaningful. Notably, this definition relates reversibility to causality instead of time, thus it can be applied even in those settings, such as some distributed systems, where no unique notion of time exists. A survey on causal-consistent reversibility can be found in [120].

6.1 Reversing Process Calculi

Following [44], causal-consistent extensions of other and more expressive process calculi have been defined. They can be divided into two families, one dealing with calculi equipped with labelled transition system semantics (describing interactions between the process and the outside world), and one dealing with reduction semantics (describing the evolution of processes in isolation). The former is more general, while the latter is normally simpler and hence more easily applicable to expressive calculi. The first approach extended causal-consistent reversibility from CCS to any calculus defined using a specific SOS format (a subset of the *path* format [146]) [160,161], and to π-calculus [42]. In the second line of research we find extensions of a fragment of CCS with biological relevance [35,36], of the higher-order π-calculus [117,119], of the coordination language Klaim [56], of a π-calculus with sessions [179], and of a CCS with broadcast communications [133]. The instance of the framework in [160] on CCS is called CCSK. CCSK differs from the reversible CCS in [44] in the way history is kept. Indeed, the approach of [160] can be considered static, since the structure of processes does not change during computation, and the minimal history information needed to enable reversibility is kept in the processes themselves, while in [44] the process is consumed during execution (as standard in process calculi) and larger

memories are added to store history information. Nonetheless the two methods are equivalent as hinted at by [130] and fully proved by [115], where a mapping from an instance on CCS of [160] to the reversible CCS of [44] and vice versa is presented.

As discussed above, causally-consistent reversibility relates reversibility with causality. In CCS just one main notion of causality exists, and both the reversible variants of CCS above are based on it. In the π-calculus, many relevant notions of causality exist, which differ in the treatment of parallel extrusions of the same name. In [131] a uniform framework to define reversible π-calculi is presented. The framework is parametric w.r.t. a data structure that stores information about extrusions of a name. Different data structures yield different approaches to the parallel extrusion problem, leading to different ways of reversing a name extrusion, thus giving rise to different reversible variants of the π-calculus.

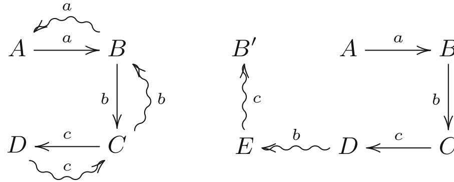


Fig. 1. Example of causal-consistent (left) and out-of-causal order reversibility (right)

6.2 Controlled Reversibility

The line of research described above focused on uncontrolled reversibility, defining how to reverse a process execution (in particular, which history and causal information is needed, and how to manage it), but not specifying when and whether to prefer backward execution over forward execution or vice versa. Uncontrolled reversibility allows one to understand how reversibility works, but not to exploit it into applications. Indeed, different application areas need different mechanisms to control reversibility. For instance, in biological systems the direction of the computation depends on physical conditions such as temperature and pressure, while in reliable systems reversibility is used to recover a consistent state when a bad event occurs. Triggered by these needs different mechanisms for controlling reversibility have been proposed (see the categorisation in [118]). For instance, [45,179] introduced irreversible actions to avoid going backward after a relevant result has been computed. Instead, [56,57,114,116,118,126] proposed an explicit rollback operator undoing a past action inside calculi where normal computation is forward, and a mechanism of alternatives allowing one to avoid trying the same path again and again. As shown in [57], the rollback operator satisfies a simple intuitive specification, namely that it is the smallest causal-consistent sequence of backward moves undoing the target action. Also, [18] let an energy potential drive the direction of computation while [158] introduced a forward monitor controlling the direction of execution of a reversible monitored

process. A process calculus with a prefixing operator to model locally-controlled reversibility is introduced in [102,103]. Actions can be undone spontaneously, as in other reversible process calculi, or as pairs of concerted actions, where performing a weak action forces the undoing of a past action. Concerted actions allow one to model out-of-causal order computation, where effects can be undone before their causes, which is forbidden in most other reversible calculi. This form of reversibility is common in biochemical reactions, e.g., in the hydration of formaldehyde in water into methanediol. Such a feature can be disabled by considering a reduced form of concerted actions.

Reversibility, both in causal order and out-of-causal order, can be modelled in reversible event structures [157].

Figure 1 shows the difference between causal-consistent (left) and out-of-causal order reversibility. In both cases, the system performs actions a, b and c to reach state D. On the left, in order to get back to the original state, one has to first undo (in Fig. 1 undoing is represented with squiggly arrows) c then b and finally a. On the right, since causes do not need to be respected, the system can undo b before c, reaching in this way a new state E which may not have been reachable from the initial configuration by just using forward steps. From there, a and c may or may not be undoable. In the example, only c can be undone, leading to B'. If undoing b and undoing c do commute, then $B = B'$.

6.3 Analysis Techniques

Despite the proliferation of calculi for reversibility, when the COST Action IC1405 started, analysis techniques for reversible calculi were very limited, consisting essentially in some limited analysis about behavioural equivalences (in particular, forward-reverse bisimilarity [161]) and a technique for causal compression in CCS with irreversible actions [101]. Thus, the work in the COST Action tackled analysis techniques in depth, considering behavioural equivalences, contracts [77] and session types [77].

Behavioural Equivalences. Understanding which notions of behavioural equivalences are suitable for reversible process calculi is a non-trivial, and still open, problem.

As shown in [119], notions of weak bisimilarity that do not distinguish forward actions from backward actions are very coarse, while notions of strong bisimilarity distinguishing them, such as forward-reverse bisimilarity [161], are very fine-grained, hence other notions are worth exploring.

In [135] Mezzina and Koutavas studied testing preorders, and in particular a safety one and a liveness one, in a reversible CCS where reductions are totally ordered and rollbacks lead systems to past states. Liveness and safety in this setting correspond to the should-testing [166] and inverse may-testing preorders [50] for the underlying forward calculus, respectively. In general, one would expect the models of these preorders to be based on both forward and backward transitions, thus offering complex proof techniques for verification. Instead, in [135] full abstraction of liveness and safety is based only on forward

transitions and limited rollback points, giving rise to considerably simpler proof techniques. Moreover, total reversibility allows one to make finer observations w.r.t. liveness, but not w.r.t. safety.

Contracts. (Binary) contracts are a behavioural model [77] to study the interactions between a client and a server. The first investigation of contracts in a reversible setting appeared in [21,22]. There, both the client and the server could rollback to a previous checkpoint at any moment. The main result was that the compliance relation, ensuring that the client and the server can successfully interact, and the sub-behaviour relation, are both decidable, and they remain so also when the possibility of skipping some messages is added.

In retractable contracts [23,24] the client and the server can both get back to previous decision points and take alternative paths only when the interaction is stuck. The main results in [23,24] are that retractable contracts are a conservative extension of contracts, both compliance and the subcontract relation are decidable in polynomial time, and the dual of a contract always exists and has a simple syntactic characterisation. Furthermore, retractable contracts are equivalent to a novel model of contracts featuring a speculative choice: all the options of the choice are explored concurrently, and the computation succeeds if at least one of the options is successful. In [20], a three-party game-theoretic interpretation of retractable session contracts [23] has been proposed. In such an interpretation a client is compliant with a server if and only if there exists a winning strategy for a particular player in a game-theoretic model of contracts. Such a player can be looked at as a mediator, driving the choices in the retractable points.

Session Types. Session types [77] are one of the formalisms that have been proposed to structure interaction and reason over communicating processes and their behaviour. In a series of works [136–138] reversible monitored semantics for binary [136,138] and multiparty [137] session types is investigated. The novelty of the approach is that monitors are derived by types, and they store all the needed information to bring the system back to previous states. This implies that processes of the system are oblivious to reversibility, as they do not store any information about past computations. A deeper discussion on session types and reversibility can be found in [134].

7 Petri Nets

Petri nets [165] are a mathematical formalism for modelling and reasoning on concurrent systems. In most of the cases, Petri nets are four-tuples containing two finite sets, of active (actions/transitions) and static (places) elements, which are connected by a flow function (or relation) with initial state given by tokens scattered on places. In what follows, by Petri net we mean its most common variant, called place-transition net.

Petri nets support both action-based and state-based approaches (via reachability graphs which are equivalent to transition systems). Reversibility in Petri

nets was always an important notion, however its meaning changed in time. At first, in the seventies, the notion of reversibility referred to nets where each transition has its inverse [54]. Such a notion of local reversibility is very close to the one currently used in other fields, like programming languages or process calculi. This notion of reversible nets (also called symmetric nets [54]) is still occasionally used to define the inverse net [33]. The time complexity of some decision problems in bounded symmetric Petri nets is lower than in the general case of bounded nets. The other meaning of reversibility in Petri nets, also called cyclicity [33], takes a global approach and requires the initial state of the net to be reachable from any other reachable state [147]. Petri nets are called symmetric also in other situations than the described local notion of reversibility [41].

During the four years of the COST Action IC1045, "Reversible Computation - Extending Horizons of Computing", the notion of local reversibility was investigated. One can divide the proposed contributions into three main threads: two of them consider how to reverse a single transition in a Petri net, allowing one to use, respectively, a single reverse transition or a set of reverses. The last thread focuses on modelling reversible semantics in specific models based on Petri nets.

An approach to invert a single transition using a single (strict) reverse was investigated under both the sequential semantics and the true concurrent semantics. The case of sequential semantics was considered in [28]. The strict reverse is added to the net as a fresh transition with arcs copied from the original one, but with the opposite direction. The problem of checking whether the set of reachable markings in a net changes, when a strict reverse for a single transition is added, was proven to be undecidable. The opposite result was shown for the set of all coverable markings. Another important fact shown in [28] is related to cyclicity: introducing a strict reverse in a cyclic net may change the set of reachable markings.

The above problem of checking whether the set of reachable markings in a net changes by adding a strict reverse for a single transition becomes decidable for the bounded nets. Therefore, one can ask a more general question - is it possible to reverse the specified transition while only requiring the resulting net and the given one to have isomorphic behaviour (i.e., isomorphic reachability graph), but allowing one to change the structure of the net? The question has been answered by using well-known techniques from region theory [19]. There are transition systems which are reachability graphs of a bounded Petri net where transitions cannot be inverted by strict reverses, but one can easily combine separate solutions for different transitions to solve the problem [26]. Even in the special case of linear transition systems over binary sets of actions the transitions cannot be always inverted by strict reverses. In such systems, the time complexity of the problem of checking whether the set of reachable markings changes by adding a strict reverse for a single transition is linear [48]. Another special case of bounded nets are occurrence nets, that is 1-safe and acyclic nets without backward conflicts, where one can always use strict reverses. This property of occurrence nets and their infinite extensions was used as an intermediate step in [132], described later on.

Another line of research on strict reverses considers systems under concurrent semantics of action execution. In such systems one can execute more than one action at the same time, including the situation when a single action is executed multiple times (auto-concurrence). Reversing atomic transitions in such systems is discussed in [49]. In simple cases, where auto-concurrence is excluded, one can reduce reversing under the concurrent semantics to the sequential case. However, in the case of true multisets of actions executed simultaneously, one needs to allow mixed reverses (i.e., steps where both forward and backward actions are present) and true concurrent reversing can be reduced to coping with all spikes (i.e., multisets of actions with singleton support).

In a more general setting, in order to invert a single transition, one can allow to define a set of reverses with the opposite effect, called effect reverses [26]. In such a case, the problem of finding a bounded Petri net where each transition can be reversed and with isomorphic behaviour becomes always solvable [26]. Hence, some systems where inverting transitions using strict reverses was impossible become reversible in this setting. Moreover, the price to make any bounded net ready for inverting by the sets of effect reverses is not high - one needs to transform the original net into its complementary version, which doubles the size of the set of places [26].

A similar attempt for unbounded nets is presented in [139]. There are unbounded nets which cannot be inverted even using infinite sets of effect reverses for their transitions. However, if it is possible, then finite sets are enough. The problem of finding a possibly totally different net with isomorphic behaviour that can be reversed was reduced to extending the existing one by new places which do not disable any transitions in any reachable state and checking whether there exists a pair of problematic states. Those pairs of problematic states are strongly structured, with a natural partial order. The set of all minimal pairs of problematic states for a given system is finite, however, the problem of checking whether two given states form a problematic pair is not elementary, while the problem of checking whether there exists at least one such pair is undecidable [139].

A different line of research considers extensions of Petri nets with causal-consistent local reversibility [132]. Such an extension can be obtained for any place transition net by unfolding it into occurrence nets and folding them back to a coloured Petri net with an infinite number of colours. Those colours are used to encode the content of a stack used to reverse the computation. The price to be paid is that coloured Petri nets with infinitely many colours are in general Turing complete.

Another approach to investigate causal-consistent local reversibility, but also out-of-order local reversibility, is the biologically inspired model of reversing Petri nets [155]. There tokens are persistent bases connected by bonds which are relocated by transitions of the net. The greatest limitation of the approach is the requirement of finiteness and acyclicity of the net modelled in this way. On the other hand, one can encode reversing Petri nets into coloured Petri nets with a finite number of colours [27], hence also into classical bounded

place-transition systems. Moreover, reversing Petri nets were successfully applied to the distributed antenna selection problem [156].

Petri net theory has been deeply studied. Cyclic and symmetric systems play quite an important role, however the issue of equipping concurrent systems with reversing mechanisms was not explored. The research conducted as a part of the COST Action IC1405 "Reversible Computation - Extending Horizons of Computing" enriched the theory of Petri nets by exploring some approaches to reverse transitions in existing systems. Although the effect of adding reverses of the actions to the existing system is in general difficult to evaluate (the problem of behaviour preservation is undecidable for place-transition nets), the problem can be solved if one allows unbounded stacks (coloured Petri nets approach) or restricts oneself to bounded models.

8 Automata

Automata theory studies abstract machines, or automata, as mathematical models of computation. They help in understanding limits of computation and the role of various resources – such as time and space – on the computational power. Examples of widely studied classes of automata include finite automata (bounded memory), pushdown automata (infinite memory organised as a stack), counter machine (infinite memory organised as counters), Turing machines (infinite memory tape) and cellular automata (massively parallel regular network of finite automata). These come in several flavours and variations, e.g., with respect to determinism. An automaton is reversible if it preserves information so that its computation can be retraced back in time. All the automata classes above can support reversibility. See [105,143] for details on computation by various models of reversible automata.

8.1 Finite Automata

Reversibility in finite automata has been widely investigated, e.g., [9,162]. The class of languages having a reversible one-way automaton is a proper subclass of the regular one. However, different models have been considered, depending on whether automata are required to have only one initial state and/or only one final state. Languages not having any reversible classical automaton have been characterised in terms of a forbidden pattern in the minimum automaton [73]. In the same paper, an NL-complete method to decide whether the language accepted by a given deterministic finite automaton can also be accepted by some reversible deterministic finite automaton has been derived.

In case the language accepted by a deterministic finite automaton is reversible, the size of the smallest reversible automaton may be exponential with respect to the size of the minimal irreversible one [73]. Recently analyses about the descriptional complexity of reversible deterministic finite automata provided some techniques to simulate these devices in an efficient way [123,125]. Indeed, though converting a deterministic automaton into a reversible one may require

an exponential increase in size, the proposed representation allows to limit this cost by concisely representing the reversible automaton rather than explicitly writing down its description.

Based on the forbidden pattern approach, the degree of irreversibility for a regular language has been studied [13]. The degree is defined to be the minimal number of such forbidden patterns necessary in any deterministic finite automaton accepting the language. It is shown that the degree induces a strict infinite hierarchy of language families. The behaviour of the degree of irreversibility under the usual language operations union, intersection, complement, concatenation, and Kleene star, has been studied, showing tight bounds (some asymptotic) on the degree.

Because of the narrowness of the power of reversible finite automata with respect to the irreversible ones, the definition of reversibility has been relaxed, by considering finite automata whose computations can be reversed, at any point, by accessing the last k symbols read from the input, for a fixed k. These devices are said to be "weakly irreversible". Characterisations of languages accepted by weakly irreversible automata and languages not having any weakly irreversible automaton ("strongly irreversible" languages) have been given [124].

Another treatment of a relaxed definition of reversibility concerns nondeterminism. It turned out that reversible nondeterministic finite automata are more powerful compared to their reversible deterministic counterparts, but still cannot accept all regular languages [74]. The two notions of relaxed reversibility have been compared and closure properties of the language family induced by these devices have been derived.

8.2 Pushdown Automata

Reversible classical pushdown automata have been introduced in [107]. Their computational capacity turned out to lie properly in between the regular and deterministic context-free languages. In the same paper, it is shown that a deterministic context-free language cannot be accepted reversibly if more than real-time is necessary for acceptance. Closure properties as well as decidability questions for reversible pushdown automata are studied and it is shown that the problem to decide whether a given nondeterministic or deterministic pushdown automaton is reversible is P-complete, whereas it is undecidable whether the language accepted by a given nondeterministic pushdown automaton is reversible.

One extension of finite automata in order to enlarge the underlying language class as well as to preserve many positive closure properties and decidable questions is represented by input-driven pushdown automata. Such automata share many desirable properties with finite automata, but still are powerful enough to describe important non-regular behaviour. Basically, for such devices the operations on the pushdown store are determined by the input symbols. With respect to reversibility they have been studied in [110]. So, the sub-family of the context-free languages that share the two important properties of being accepted by an input-driven pushdown automaton as well as of being accepted by a reversible pushdown automaton are considered. This intersection can be defined on the

underlying language families or on the underlying machine classes. It turned out that the latter class is properly included in the former. The relationships between the language families obtained in this way and to reversible context-free languages as well as to input-driven languages are studied. In general, a hierarchical inclusion structure within the real-time deterministic context-free languages is obtained. Finally, the closure properties of these families under the standard operations are investigated and it turned out that all language families introduced are anti-AFLs (that is, they are not closed under any of the operations required to be an Abstract Family of Languages).

Since reversible finite automata do not accept all regular languages and reversible pushdown automata do not accept all deterministic context-free languages, it is of significant interest both from a practical and theoretical point of view to close these gaps. Therefore these reversible models have been extended by a preprocessing unit which is basically a reversible injective and length-preserving sequential transducer [16]. It turned out that preprocessing the input using such weak devices increases the computational power of reversible deterministic finite automata to the acceptance of all regular languages. On the other hand, for reversible pushdown automata the accepted family of languages lies strictly in between the reversible deterministic context-free languages and the real-time deterministic context-free languages. Moreover, it has been derived that the computational power of both types of machines is not changed by allowing the preprocessing sequential transducer to work irreversibly.

Two-pushdown automata where the input is placed in one pushdown and that perform computations by inspecting and rewriting words at the top of the pushdowns are of particular interest as the deterministic variant is known to characterise the class of Church-Rosser languages when the rewriting is length-reducing. Such reversible two-pushdown automata are studied in [14]. A separation of the deterministic and reversible variants are obtained as well as the incomparability with the (deterministic) context-free languages. However, their properties of emptiness, (in)finiteness, universality, inclusion, equivalence, regularity, and context-freeness are not even semi-decidable.

8.3 Finite State and Pushdown Transducers

Computational models are not only interesting from the viewpoint of accepting some input, but also from the more applied perspective of transforming some input into some output. Transductions that are computed by different variants of transducers are studied in detail in the book of Berstel [31].

Reversibility in transducing devices has been investigated recently in [47,111] for deterministic finite state transducers. In [111], the families of transductions computed are classified with regard to three types of length-preserving transductions as well as to the property of working reversibly. It is possible to settle all inclusion relations between these families of transductions even with injective witness transductions. Furthermore, the standard closure properties and decidability questions have been investigated. It turned out that the non-closure under almost all operations can be shown, whereas all decidability questions

can be answered in polynomial time. Finally, the strict concept of reversibility is relaxed and an infinite and dense hierarchy with respect to the grade of reversibility is obtained.

Deterministic pushdown transducers have also been introduced, and analysed with respect to their ability to compute reversible transductions [66]. Now, the families of transductions computed are classified with regard to four types of length-preserving transductions as well as to the property of working reversibly. It turns out that accurate to one case separating witness transductions can be provided. For the remaining case it is possible to establish the equivalence of both families by proving that stationary moves can always be removed in length-preserving reversible pushdown transductions.

8.4 Queue Automata and Limited Automata

A further natural and well-studied extension of finite automata are queue automata, where the extension is by a storage media of type queue. Their reversible variant has been studied in [109]. In contrast to, for example, finite or pushdown automata, it has been shown that any queue automaton can be simulated by a reversible one. So, reversible queue automata are as powerful as Turing machines. Therefore it is of interest to impose time restrictions on queue automata. Quasi real-time and real-time computations have been considered. It has been shown that every reversible quasi real-time queue automaton can be sped up to real-time. On the other hand, under real-time conditions reversible queue automata are less powerful than general queue automata. Furthermore, a lower bound of $\Omega\left(\frac{n^2}{\log(n)}\right)$ time steps for real-time queue automata witness languages to be accepted by any equivalent reversible queue automaton has been exhibited. The closure properties of reversible real-time queue automata are similar as for reversible deterministic pushdown automata. Moreover, all commonly studied decidability questions such as emptiness, finiteness, or equivalence are not semi-decidable for reversible real-time queue automata. Furthermore, it is not semi-decidable whether an arbitrary given real-time queue automaton is reversible.

A k-limited automaton is a linear bounded automaton that may rewrite each tape square only in the first k visits, where $k \geq 0$ is a fixed constant. It is known that these automata accept context-free languages only. The deterministic k-limited automata have been investigated towards their ability to perform reversible computations [112]. It turned out that, for all $k \geq 0$, sweeping k-limited automata accept regular languages only. In contrast to reversible finite automata, all regular languages are accepted by sweeping 0-limited automata. Then the computational power gained in the number k of possible rewrite operations has been studied. It has been shown that the reversible 2-limited automata accept regular languages only and, thus, are strictly weaker than general 2-limited automata. Furthermore, a proper inclusion between reversible 3-limited and 4-limited automata languages has been obtained. The next levels of the hierarchy are separated between every k and $k + 3$ rewrite operations. Finally,

it turned out that all k-limited automata accept Church-Rosser languages only, that is, the intersection between context-free and Church-Rosser languages contains an infinite hierarchy of language families beyond the deterministic context-free languages.

8.5 Cellular Automata

A cellular automaton (CA) is a dynamical system on an infinite grid of cells defined by a local update rule that is applied simultaneously at all cells. More precisely, in the usual rectilinear d-dimensional setting the cells are the elements of $\mathbb{Z}^d$ and each cell stores an element of a finite state set A. The dynamics is specified by a finite neighbourhood $D \subseteq \mathbb{Z}^d$ that gives the relative offsets to neighbours of cells, and a local rule $f : A^D \longrightarrow A$ that gives the new state of a cell based on the previous states in its neighbourhood. A configuration $c : \mathbb{Z}^d \longrightarrow A$, specifying the global state of the system, changes in a single time unit to become the new configuration c' with $c'(\vec{n}) = f(\sigma^{\vec{n}}(c)_{|D})$ for every cell $\vec{n} \in \mathbb{Z}^d$, where $\sigma^{\vec{n}}$ denotes the shift map that translates the configurations so that cell $\vec{n}$ moves to the origin.

By carefully choosing the update rule f, the global dynamics $c \mapsto c'$ can be made information preserving. In this case, an inverse cellular automaton retraces the computation back in time, and the cellular automaton is called reversible (RCA). See [90] for a recent survey on reversible cellular automata. Cellular automata have an important role as providing simple models in microscopic physics, and because of time-reversibility of microscopic dynamics the cellular automata models are also typically reversible [181]. Reversible cellular automata are able to carry out universal computation [180], even in the one-dimensional setting [144].

In the symbolic dynamics nomenclature reversible cellular automata are called automorphisms of the (full) shift. By Hedlund's theorem [69] cellular automata are precisely the transformations $A^{\mathbb{Z}^d} \longrightarrow A^{\mathbb{Z}^d}$ of the configuration space that commute with shifts $\sigma^{\vec{n}}$ and that are continuous under the compact prodiscrete topology on $A^{\mathbb{Z}^d}$. Reversibility then just means that the transformation is a bijection, i.e., a homeomorphism. Automorphisms form a group under composition, and the structure of the automorphism group of the full shift (as well as of its subshifts) is a topic of active research [168]. For example, it is not known if the groups of one-dimensional RCA over two states and over three states are isomorphic with each other.

Decision Problems. Decision problems concerning reversibility and related properties have been extensively studied. There are efficient algorithms to test one-dimensional cellular automata for reversibility [177] while in higher dimensional cases reversibility is undecidable [88]. It is also undecidable, even in the one-dimensional case, whether a given RCA is periodic [92], that is, whether some iteration of the CA amounts to the identity function. Periodicity among one-sided RCA is not known to be decidable or undecidable at this time, where

one-sidedness refers to the property that the neighbours to the left of a cell have no influence on its next state, nor on the previous state given by the inverse automaton. Periodicity in the one-sided case remains an active research topic due to its link to the finiteness problem of groups generated by Mealy automata [51].

Two dynamical systems are called conjugate if there is a homeomorphism between them that maps orbits to orbits. Conjugate systems are essentially identical. It is undecidable if two given cellular automata are conjugate [81]. This is true even for one-dimensional cellular automata, but if the considered CA are reversible then the undecidability is known in the two- and higher dimensional cases only.

Physical Universality and Glider Automorphisms. A cellular automaton is called physically universal if it can implement any transformation of patterns on any finite domain of cells by suitably choosing the initial states outside the domain. There are reversible cellular automata that are physically universal [170], even in the one-dimensional setting [169]. These automata (reversibly) break the input pattern into fleets of gliders that scatter out of the finite domain. Symmetrically, the inverse automaton breaks the desired output pattern into fleets of inverse gliders. The task of the surrounding gadget is to change the first fleet into the second fleet to implement the desired transformation.

Glider automorphisms that decompose finite configurations into fleets of gliders have been studied in more general subshifts, and they have found applications in understanding the structure of the automorphism groups [100].

Reversible Cellular Automata and Mahler's Problem in Number Theory. If real numbers are written in base pq for some co-primes p and q then there is no carry propagation when numbers are multiplied by constant p. This means that multiplying by p is a local operation, that is, a reversible cellular automaton. Composing such reversible cellular automata yields, for example, an RCA for multiplying numbers in base 6 by constant $3/2$.

Mahler's problem asks whether there exists some positive real number ξ such that the fractional part of $\xi\left(\frac{3}{2}\right)^n$ is less than 0.5 for all positive integers n [127]. So the fractional part of the number should remain below one half no matter how many times the number is multiplied by $3/2$. The problem is still unsolved. The problem has a very simple interpretation in terms of the RCA that multiplies by $3/2$ in base six [89], and using this link it has been proved that for arbitrarily small $\varepsilon > 0$ there is a number $\xi > 0$ and a finite union $U \subseteq [0, 1)$ of intervals of total length ε such that the fractional parts of all $\xi\left(\frac{3}{2}\right)^n$ are in U [91]. The dynamical property of expansivity of the associated reversible cellular automaton plays a central role in the proof. Conversely, there is also a finite union $V \subseteq [0, 1)$ of intervals of total length $1 - \varepsilon$ that does not contain the fractional parts of all $\xi\left(\frac{3}{2}\right)^n$ for any $\xi > 0$.

Asynchronous Updating. In an asynchronous cellular automaton (ACA) only some cells are updated simultaneously. In the one-dimensional setting, one possibility is that states are updated sequentially during a left-to-right (or right-to-left) sweep across the entire infinite line of cells. Such a setup is studied in [93] where the update performed once in each position is given by a reversible block rule $A^n \longrightarrow A^n$ on n consecutive cells. The authors give a precise characterisation of the one-dimensional cellular automata that can be realised by such a sweep. It turns out that not all reversible CA can be realised, while also some non-reversible ones can be obtained. It is decidable whether a CA can be realised that way or not.

Self-Timed Cellular Automata. *Self-Timed Cellular Automata (STCA)* are a form of Asynchronous Cellular Automata where transitions of cells can take place if they are triggered by transitions of the neighbouring cells. *Delay-Insensitive (DI)* circuits are asynchronous circuits which make no assumption about delays within modules or wires of circuits, and where there is no global clock [97]. As a result, logical gates such as NAND and XOR are not Turing-complete when operated in a DI environment. A lot of research went into finding universal sets of DI modules and [145] contributes a solution for reversible DI circuits in terms of STCAs. Serial and parallel DI circuits are simulated with new STCAs that contain rules for signal movement, right and left turn, memory toggle, merge, fork and join, and parallel crossing of signals. In addition to a number of reversibility and determinism properties, including local determinism and local reversibility, the STCAs exhibit direction-reversibility, where reversing the direction of a signal and running a circuit forwards is equivalent to running the circuit in reverse. Benefits of direction-reversibility are discussed, including garbage-less implementation of reversible functions.

Cellular Automata as Language Acceptors. From the perspective of language recognition, real-time bounded cellular automata which are reversible on the core of computation, that is, from initial configuration to the configuration given by the time complexity, have been studied in [106]. The question whether for a given real-time CA working on finite configurations with fixed boundary conditions there exists a reverse real-time CA with the same neighbourhood has been addressed. It has been shown that real-time reversibility is undecidable, which contrasts the general case, where reversibility is decidable for one-dimensional devices. Moreover, the undecidability of emptiness, finiteness, infiniteness, inclusion, equivalence, regularity, and context-freedom has been proved. First steps towards the exploration of the computational capacity have been done and closure under Boolean operations have been shown.

Similar investigations for real-time one-way cellular automata have been done in [108]. In this case, it turned out that the standard model with fixed boundary conditions is quite weak in terms of reversible information processing, since it accepts exactly the regular languages reversibly. The extension that allows the information to flow circularly from the leftmost cell into the rightmost cell does

not increase the computational power in the general case, but does increase it for reversible computations. On the other hand, the model is less powerful than real-time reversible two-way cellular automata. Additionally, it has been derived that the corresponding language class is closed under Boolean operations, and the undecidability of several decidability questions has been proved. Finally, it turned out that the reversibility of an arbitrary real-time circular one-way cellular automaton is undecidable as well.

8.6 Turing Machines

Turing machines (TM) are a classical model of computation where a finite state control unit, the head, moves along a bi-infinite tape of cells, each containing a tape symbol. The head reads and writes symbols on the tape, changes its internal state, and moves to neighbouring cells at discrete time steps as instructed by a fixed transition rule, the program of the TM. A suitable choice of the program makes the machine reversible (RTM). Turing machines are traditionally viewed as language acceptors, but one can also incorporate outputs in the model so that the machine becomes a transducer that computes a (partial) function. In [12] the authors investigate RTM under the strict function semantics that requires that at the end of the computation only the output remains on the tape, and they develop a rigorous foundational theory of reversible computation of functions in this semantics, including the appropriate concept of universality and a design of a universal machine.

Turing machines with bi-infinite tape contents are also discrete dynamical systems (on a compact space) under two possible viewpoints [104]: in the moving tape view (TMT) the position of the head is fixed but the entire tape shifts left or right depending on the current instruction, while in the usual moving head view (TMH) one needs to allow configurations without a head to make the configuration space compact. In [38] the authors present a reversible TMT with the rather surprising property that it has no halting or temporally periodic configurations, thus answering positively a conjecture made in [92]. The machine, dubbed “SMART”, is small (4 internal states, 3 tape symbols) and nicely symmetric in both time and space. It possesses the good dynamical properties of transitivity and minimality. The machine is further applied to settle another conjecture made in [92]: it is undecidable whether a given complete reversible Turing machine has a periodic orbit.

The class of RTM dynamical systems becomes more robust if the head is allowed to view and modify locally blocks of several tape symbols at once. In particular, compositions of machines and inverse machines are now in the same class so that reversible Turing machines with any fixed states and tape symbols form a group under composition. The structure of this group and algorithmic questions concerning the group are studied in [25]. The paper also introduces a number of natural subgroups. The model includes multidimensional Turing machines where the tape cells are indexed by $\mathbb{Z}^d$ for dimension d, and both the moving head and the moving tape viewpoints can be taken.

Finally, reversible Turing machines with a working tape and a one-way or two-way read-only input tape are considered as language recognisers [15]. In particular, the classes of languages acceptable by such devices with small time bounds in the range between real time and linear time, that is, with time bounds of the form $n+r(n)$ where $r \in o(n)$ is a sublinear function, have been considered. It has been shown that there exist infinite time hierarchies of separated complexity classes in that range. The question of whether reversible Turing machines in the range of interest are weaker than general ones or not is answered positively by proving that there are languages accepted by irreversible one-way Turing machines in real time that cannot be accepted by any reversible one-way machine in less than linear time.

9 Quantum Formal Verification and Quantum Machine Learning

Large-scale, fault-tolerant quantum computers are still under development and, despite a recent major push for "quantum supremacy" by companies like IBM, Google and Intel, it is not clear when they will become a reality. On the other hand there is much recent interest in using Noisy Intermediate Scale Quantum (NISQ) computers to provide a "quantum advantage". This involves the use of existing or near-term quantum computers to solve valuable problems, faster, cheaper, or more efficiently than any available classical solution. Potential application areas include simulation of many-body physics, quantum chemistry, optimisation and quantum machine learning. Airbus has issued its Quantum Computing Challenge to tackle aerospace flight physics problems using quantum computers. Many companies such as IBM, Microsoft, D-Wave, Rigetti and Xanadu are developing full-stack solutions for implementing quantum algorithms. This typically starts from a high-level programming language and a compiler, down to an assembly language and quantum hardware. These resources are usually accessible via the cloud. Much of these developments will need guarantees regarding security and correctness. Formal verification, which has been used successfully in classical computing for a number of years, could be extremely valuable in increasing confidence in quantum systems.

Quantum cryptography aims to overcome the limitations of classical cryptography by providing unconditional security, which is not dependent on the difficulty of inverting a particular computation. Quantum Key Distribution protocols have been implemented in commercial products by Id Quantique, MagiQ, NEC and Toshiba, amongst others, and have been used in practical applications, e.g. the Geneva election ballot count. Various QKD networks have been built, including the DARPA Quantum Network in Boston, the SeCoQC network around Vienna and the Tokyo QKD Network. China has launched a dedicated satellite "Micius" for quantum communication. On the theoretical side, quantum key distribution protocols such as BB84 [29] have been proved to be unconditionally secure. It is important to understand, however, that this is an

information-theoretic proof, which does not necessarily guarantee that *implemented systems* are unconditionally secure. This area is also where approaches such as those based on formal methods could be useful in analysing behaviour of implemented systems.

The paper [32] presents a novel framework for modelling and verifying quantum protocols and their implementations using the proof assistant Coq. It provides a Coq library for quantum bits (qubits), quantum gates, and quantum measurement. As a step towards verifying practical quantum communication and security protocols such as Quantum Key Distribution, it supports multiple qubits, communication and entanglement. These concepts are illustrated by modelling the Quantum Teleportation Protocol, which communicates the state of an unknown quantum bit using only a classical channel. In more recent work, a Quantum IO monad has been implemented in Coq for the specification of the protocols. In addition to quantum operations and measurement, the monad gives us a lightweight process calculus which supports sequencing of operations and keeping of state. This monad has the necessary properties. The process simulation function that gives the QIO monad its semantics has also been written. Current work concerns proving properties of simple quantum protocols.

In [10], the authors present CCSq, a concurrent language for describing quantum systems, and develop verification techniques for checking equivalence between CCSq processes. CCSq has well-defined operational and superoperator semantics for protocols that are functional, in the sense of computing a deterministic input-output relation for all interleavings arising from concurrency in the system. They have implemented QEC (Quantum Equivalence Checker), a tool that takes the specification and implementation of quantum protocols, described in CCSq, and automatically checks their equivalence. QEC is the first fully automatic equivalence checking tool for concurrent quantum systems. For efficiency purposes, the approach is restricted to Clifford operators in the stabiliser formalism, but it is able to verify protocols over all input states. A collection of interesting and practical quantum protocols, ranging from quantum communication and quantum cryptography to quantum error correction, have been specified and verified.

In other recent work, a version of the quantum process calculus CQP has been implemented. The implementation, which has the working title qtpi and is available from `github.com/mdxtoc/qtpi`, uses symbolic rather than numeric probability calculation. Programs are checked statically, before they run, to ensure that they obey real-world restrictions on the use of qbits (e.g. no cloning, no sharing). Qtpi has been used to simulate some simple protocols such as teleportation, and some more involved ones including QKD. It is early days in the development of the tool, but it can already simulate well over 1M qbit transfers per minute.

Quantum machine learning is the aspect of quantum computing concerned with the design of algorithms capable of generalised learning from labelled training data by effectively exploiting quantum effects. The undertaken work makes various contributions to this emerging area; in particular it has pursued the

issue of classification error within a standard quantum computational setting, and explored the congruence of Kernel Methods with the topological quantum computational setting (a congruence that will be developed further in future work).

Specifically, the following have been achieved:

In [52] the authors present a novel approach to computing Hamming distance and its kernelisation within Topological Quantum Computation. This approach is based on an encoding of two binary strings into a topological Hilbert space, whose inner product yields a natural Hamming distance kernel on the two strings. Kernelisation forges a link with the field of Machine Learning, particularly in relation to binary classifiers such as the Support Vector Machine (SVM). This makes our approach of potentially wide interest to the quantum machine learning community.

In [183], the authors set out a strategy for quantising attribute bootstrap aggregation to enable variance-resilient quantum machine learning. To do so, they utilise the linear decomposability of decision boundary parameters in the Rebentrost et al. Support Vector Machine [164] to guarantee that stochastic measurement of the output quantum state will give rise to an ensemble decision without destroying the superposition over projective feature subsets induced within the chosen SVM implementation. It achieves a linear performance advantage, $O(d)$, in addition to the existing $O(\log(n))$ advantages of quantisation as applied to Support Vector Machines. The approach extends to any form of quantum learning giving rise to linear decision boundaries.

Error-correcting output codes (ECOC) are a standard setting in machine learning for efficiently rendering the collective outputs of a binary classifier, such as the support vector machine, as a multi-class decision procedure. Appropriate choice of error-correcting codes further enables incorrect individual classification decisions to be effectively corrected in the composite output. In [182], the authors propose an appropriate quantisation of the ECOC process, based on the quantum support vector machine. They show that, in addition to the usual benefits of quantising machine learning, this technique leads to an exponential reduction in the number of logic gates required for effective correction of classification error.

10 Conclusion

We gave in the previous sections an overview of the status and recent developments of different research threads on the foundations of reversible computation. While many interesting results have been found, we notice that the field is still very heterogeneous. For instance, while process calculi, Petri nets and cellular automata are all models of concurrent systems, they come equipped with different notions of reversibility. Cellular automata are considered reversible if the global dynamics is bijective (similarly to what is done in sequential reversible models), Petri nets if reverse transitions can be added without changing the behaviour of the net, while process calculi are mainly based on the notion of causal-consistent reversibility. Some initial cross-fertilisation results came thanks

to the COST Action, e.g. there have been works applying causal-consistent reversibility to Petri nets [132] and related models [27,155]. We also remark that some of the developments described in this chapter have been instrumental to better understand reversibility in programming languages and to advance on a number of application areas, as discussed in the rest of the book.

References

1. Abramsky, S.: Retracing some paths in process algebra. In: Montanari, U., Sassone, V. (eds.) CONCUR 1996. LNCS, vol. 1119, pp. 1–17. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61604-7_44
2. Abramsky, S.: A structural approach to reversible computation. Theoret. Comput. Sci. **347**(3), 441–464 (2005)
3. Abramsky, S., Coecke, B.: A categorical semantics of quantum protocols. In: Logic in Computer Science, LICS 2004, pp. 415–425. IEEE (2004)
4. Abramsky, S., Haghverdi, E., Scott, P.: Geometry of interaction and linear combinatory algebras. Math. Struct. Comput. Sci. **12**(5), 625–665 (2002)
5. Agrigoroaiei, O., Ciobanu, G.: Dual P systems. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 95–107. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-95885-7_7
6. Agrigoroaiei, O., Ciobanu, G.: Reversing computation in membrane systems. J. Logic Algebraic Program. **79**(3–5), 278–288 (2010)
7. Aman, B., Ciobanu, G.: Reversibility in parallel rewriting systems. J. Univers. Comput. Sci. **23**(7), 692–703 (2017)
8. Aman, B., Ciobanu, G.: Controlled reversibility in reaction systems. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) CMC 2017. LNCS, vol. 10725, pp. 40–53. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73359-3_3
9. Angluin, D.: Inference of reversible languages. J. ACM **29**(3), 741–765 (1982)
10. Ardeshir-Larijani, E., Gay, S.J., Nagarajan, R.: Automated equivalence checking of concurrent quantum systems. ACM Trans. Comput. Logic **19**(4), 28:1–28:32 (2018)
11. Axelsen, H.B., Glück, R.: Reversible representation and manipulation of constructor terms in the heap. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 96–109. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38986-3_9
12. Axelsen, H.B., Glück, R.: On reversible turing machines and their function universality. Acta Inf. **53**(5), 509–543 (2016)
13. Axelsen, H.B., Holzer, M., Kutrib, M.: The degree of irreversibility in deterministic finite automata. Int. J. Found. Comput. Sci. **28**, 503–522 (2017)
14. Axelsen, H.B., Holzer, M., Kutrib, M., Malcher, A.: Reversible shrinking two-pushdown automata. In: Dediu, A.-H., Janoušek, J., Martín-Vide, C., Truthe, B. (eds.) LATA 2016. LNCS, vol. 9618, pp. 579–591. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30000-9_44
15. Axelsen, H.B., Jakobi, S., Kutrib, M., Malcher, A.: A hierarchy of fast reversible turing machines. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 29–44. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_2
16. Axelsen, H.B., Kutrib, M., Malcher, A., Wendlandt, M.: Boosting reversible pushdown machines by preprocessing. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 89–104. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_6

17. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
18. Bacci, G., Danos, V., Kammar, O.: On the statistical thermodynamics of reversible communicating processes. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 1–18. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22944-2_1
19. Badouel, E., Bernardinello, L., Darondeau, P.: Petri Net Synthesis. TTCSAES. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47967-4
20. Barbanera, F., de'Liguoro, U.: A game interpretation of retractable contracts. In: Lluch Lafuente, A., Proença, J. (eds.) COORDINATION 2016. LNCS, vol. 9686, pp. 18–34. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39519-7_2
21. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Compliance for reversible client/server interactions. In: Workshop on Behavioural Types, BEAT 2014. EPTCS, vol. 162, pp. 35–42 (2014)
22. Barbanera, F., Dezani-Ciancaglini, M., de'Liguoro, U.: Reversible client/server interactions. Formal Asp. Comput. **28**(4), 697–722 (2016)
23. Barbanera, F., Dezani-Ciancaglini, M., Lanese, I., de'Liguoro, U.: Retractable contracts. In: Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2015. EPTCS, vol. 203, pp. 61–72 (2015)
24. Barbanera, F., Lanese, I., de'Liguoro, U.: Retractable and speculative contracts. In: Jacquet, J.-M., Massink, M. (eds.) COORDINATION 2017. LNCS, vol. 10319, pp. 119–137. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59746-1_7
25. Barbieri, S., Kari, J., Salo, V.: The group of reversible turing machines. In: Cook, M., Neary, T. (eds.) AUTOMATA 2016. LNCS, vol. 9664, pp. 49–62. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39300-1_5
26. Barylska, K., Erofeev, E., Koutny, M., Mikulski, Ł., Piątkowski, M.: Reversing transitions in bounded Petri nets. Fund. Inf. **157**(4), 341–357 (2018)
27. Barylska, K., Gogolińska, A., Mikulski, Ł., Philippou, A., Piątkowski, M., Psara, K.: Reversing computations modelled by coloured Petri nets. In: Workshop on Algorithms & Theories for the Analysis of Event Data. CEUR Workshop Proceedings, vol. 2115, pp. 91–111. CEUR-WS.org (2018)
28. Barylska, K., Koutny, M., Mikulski, Ł., Piątkowski, M.: Reversible computation vs. reversibility in Petri nets. Sci. Comput. Program. **151**, 48–60 (2018)
29. Bennett, C.H., Brassard, G.: Quantum cryptography: public key distribution and coin tossing. In: Conference on Computers, Systems & Signal Processing, CSSP 1984, pp. 175–179 (1984)
30. Berry, G., Boudol, G.: The chemical abstract machine. Theor. Comput. Sci. **96**(1), 217–248 (1992)
31. Berstel, J.: Transductions and Context-Free Languages. Teubner, Stuttgart (1979)
32. Boender, J., Kammüller, F., Nagarajan, R.: Formalization of quantum protocols using Coq. In: Workshop on Quantum Physics and Logic, QPL 2015, pp. 71–83 (2015)
33. Bouziane, Z., Finkel, A.: Cyclic Petri net reachability sets are semi-linear effectively constructible. In: Workshop on Verification of Infinite State Systems, INFINITY 1997, ENTCS, pp. 15–24. Elsevier (1997)
34. Bowman, W.J., James, R.P., Sabry, A.: Dagger traced symmetric monoidal categories and reversible programming. In: Reversible Computation, RC 2011, pp. 51–56. Ghent University (2011)

35. Cardelli, L., Laneve, C.: Reversibility in massive concurrent systems. Sci. Ann. Comp. Sci. **21**(2), 175–198 (2011)
36. Cardelli, L., Laneve, C.: Reversible structures. In: Computational Methods in Systems Biology, CMSB 2011, pp. 131–140. ACM (2011)
37. Carothers, C.D., Perumalla, K.S., Fujimoto, R.: Efficient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul. **9**(3), 224–253 (1999)
38. Cassaigne, J., Ollinger, N., Torres-Avilés, R.: A small minimal aperiodic reversible Turing machine. J. Comput. Syst. Sci. **84**, 288–301 (2017)
39. Clavel, M., et al.: Maude: specification and programming in rewriting logic. Theor. Comput. Sci. **285**(2), 187–243 (2002)
40. Cockett, J.R.B., Lack, S.: Restriction categories I: categories of partial maps. Theoret. Comput. Sci. **270**(1–2), 223–259 (2002)
41. Colange, M., Baarir, S., Kordon, F., Thierry-Mieg, Y.: Crocodile: a symbolic/symbolic tool for the analysis of symmetric nets with bag. In: Kristensen, L.M., Petrucci, L. (eds.) PETRI NETS 2011. LNCS, vol. 6709, pp. 338–347. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21834-7_20
42. Cristescu, I., Krivine, J., Varacca, D.: A compositional semantics for the reversible π-calculus. In: Logic in Computer Science, LICS 2013, pp. 388–397. IEEE Computer Society (2013)
43. Cservenka, M.H., Glück, R., Haulund, T., Mogensen, T.Æ.: Data structures and dynamic memory management in reversible languages. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 269–285. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_19
44. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
45. Danos, V., Krivine, J.: Transactions in RCCS. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 398–412. Springer, Heidelberg (2005). https://doi.org/10.1007/11539452_31
46. Danos, V., Krivine, J.: Formal molecular biology done in CCS-R. In: Workshop on Concurrent Models in Molecular Biology, BioConcur 2003, vol. 180(3) (2003). Electr. Notes Theor. Comput. Sci., 31–49. Elsevier (2007)
47. Dartois, L., Fournier, P., Jecker, I., Lhote, N.: On reversible transducers. In: International Colloquium on Automata, Languages, and Programming, ICALP 2017. LIPIcs, vol. 80, pp. 113:1–113:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017)
48. de Frutos Escrig, D., Koutny, M., Mikulski, Ł.: An efficient characterization of Petri net solvable binary words. In: Khomenko, V., Roux, O.H. (eds.) PETRI NETS 2018. LNCS, vol. 10877, pp. 207–226. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91268-4_11
49. de Frutos Escrig, D., Koutny, M., Mikulski, Ł.: Reversing steps in Petri nets. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 171–191. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_11
50. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. **34**, 83–133 (1984)
51. Delacourt, M., Ollinger, N.: Permutive one-way cellular automata and the finiteness problem for automaton groups. In: Kari, J., Manea, F., Petre, I. (eds.) CiE 2017. LNCS, vol. 10307, pp. 234–245. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58741-7_23

52. Di Pierro, A., Mengoni, R., Nagarajan, R., Windridge, D.: Hamming distance kernelisation via topological quantum computation. In: Martín-Vide, C., Neruda, R., Vega-Rodríguez, M.A. (eds.) TPNC 2017. LNCS, vol. 10687, pp. 269–280. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-71069-3_21
53. Ehrenfeucht, A., Rozenberg, G.: Reaction systems. Fund. Inf. **75**(1), 263–280 (2007)
54. Esparza, J., Nielsen, M.: Decidability issues for Petri nets. BRICS Rep. Ser. **1**(8) (1994)
55. Foster, N., Matsuda, K., Voigtländer, J.: Three complementary approaches to bidirectional programming. In: Gibbons, J. (ed.) Generic and Indexed Programming. LNCS, vol. 7470, pp. 1–46. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32202-0_1
56. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility in a tuple-based language. In: Parallel, Distributed, and Network-Based Processing, PDP 2015, pp. 467–475. IEEE Computer Society (2015)
57. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. J. Log. Algebr. Meth. Program. **88**, 99–120 (2017)
58. Giles, B.G.: An investigation of some theoretical aspects of reversible computing. Ph.D. thesis, University of Calgary (2014)
59. Glück, R., Kaarsgaard, R.: A categorical foundation for structured reversible flowchart languages. In: Mathematical Foundations of Programming Semantics, MFPS 2018. Electronic Notes in Theoretical Computer Science, vol. 341, pp. 155–171. Elsevier (2018)
60. Glück, R., Kaarsgaard, R.: A categorical foundation for structured reversible flowchart languages: soundness and adequacy. Logical Methods Comput. Sci. **14**(3) (2018)
61. Glück, R., Kaarsgaard, R., Yokoyama, T.: Reversible programs have reversible semantics. In: Reversibility in Programming, Languages, and Automata, RPLA 2019. Lecture Notes in Computer Science. Springer (2019, to appear)
62. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. Comput. Soft. **33**(3), 108–128 (2016)
63. Glück, R., Yokoyama, T.: A minimalist's reversible while language. IEICE Trans. Inf. Syst. **E100–D**(5), 1026–1034 (2017)
64. Glück, R., Yokoyama, T.: Constructing a binary tree from its traversals by reversible recursion and iteration. Inf. Process. Lett. **147**, 32–37 (2019)
65. Graversen, E., Phillips, I., Yoshida, N.: Towards a categorical representation of reversible event structures. J. Logical Algebraic Methods Program. **104**, 16–59 (2019)
66. Guillon, B., Kutrib, M., Malcher, A., Prigioniero, L.: Reversible pushdown transducers. In: Hoshi, M., Seki, S. (eds.) DLT 2018. LNCS, vol. 11088, pp. 354–365. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-98654-8_29
67. Guo, X.: Products, joins, meets, and ranges in restriction categories. Ph.D. thesis, University of Calgary (2012)
68. Haulund, T., Mogensen, T.Æ., Glück, R.: Implementing reversible object-oriented language features on reversible machines. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 66–73. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_5
69. Hedlund, G.A.: Endomorphisms and automorphisms of the shift dynamical systems. Mathe. Syst. Theor. **3**(4), 320–375 (1969)

70. Heunen, C., Kaarsgaard, R., Karvonen, M.: Reversible effects as inverse arrows. In: Mathematical Foundations of Programming Semantics, MFPS XXXIV. Electronic Notes in Theoretical Computer Science, vol. 341, pp. 179–199. Elsevier (2018)
71. Heunen, C., Karvonen, M.: Monads on dagger categories. Theor. Appl. Categories **31**, 1016–1043 (2016)
72. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: Expressiveness in Concurrency/Structural Operational Semantics. Electronic Proceedings in Theoretical Computer Science, vol. 276, pp. 69–86 (2018)
73. Holzer, M., Jakobi, S., Kutrib, M.: Minimal reversible deterministic finite automata. Int. J. Found. Comput. Sci. **29**(2), 251–270 (2018)
74. Holzer, M., Kutrib, M.: Reversible nondeterministic finite automata. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 35–51. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_3
75. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Bidirectional transformation "bx" (Dagstuhl Seminar 11031). Dagstuhl Reports **1**(1), 42–67 (2011). http://drops.dagstuhl.de/volltexte/2011/3144/
76. Hullot, J.-M.: Canonical forms and unification. In: Bibel, W., Kowalski, R. (eds.) CADE 1980. LNCS, vol. 87, pp. 318–334. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10009-1_25
77. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
78. European COST Action IC1405 on "Reversible Computation - Extending Horizons of Computing". http://www.revcomp.eu/
79. Jacobs, B.: New directions in categorical logic, for classical, probabilistic and quantum logic. Logical Methods Comput. Sci. **11**(3), 1–76 (2015)
80. Jacobsen, P.A.H., Kaarsgaard, R., Thomsen, M.K.: CoreFun: a typed functional reversible core language. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 304–321. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_21
81. Jalonen, J., Kari, J.: Conjugacy of one-dimensional one-sided cellular automata is undecidable. In: Tjoa, A.M., Bellatreche, L., Biffl, S., van Leeuwen, J., Wiedermann, J. (eds.) SOFSEM 2018. LNCS, vol. 10706, pp. 227–238. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73117-9_16
82. James, R.P., Sabry, A.: Theseus: a high level language for reversible computing. In: Work-in-Progress Report Presented at RC 2014. http://www.cs.indiana.edu/~sabry/papers/theseus.pdf
83. James, R.P., Sabry, A.: Information effects. ACM SIGPLAN Not. **47**(1), 73–84 (2012)
84. Jones, N.D.: Computability and Complexity: From a Programming Language Perspective. Foundations of Computing. MIT Press, Cambridge (1997)
85. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Math. Proc. Cambridge Philos. Soc. **119**(3), 447–468 (1996)
86. Kaarsgaard, R., Axelsen, H.B., Glück, R.: Join inverse categories and reversible recursion. J. Logical Algebraic Methods Program. **87**, 33–50 (2017)
87. Kaarsgaard, R., Glück, R.: A categorical foundation for structured reversible flowchart languages: soundness and adequacy. Logical Methods Comput. Sci. **14**(3), 1–38 (2018)
88. Kari, J.: Reversibility of 2D cellular automata is undecidable. Physica D **45**(1), 379–385 (1990)

89. Kari, J.: Universal pattern generation by cellular automata. Theoret. Comput. Sci. **429**, 180–184 (2012)
90. Kari, J.: Reversible cellular automata: from fundamental classical results to recent developments. New Generation Comput. **36**(3), 145–172 (2018)
91. Kari, J., Kopra, J.: Cellular automata and powers of p/q. RAIRO - Theor. Inf. Applic. **51**(4), 191–204 (2017)
92. Kari, J., Ollinger, N.: Periodicity and immortality in reversible computing. In: Ochmański, E., Tyszkiewicz, J. (eds.) MFCS 2008. LNCS, vol. 5162, pp. 419–430. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-85238-4_34
93. Kari, J., Salo, V., Worsch, T.: Sequentializing cellular automata. In: Baetens, J.M., Kutrib, M. (eds.) AUTOMATA 2018. LNCS, vol. 10875, pp. 72–87. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92675-9_6
94. Karvonen, M.: The way of the dagger. Ph.D. thesis, School of Informatics, University of Edinburgh (2019)
95. Kastl, J.: Inverse categories. In: Algebraische Modelle, Kategorien und Gruppoide. Studien zur Algebra und ihre Anwendungen, vol. 7, pp. 51–60. Akademie-Verlag (1979)
96. Kawabe, M., Glück, R.: The program inverter LRinv and its structure. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 219–234. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30557-6_17
97. Keller, R.: Towards a theory of universal speed-independent modules. IEEE Trans. Comput. **23**(1), 21–33 (1974)
98. Klop, J.W.: Term rewriting systems. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. I, pp. 1–112. Oxford University Press (1992)
99. Knowlton, K.C.: A fast storage allocator. Commun. ACM **8**(10), 623–625 (1965)
100. Kopra, J.: Glider automorphisms on some shifts of finite type and a finitary Ryan's theorem. In: Baetens, J.M., Kutrib, M. (eds.) AUTOMATA 2018. LNCS, vol. 10875, pp. 88–99. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92675-9_7
101. Krivine, J.: A verification technique for reversible process algebra. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 204–217. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_17
102. Kuhn, S., Ulidowski, I.: A calculus for local reversibility. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 20–35. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_2
103. Kuhn, S., Ulidowski, I.: Local reversibility in a calculus of covalent bonding. Sci. Comput. Program. **151**, 18–47 (2018)
104. Kurka, P.: On topological dynamics of Turing machines. Theor. Comput. Sci. **174**(1–2), 203–216 (1997)
105. Kutrib, M.: Reversible and irreversible computations of deterministic finite-state devices. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 38–52. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_3
106. Kutrib, M., Malcher, A.: Fast reversible language recognition using cellular automata. Inf. Comput. **206**, 1142–1151 (2008)
107. Kutrib, M., Malcher, A.: Reversible pushdown automata. J. Comput. Syst. Sci. **78**, 1814–1827 (2012)

108. Kutrib, M., Malcher, A., Wendlandt, M.: Real-time reversible one-way cellular automata. In: Isokawa, T., Imai, K., Matsui, N., Peper, F., Umeo, H. (eds.) AUTOMATA 2014. LNCS, vol. 8996, pp. 56–69. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18812-6_5
109. Kutrib, M., Malcher, A., Wendlandt, M.: Reversible queue automata. Fund. Inf. **148**, 341–368 (2016)
110. Kutrib, M., Malcher, A., Wendlandt, M.: When input-driven pushdown automata meet reversiblity. RAIRO - Theor. Inf. Applic. **50**, 313–330 (2016)
111. Kutrib, M., Malcher, A., Wendlandt, M.: Transducing reversibly with finite state machines. Theor. Comput. Sci. **787**, 111–126 (2019)
112. Kutrib, M., Wendlandt, M.: Reversible limited automata. Fund. Inf. **155**, 31–58 (2017)
113. Landauer, R.: Irreversibility and heat generated in the computing process. IBM J. Res. Dev. **5**, 183–191 (1961)
114. Lanese, I., Lienhardt, M., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Concurrent flexible reversibility. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 370–390. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_21
115. Lanese, I., Medic, D., Mezzina, C.A.: Static versus dynamic reversibility in CCS. Acta Informatica (2019)
116. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order Pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_20
117. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Reversing higher-order Pi. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 478–493. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_33
118. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Controlled reversibility and compensations. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 233–240. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_19
119. Lanese, I., Mezzina, C.A., Stefani, J.-B.: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. **625**, 25–84 (2016)
120. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. Bull. EATCS **114** (2014)
121. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
122. Laursen, J.S., Schultz, U.P., Ellekilde, L.: Automatic error recovery in robot assembly operations using reverse execution. In: Intelligent Robots and Systems, IROS 2015, pp. 1785–1792. IEEE (2015)
123. Lavado, G.J., Pighizzini, G., Prigioniero, L.: Minimal and reduced reversible automata. J. Automata, Lang. Comb. **22**(1–3), 145–168 (2017)
124. Lavado, G.J., Pighizzini, G., Prigioniero, L.: Weakly and strongly irreversible regular languages. In: Automata and Formal Languages, AFL 2017. EPTCS, vol. 252, pp. 143–156 (2017)
125. Lavado, G.J., Prigioniero, L.: Concise representations of reversible automata. Int. J. Found. Comput. Sci. **30**(6–7), 1157–1175 (2019)

126. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.-B.: A reversible abstract machine and its space overhead. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE -2012. LNCS, vol. 7273, pp. 1–17. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30793-5_1
127. Mahler, K.: An unsolved problem on the powers of 3/2. J. Australian Math. Soc. **8**(2), 313–321 (1968)
128. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: International Conference on Functional Programming, ICFP 2007, pp. 47–58. ACM (2007)
129. McNellis, J., Mola, J., Sykes, K.: Time travel debugging: root causing bugs in commercial scale software. CppCon talk (2017). https://www.youtube.com/watch?v=l1YJTg_A914
130. Medić, D., Mezzina, C.A.: Static VS dynamic reversibility in CCS. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 36–51. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_3
131. Medic, D., Mezzina, C.A., Phillips, I., Yoshida, N.: A parametric framework for reversible pi-calculi. In: Workshop on Expressiveness in Concurrency and Workshop on Structural Operational Semantics, EXPRESS/SOS 2018. EPTCS, vol. 276, pp. 87–103 (2018)
132. Melgratti, H., Mezzina, C.A., Ulidowski, I.: Reversing P/T Nets. In: Riis Nielson, H., Tuosto, E. (eds.) COORDINATION 2019. LNCS, vol. 11533, pp. 19–36. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22397-7_2
133. Mezzina, C.A.: On reversibility and broadcast. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 67–83. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_5
134. Mezzina, C.A., et al.: Software and reversible systems: a survey of recent activities. In: Ulidowski, I., et al. (eds.) Reversible Computation. LNCS 12070, pp. 41–59. Springer, Cham (2020)
135. Mezzina, C.A., Koutavas, V.: A safety and liveness theory for total reversibility. In: Theoretical Aspects of Software Engineering, TASE 2017, pp. 1–8. IEEE Computer Society (2017)
136. Mezzina, C.A., Pérez, J.A.: Reversible sessions using monitors. In: Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016. EPTCS, vol. 211, pp. 56–64 (2016)
137. Mezzina, C.A., Pérez, J.A.: Causally consistent reversible choreographies: a monitors-as-memories approach. In: Principles and Practice of Declarative Programming, PPDP 2017, pp. 127–138. ACM (2017)
138. Mezzina, C.A., Pérez, J.A.: Reversibility in session-based concurrency: a fresh look. J. Log. Algebr. Meth. Program. **90**, 2–30 (2017)
139. Mikulski, Ł., Lanese, I.: Reversing unbounded Petri nets. In: Donatelli, S., Haar, S. (eds.) PETRI NETS 2019. LNCS, vol. 11522, pp. 213–233. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21571-2_13
140. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980). https://doi.org/10.1007/3-540-10235-3
141. Mogensen, T.Æ.: RSSA: a reversible SSA form. In: Mazzara, M., Voronkov, A. (eds.) PSI 2015. LNCS, vol. 9609, pp. 203–217. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41579-6_16
142. Mogensen, T.Æ.: Reversible garbage collection for reversible functional languages. New Gener. Compu. **36**(3), 203–232 (2018)

143. Morita, K.: Theory of Reversible Computing. Monographs in Theoretical Computer Science. An EATCS Series. Springer, Tokyo (2017). https://doi.org/10.1007/978-4-431-56606-9
144. Morita, K., Harao, M.: Computation universality of one-dimensional reversible (injective) cellular automata. IEICE Trans. **E72**(6), 758–762 (1989)
145. Morrison, D., Ulidowski, I.: Direction-reversible self-timed cellular automata for delay-insensitive circuits. J. Cellular Automata **12**(1–2), 101–120 (2016)
146. Mousavi, M.R., Reniers, M.A., Groote, J.F.: SOS formats and meta-theory: 20 years after. Theor. Comput. Sci. **373**(3), 238–272 (2007)
147. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
148. Nishida, N., Palacios, A., Vidal, G.: Reversible term rewriting. In: Formal Structures for Computation and Deduction, FSCD 2016. LIPIcs, vol. 52. pp. 28:1–28:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
149. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 259–274. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_15
150. Nishida, N., Palacios, A., Vidal, G.: Reversible computation in term rewriting. J. Log. Algebr. Meth. Program. **94**, 128–149 (2018)
151. Nishida, N., Vidal, G.: Program inversion for tail recursive functions. In: Rewriting Techniques and Applications, RTA 2011. LIPIcs, vol. 10, pp. 283–298. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2011)
152. Nishida, N., Vidal, G.: Characterizing compatible view updates in syntactic bidirectionalization. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 67–83. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_5
153. Paolini, L., Piccolo, M., Roversi, L.: A certified study of a reversible programming language. In: Types for Proofs and Programs, TYPES 2018. LIPIcs, vol. 69, pp. 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018)
154. Păun, G.: Computing with membranes. J. Comput. Syst. Sci. **61**(1), 108–143 (2000)
155. Philippou, A., Psara, K.: Reversible computation in Petri nets. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 84–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_6
156. Philippou, A., Psara, K., Siljak, H.: Controlling reversibility in reversing Petri nets with application to wireless communications. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 238–245. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_15
157. Phillips, I., Ulidowski, I.: Reversibility and asymmetric conflict in event structures. J. Log. Algebr. Meth. Program. **84**(6), 781–805 (2015)
158. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_18
159. Phillips, I., Ulidowski, I., Yuen, S.: Modelling of bonding with processes and events. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 141–154. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38986-3_12
160. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. In: Aceto, L., Ingólfsdóttir, A. (eds.) FoSSaCS 2006. LNCS, vol. 3921, pp. 246–260. Springer, Heidelberg (2006). https://doi.org/10.1007/11690634_17

161. Phillips, I.C.C., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. **73**(1–2), 70–96 (2007)
162. Pin, J.-E.: On reversible automata. In: Simon, I. (ed.) LATIN 1992. LNCS, vol. 583, pp. 401–416. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0023844
163. Pinna, G.M.: Reversing steps in membrane systems computations. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) CMC 2017. LNCS, vol. 10725, pp. 245–261. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-73359-3_16
164. Rebentrost, P., Mohseni, M., Lloyd, S.: Quantum support vector machine for big data classification. Phys. Rev. Lett. **113**, 130503 (2014)
165. Reisig, W.: Petri Nets: An Introduction. EATCS Monographs on Theoretical Computer Science, vol. 4. Springer, Heidelberg (1985). https://doi.org/10.1007/978-3-642-69968-9
166. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. **205**(2), 125–198 (2007)
167. Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control. In: Baier, C., Dal Lago, U. (eds.) FoSSaCS 2018. LNCS, vol. 10803, pp. 348–364. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89366-2_19
168. Salo, V.: Groups and monoids of cellular automata. In: Kari, J. (ed.) AUTOMATA 2015. LNCS, vol. 9099, pp. 17–45. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-47221-7_3
169. Salo, V., Törmä, I.: A one-dimensional physically universal cellular automaton. In: Kari, J., Manea, F., Petre, I. (eds.) CiE 2017. LNCS, vol. 10307, pp. 375–386. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58741-7_35
170. Schaeffer, L.: A physically universal cellular automaton. In: Innovations in Theoretical Computer Science, ITCS 2015, pp. 237–246. ACM (2015)
171. Schordan, M., Oppelstrup, T., Jefferson, D., Barnes Jr., P.D.: Generation of reversible C++ code for optimistic parallel discrete event simulation. New Gener. Comput. **36**(3), 257–280 (2018)
172. Schordan, M., Oppelstrup, T., Thomsen, M.K., Glück, R.: Reversible languages and incremental state saving in optimistic parallel discrete event simulation. In: Ulidowski, I., et al. (eds.) Reversible Computation. LNCS 12070, pp. 187–207. Springer, Cham (2020)
173. Schultz, U.P., Axelsen, H.B.: Elements of a reversible object-oriented language. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 153–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_10
174. Selinger, P.: Dagger compact closed categories and completely positive maps. In: Workshop on Quantum Programming Languages, QPL 2005. Electronic Notes in Theoretical Computer Science, vol. 170, pp. 139–163 (2005)
175. Selinger, P.: A survey of graphical languages for monoidal categories. New Structures for Physics. Lecture Notes in Physics, vol. 813, pp. 289–355. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-12821-9_4
176. Slagle, J.R.: Automated theorem-proving for theories with simplifiers, commutativity and associativity. J. ACM **21**(4), 622–642 (1974)
177. Sutner, K.: De Bruijn graphs and linear cellular automata. Complex Syst. **5**(1), 19–30 (1991)
178. Terese: Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press (2003)
179. Tiezzi, F., Yoshida, N.: Reversible session-based pi-calculus. J. Log. Algebr. Meth. Program. **84**(5), 684–707 (2015)

180. Toffoli, T.: Computation and construction universality of reversible cellular automata. J. Comput. Syst. Sci. **15**(2), 213–231 (1977)
181. Toffoli, T., Margolus, N.: Cellular Automata Machines: A New Environment for Modeling. MIT Press, Cambridge (1987)
182. Windridge, D., Mengoni, R., Nagarajan, R.: Quantum error-correcting output codes. Int. J. Quantum Inf. **16**(8), 1840003 (2018)
183. Windridge, D., Nagarajan, R.: Quantum bootstrap aggregation. In: de Barros, J.A., Coecke, B., Pothos, E. (eds.) QI 2016. LNCS, vol. 10106, pp. 115–121. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-52289-0_9
184. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29517-1_2
185. Yokoyama, T., Axelsen, H.B., Glück, R.: Fundamentals of reversible flowchart languages. Theoret. Comput. Sci. **611**, 87–115 (2016)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Software and Reversible Systems: A Survey of Recent Activities

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

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

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

1 Introduction

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

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

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

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

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

2 Behavioural Types

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

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

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

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

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

3 Recovery

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

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

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

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

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

4 Reversibility and Object-Oriented Languages

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

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

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

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

4.1 Object Orientation and Data Structures

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Reversing Imperative Concurrent Programs

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

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

5.1 Language and Program State

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

$$
\begin{aligned}
\texttt{P} &::= \varepsilon \mid \texttt{S} \mid \texttt{P; P} \mid \texttt{P par P} \\
\texttt{S} &::= \texttt{skip I} \mid \texttt{X = E (pa,A)} \mid \texttt{if In B then P else Q end (pa,A)} \mid \\
&\quad\ \texttt{while Wn B do P end (pa,A)} \mid \texttt{begin Bn BB end} \mid \\
&\quad\ \texttt{call Cn n (pa,A)} \mid \texttt{runc Cn P end} \\
\texttt{BB} &::= \texttt{DV; DP; P; RP; RV}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{DV} &::= \varepsilon \mid \texttt{var X = v (pa,A); DV} & \texttt{DP} &::= \varepsilon \mid \texttt{proc Pn n is P end (pa,A); DP} \\
\texttt{RV} &::= \varepsilon \mid \texttt{remove X = v (pa,A); RV} & \texttt{RP} &::= \varepsilon \mid \texttt{remove Pn n is P end (pa,A); RP}
\end{aligned}
$$

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

5.2 Annotation, Inversion and Operational Semantics

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

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

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

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

(a) Executed annotated program

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

(b) Inverted program

Fig. 4. Identifier use example

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

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

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

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

5.3 Correctness of Annotation and Inversion

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

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

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

5.4 Simulator and Performance Evaluation

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

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

5.5 Application to Debugging

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

6 Reversible Debugger for Message Passing Systems

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

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

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

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

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

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

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

7 Control Theory

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

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

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

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

8 Conclusions

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

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

References

1. Axelsen, H.B., Glück, R.: Reversible representation and manipulation of constructor terms in the heap. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 96–109. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38986-3_9
2. Axelsen, H.B., Glück, R., Yokoyama, T.: Reversible machine code and its abstract processor architecture. In: Diekert, V., Volkov, M.V., Voronkov, A. (eds.) CSR 2007. LNCS, vol. 4649, pp. 56–69. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74510-5_9
3. Bagnoli, F., Rechtman, R., El Yacoubi, S.: Control of cellular automata. Phys. Rev. E **86**(6), 066201 (2012)
4. Bagnoli, F., Siljak, H.: Control of reversible cellular automata (2019, Manuscript in preparation)
5. Barbanera, F., Lanese, I., de'Liguoro, U.: A theory of retractable and speculative contracts. Sci. Comput. Program. **167**, 25–50 (2018)
6. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. **17**(6), 525–532 (1973)
7. Bishop, P.G.: Using reversible computing to achieve fail-safety. In: Proceedings the Eighth International Symposium on Software Reliability Engineering, pp. 182–191, November 1997
8. Carlsson, R., et al.: Core Erlang 1.0.3. Language specification (2004). https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf
9. Carothers, C.D., Perumalla, K.S., Fujimoto, R.: Efficient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul. **9**(3), 224–253 (1999)
10. Cassar, I., Francalanza, A., Mezzina, C.A., Tuosto, E.: Reliability and fault-tolerance by choreographic design. In: Francalanza, A., Pace, G.J. (eds.) Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017. EPTCS, vol. 254, pp. 69–80 (2017)
11. Castellani, I., Dezani-Ciancaglini, M., Giannini, P.: Concurrent reversible sessions. In: Meyer, R., Nestmann, U. (eds.) International Conference on Concurrency Theory, CONCUR 2017. LIPIcs, vol. 85, pp. 30:1–30:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2017)
12. Chen, S.-K., Fuchs, W.K., Chung, J.-Y.: Reversible debugging using program instrumentation. IEEE Trans. Softw. Eng. **27**, 715–727 (2001)
13. Cservenka, M.H., Glück, R., Haulund, T., Mogensen, T.Æ.: Data structures and dynamic memory management in reversible languages. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 269–285. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_19
14. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
15. de Vries, F., Pérez, J.A.: Reversible session-based concurrency in Haskell. In: Pałka, M., Myreen, M. (eds.) TFP 2018. LNCS, vol. 11457, pp. 20–45. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-18506-0_2
16. Dezani-Ciancaglini, M., Giannini, P.: Reversible multiparty sessions with checkpoints. In: Gebler, D., Peters, K. (eds.) Proceedings Combined 23rd International Workshop on Expressiveness in Concurrency and 13th Workshop on Structural Operational Semantics, EXPRESS/SOS 2016. EPTCS, vol. 222, pp. 60–74 (2016)

17. Elnozahy, E.N., Zwaenepoel, W.: Manetho: transparent rollback-recovery with low overhead, limited rollback, and fast output commit. IEEE Trans. Comput. **41**(5), 526–531 (1992)
18. Engblom, J.: A review of reverse debugging. In: System, Software, SoC and Silicon Debug, pp. 1–6. IEEE (2012)
19. Fink, M.: Time reversal of ultrasonic fields. I. Basic principles. IEEE Trans. Ultrason. Ferroelectr. Freq. Control **39**(5), 555–566 (1992)
20. Francalanza, A., Mezzina, C.A., Tuosto, E.: Reversible choreographies via monitoring in Erlang. In: Bonomi, S., Rivière, E. (eds.) DAIS 2018. LNCS, vol. 10853, pp. 75–92. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93767-0_6
21. Francalanza, A., Mezzina, C.A., Tuosto, E.: Towards choreographic-based monitoring. In: Ferreira, C., Lanese, I., Schultz, U., Ulidowski, I. (eds.) Reversible Computation: Theory and Applications. LNCS, vol. 12070. Springer, Heidelberg (2020)
22. Fredkin, E., Toffoli, T.: Conservative logic. Int. J. Theor. Phys. **21**, 219–253 (1982)
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
24. Gay, D., Aiken, A.: Language support for regions. In: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI 2001, pp. 70–80. ACM (2001)
25. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 370–384. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_26
26. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. J. Log. Algebr. Methods Program. **88**, 99–120 (2017)
27. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. Comput. Softw. **33**(3), 108–128 (2016)
28. Glück, R., Yokoyama, T.: Constructing a binary tree from its traversals by reversible recursion and iteration. Inf. Process. Lett. **147**, 32–37 (2019)
29. Haulund, T., Mogensen, T.Æ., Glück, R.: Implementing reversible object-oriented language features on reversible machines. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 66–73. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_5
30. Hoey, J.: Reversing imperative concurrent programs. Ph.D. thesis, University of Leicester (2020)
31. Hoey, J., Lanese, I., Nishida, N., Ulidowski, I., Vidal, G.: A case study for reversible computing: reversible debugging. In: Ferreira, C., Lanese, I., Schultz, U., Ulidowski, I. (eds.) Reversible Computation: Theory and Applications. LNCS, vol. 12070. Springer, Heidelberg (2020)
32. Hoey, J., Ulidowski, I.: Reversible imperative parallel programs and debugging. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 108–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_7
33. Hoey, J., Ulidowski, I., Yuen, S.: Reversing imperative parallel programs. In: Peters, K., Tini, S. (eds.) Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS. EPTCS, vol. 255, pp. 51–66 (2017)
34. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: Pérez, J.A., Tini, S. (eds.) Proceedings Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics, EXPRESS/SOS. EPTCS, vol. 276, pp. 69–86 (2018)

35. Hüttel, H., et al.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016)
36. Knuth, D.E.: The Art of Computer Programming: Fundamental Algorithms. Addison-Wesley, Boston (1998)
37. Landauer, R.: Irreversibility and heat generated in the computing process. IBM J. Res. Dev. **5**, 183–191 (1961)
38. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.-B.: Controlling reversibility in higher-order pi. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 297–311. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23217-6_20
39. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr tracer website. https://github.com/mistupv/tracer/
40. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr website. https://github.com/mistupv/cauder
41. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
42. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebr. Methods Program. **100**, 71–97 (2018)
43. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. In: Pérez, J.A., Yoshida, N. (eds.) FORTE 2019. LNCS, vol. 11535, pp. 167–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21759-4_10
44. Leeman Jr., G.B.: A formal approach to undo operations in programming languages. ACM Trans. Program. Lang. Syst. **8**(1), 50–87 (1986)
45. Letuchy, E.: Erlang at Facebook (2009). http://www.erlang-factory.com/conference/SFBayAreaErlangFactory2009/speakers/EugeneLetuchy
46. Mezzina, C.A., Pérez, J.A.: Reversible semantics in session-based concurrency. In: Proceedings of the 17th Italian Conference on Theoretical Computer 2016, Volume 1720 of CEUR Workshop Proceedings, pp. 221–226 (2016). CEUR-WS.org
47. Mezzina, C.A., Pérez, J.A.: Reversible sessions using monitors. In: Proceedings of the Ninth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016. EPTCS, vol. 211, pp. 56–64 (2016)
48. Mezzina, C.A., Pérez, J.A.: Causally consistent reversible choreographies: a monitors-as-memories approach. In: Vanhoof, W., Pientka, B. (eds.) Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, pp. 127–138. ACM (2017)
49. Mezzina, C.A., Pérez, J.A.: Reversibility in session-based concurrency: a fresh look. J. Log. Algebr. Methods Program. **90**, 2–30 (2017)
50. Neykova, R., Yoshida, N.: Let it recover: multiparty protocol-induced recovery. In: 26th International Conference on Compiler Construction, pp. 98–108. ACM (2017)
51. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 259–274. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_15
52. Perumalla, K.: Introduction to Reversible Computing. CRC Press, Boca Raton (2014)
53. Philippou, A., Psara, K.: Reversible computation in petri nets. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 84–101. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_6

54. Philippou, A., Psara, K., Siljak, H.: Controlling reversibility in reversing petri nets with application to wireless communications. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 238–245. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_15
55. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Logic Algebraic Program. **73**(1–2), 70–96 (2007)
56. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_18
57. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 95–110. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_6
58. Schordan, M., Oppelstrup, T., Jefferson, D., Barnes Jr., P.D., Quinlan, D.J.: Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In: SIGSIM-PADS 2016 (2016)
59. Schultz, U.P.: Reversible object-oriented programming with region-based memory management. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 322–328. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_22
60. Schultz, U.P., Axelsen, H.B.: Elements of a reversible object-oriented language. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 153–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_10
61. Siljak, H.: Reversibility in space, time, and computation: the case of underwater acoustic communications. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 346–352. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_25
62. Siljak, H.: Reversible computation in wireless communications. In: Ferreira, C., Lanese, I., Schultz, U., Ulidowski, I. (eds.) Reversible Computation: Theory and Applications. LNCS, vol. 12070. Springer, Heidelberg (2020)
63. Siljak, H., de Rosny, J., Fink, M.: Reversible hardware for acoustic wave time reversal. IEEE Commun. Mag. **58**(1), 55–61 (2020)
64. Siljak, H., Psara, K., Philippou, A.: Distributed antenna selection for massive MIMO using reversing Petri nets. IEEE Wirel. Commun. Lett. **8**(5), 1427–1430 (2019)
65. Siljak, H., Psara, K., Philippou, A.: Reversing Petri nets for resource management in wireless networks (2019, Manuscript in preparation)
66. Tofte, M., Talpin, J.-P.: Region-based memory management. Inf. Comput. **132**(2), 109–176 (1997)
67. Ulidowski, I.: IC1405 - Reversible Computation: extending horizons of computing - Memorandum of Understanding. https://e-services.cost.eu/files/domain_files/ICT/Action_IC1405/mou/IC1405-e.pdf
68. Undo Software: Undodb. Commercial reversible debugger. http://undo-software.com/
69. Vassor, M., Stefani, J.-B.: Checkpoint/Rollback vs causally-consistent reversibility. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 286–303. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_20
70. Vulov, G., Hou, C., Vuduc, R.W., Fujimoto, R., Quinlan, D.J., Jefferson, D.R.: The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In: WSC 2011 (2011)

71. Yokoyama, T., Axelsen, H.B., Glück, R.: Reversible flowchart languages and the structured reversible program theorem. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70583-3_22
72. Yokoyama, T., Axelsen, H.B., Glück, R.: Towards a reversible functional language. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 14–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29517-1_2
73. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Partial Evaluation and Program Manipulation, Proceedings, pp. 144–153. ACM (2007)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Simulation and Design of Quantum Circuits

Alwin Zulehner and Robert Wille(✉)

Institute for Integrated Circuits, Johannes Kepler University Linz, Linz, Austria
{alwin.zulehner,robert.wille}@jku.at

Abstract. Currently, there is an ongoing "race" to build the first practically useful quantum computer that provides substantial speed-ups for certain problems compared to conventional computers. In addition to the development of such devices, this also requires the development of automated tools and methods that provide assistance in the simulation and design of corresponding applications. Otherwise, a situation might be reached where we have powerful quantum computers but hardly any proper means to actually use them. This work provides an overview of corresponding solutions for the task of quantum circuit simulation, the task of quantum circuit design, as well as corresponding mapping tasks. The covered solutions utilise expertise on efficient data structures and algorithms gained in the design of conventional circuits and systems over the last decades. While the respective descriptions are kept brief and mainly convey the general ideas, references to further readings are provided for a more detailed treatment.

1 Introduction

In quantum computing, so-called quantum bits (i.e., qubits) serve as elementary information unit, which—in contrast to conventional bits—can not only be in one of its two orthogonal basis states (denoted $|0\rangle$ and $|1\rangle$ using Dirac notation), but also in superposition (i.e., a linear combination) of both [1]. Together with further quantum-physical phenomena such as entanglement (the state of a qubit might be influenced by the state of other qubits), this allows that the pure state of a quantum system composed of n qubits may represent a superposition of 2^n basis states and corresponding complex amplitudes—resulting in higher information density and computational power.

Well-known initial representatives of quantum algorithms following this powerful computation paradigm are Grover's search algorithm [2] and Shor's algorithm for integer factorisation in polynomial time [3]—both allowing to significantly outperform conventional machines. Recently, the application area of quantum algorithms has significantly broadened and provides efficient methods in areas like chemistry, solving systems of linear equations, physics simulations, machine learning, and many more [4–6].

These developments are also triggered by the fact that quantum computers are reaching feasibility since "big players" such as *IBM*, *Google*, *Microsoft*, and

© The Author(s) 2020
I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 60–82, 2020.
https://doi.org/10.1007/978-3-030-47361-7_3

Intel as well as specialised startups such as *Rigetti* and *IonQ* have entered this research field and are heavily investing in it [7–11]. In 2017, this led to the first quantum computers that are publicly available through cloud access by IBM. Since then, their machines have been used by more than 100,000 users, who have run more than 6.5 million experiments thus far. Recently, IBM followed with the presentation of their prototype towards a quantum computer for commercial use (a stand-alone quantum computer to be operated outside of their labs)—the *IBM Q System One* presented in January 2019 at CES [12].

Since currently available quantum computers are still limited in the number of qubits, gate fidelity, as well as coherence time, they are classified as *Noisy Intermediate Scale Quantum* (NISQ [5]) devices that will only be able to successfully run some of the quantum algorithms outlined above (due to their limitations). In fact, unveiling the full potential of quantum computing requires—besides further reduction of error rates and improvement of coherence time—error-correcting codes where each logical qubit in a computation is realised by several (up to several hundreds) of physical qubits—eventually resulting in *fault-tolerant* devices that are capable of conducting very deep computations on a large number of qubits and with perfect accuracy [13,14].

In addition to these accomplishments and prospects, also the development of automated tools and methods that provide assistance in the simulation and design of corresponding applications is required. In this regard, the task of quantum circuit simulation, the task of quantum circuit design, as well as corresponding mapping tasks are important. Since modelling (arbitrary) quantum states on conventional machines requires exponential overhead and many design problems are of exponential nature, straightforward solutions for these tasks will not scale to relevant problem sizes. Hence, clever data-structures and algorithms are required that allow for efficient solutions (at least) in certain cases. Otherwise, we are approaching a situation where we might have powerful quantum computers but hardly any proper means to actually use them.

This work provides an overview on solutions which have been developed for these tasks and utilise expertise on efficient data structures and algorithms gained in the design automation community over the last decades for conventional circuits and systems. To this end, the simulation of quantum circuits, their design, as well as technology mapping (compiling) are covered and discussed from a design automation perspective. The reviewed solutions often yield improvements of several orders of magnitude compared to the current state of the art (regarding runtime and corresponding design objectives)—showing the tremendous available potential.

The overview is thereby structured as follows: First, Sect. 2 provides a background on quantum computing. Afterwards, Sect. 3, Sect. 4, and Sect. 5 sketch the developed methods for the considered design tasks, i.e., quantum-circuit simulation, the design of Boolean components occurring in quantum algorithms, as well as mapping quantum circuits to real hardware (including references to further reading for a more detailed treatment). Finally, Sect. 6 concludes the paper.

2 Background on Quantum Computing

Quantum computations operate on *qubits*—two-level quantum systems that can be combined into n-qubit systems. The state of a qubit is given by a linear combination (i.e., a *superposition*) of these basis states $|\varphi\rangle = \alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$, where the complex *amplitudes* α_0 and α_1 satisfy $\alpha_0\alpha_0^* + \alpha_1\alpha_1^* = 1$.

The joint state of n qubits (also denoted as the system's *wave function*) is contained in the *tensor product* of n two-dimensional Hilbert spaces—the 2^n-dimensional Hilbert space spanned by the basis $|0\rangle, \ldots, |2^n - 1\rangle$. Hence, a *superposition* of all computational basis states may need up to 2^n complex-valued parameters—appearing as the amplitudes of the unit-norm state vector.

Definition 1. *Consider a quantum system composed of n qubits. Then, all possible states of the system are of the form*

$$|\varphi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x \cdot |x\rangle\,, \text{ where } \sum_{x\in\{0,1\}^n} \alpha_x\alpha_x^* = 1 \text{ and } \alpha_x \in \mathbb{C}.$$

The state $|\varphi\rangle$ *can be also represented by a column vector* $\varphi = [\varphi_i]$ *with* $0 \le i < 2^n$ *and* $\varphi_i = \alpha_x$*, where* $nat(x) = i$*.*

Quantum states cannot be directly observed. To extract (partial) information from quantum states in the form of conventional bits, one performs a *measurement operation.* In contrast to conventional computers, this measurement modifies the quantum state. In the process of measurement, the quantum state non-deterministically collapses to one of these basis states where the probability of each outcome reflects the proximity to the respective basis state. More precisely, measuring a one-qubit state $\alpha_0 \cdot |0\rangle + \alpha_1 \cdot |1\rangle$ (with $\alpha_0\alpha_0^* + \alpha_1\alpha_1^* = 1$) changes the state to $|0\rangle$ or $|1\rangle$ with probabilities $\alpha_0\alpha_0^*$ and $\alpha_1\alpha_1^*$, respectively.

Example 1. *Consider a quantum system composed of* $n = 3$ *qubits* q_0*,* q_1*, and* q_2 *that assumes the state* $|\varphi\rangle = |q_0q_1q_2\rangle = \frac{1}{2} \cdot |010\rangle + \frac{1}{2} \cdot |100\rangle - \frac{1}{\sqrt{2}} \cdot |110\rangle$*. Then, the state vector of the system is given by*

$$\varphi = \left[0, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, -\frac{1}{\sqrt{2}}, 0\right]^T.$$

Measuring the system yields basis states $|010\rangle$*,* $|100\rangle$*, and* $|110\rangle$ *with probabilities* $\frac{1}{4}$*,* $\frac{1}{4}$*, and* $\frac{1}{2}$*, respectively. Measuring only qubit* q_0 *collapses* q_0 *into basis state* $|0\rangle$ *and* $|1\rangle$ *with probabilities* $\frac{1}{4}$ *and* $\frac{1}{4} + \frac{1}{2} = \frac{3}{4}$*, respectively—changing the state of the system either to* $|\varphi'\rangle = |010\rangle$ *or to* $|\varphi''\rangle = \frac{1}{\sqrt{3}} \cdot |100\rangle - \sqrt{\frac{2}{3}} \cdot |110\rangle$*.*

Aside from measurements, quantum computers apply quantum operations to a fixed set of qubits, altering the joint state of the qubits in a reversible fashion. These operations are described by unitary matrices of size $2^n \times 2^n$. Simple quantum operations (also denoted *gates*) are defined over one or two qubits only. Mathematically speaking, the resulting $2^n \times 2^n$ matrix can then

be computed as the Kronecker product of the matrix representing the gate's operation and a large identity matrix.

Commonly used quantum gates for generating a superposition (the Hadamard operation H), inverting a quantum state (X), and applying phase shifts by −1 (Z), are respectively defined as

$$\mathrm{H} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \mathrm{NOT} = \mathrm{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \mathrm{Z} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

Two-qubit gates can couple pairs of qubits and are represented by 4×4 unitary matrices. By applying arbitrary two-qubit gates to different pairs of qubits, it is possible to effect any 2^n-dimensional unitary, i.e., attain universal quantum computation (each quantum functionality can be realised with those gates). It is common to allow a variety of one-qubit gates but limit two-qubit gates, e.g., to CNOT gates:

$$\mathrm{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The two-qubit CNOT gate can also be defined by its action $|x\ y\rangle \mapsto |x\ x \oplus y\rangle$, where $\oplus$ represents the *exclusive-or* (XOR) operation, the unmodified qubit x is called *control*, and the other bit is called *target*.

Quantum circuits [1] are used as proper description means for a finite sequence of "small" gates that cumulatively enact some unitary operator U and, given an initial state $|\varphi\rangle$ (which is usually the basis state $|0\dots0\rangle$), produce a final state vector $|\varphi'\rangle = |U\varphi\rangle$. Hence, a quantum gate does not represent a physical entity (like in the conventional realm), rather an operation that is applied to a set of qubits.

Definition 2. *In quantum circuits, the qubits are vertically aligned in a circuit diagram, and the time axis (read from left to right) is represented by a horizontal line for each qubit. Boxes on the time axis of a qubit (or enclosing several qubits) indicate gates to be applied.*[1] *Note that measurement also counts as quantum operation in this context. Control qubits are indicated by* $\bullet$ *and are connected to the controlled operations by a single line.*

Example 2. *Figure 1 shows a quantum circuit. The circuit contains two qubits,* q_0 *and* q_1*, which are both initialised with basis state* $|0\rangle$*. First, a Hadamard operation is applied to qubit* q_0*, which is represented by a box labelled* H*. Then, a* CNOT *operation is conducted, where* q_0 *is the* control qubit *(denoted by* $\bullet$*) and* q_1 *is the target qubit (denoted by* $\oplus$*). Eventually, qubit* q_0 *is measured as indicated by the meter symbol.*

When two gates are applied on the same qubits in sequence, the resulting operation is represented by the matrix product of gate matrices. When an

[1] Note that an X gate may also be denoted by $\oplus$.

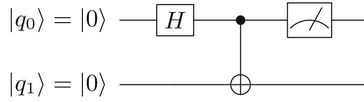


Fig. 1. Quantum circuit.

m-qubit gate A and an n-qubit gate B are applied in parallel (on different qubits), the resulting operation is represented by the *Kronecker product* $A \otimes B$ of two matrices.

Example 3. *Consider again the quantum circuit shown in Fig. 1. The resulting state $|\varphi'\rangle$ (before measurement) is determined by multiplying the respective unitary matrices to the state vector. Since the Hadamard gate shall only affect q_0, the Kronecker product of* H *and the identity matrix I_2 is formed, i.e.,*

$$H \otimes I_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}.$$

Then, $|\varphi'\rangle$ is determined by

$$|\varphi'\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \cdot \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

As can be seen, the two gates entangle the qubits q_0 and q_1—generating a so-called Bell state $|\varphi'\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$*. Measuring qubit q_0 collapses its superposition into one of the two basis states. Since q_0 and q_1 are entangled, q_1 collapses to the same basis state.*

3 Quantum-Circuit Simulation

Since physical realisations of quantum computers are limited in their availability, their number of qubits, their gate fidelity, and coherence time, quantum-circuit simulators running on conventional machines are required for many tasks. From a user's perspective, possible applications (or at least their prototypes) for quantum computers are usually first evaluated through simulators that serve as temporary substitute. Moreover, simulation can be adapted to circuit equivalence-checking and other functional verification tasks useful for circuit designers [15–17]. Simulation also plays an important role for designers of quantum systems, e.g., to foster the development of error-correcting codes. Besides that, the urgent need of verifying quantum hardware might be conducted (at least some of the required verification tasks) by comparing runs on these machines to simulation

outcome [18,19]. Ultimately, quantum-circuit simulation capabilities provide an estimate on *quantum supremacy* [18] as well as to identify classes of circuits where no quantum speed-up is reachable (i.e., in case these circuits can be simulated efficiently on a conventional machine). In all these scenarios, simulators may give additional insights since, e.g., the precise amplitudes of a quantum state are explicitly determined (while they are not observable in a real quantum computer).

However, quantum-circuit simulation in general constitutes a computationally very complex task since each quantum gate and each quantum state is eventually represented by a unitary matrix or state vector that grows exponentially with the number of qubits. In fact, each quantum operation applied to a quantum state composed of n qubits requires multiplying a $2^n \times 2^n$-dimensional matrix with a 2^n-dimensional vector.[2] This constitutes a serious bottleneck, which prevents the simulation of many quantum applications and, by this, the evaluation of their potential. In fact, the array-like representation of the state vector in current state-of-the-art simulators limits the number of qubits to be simulated to approximately 30 on a modern computer (and to 50 when considering supercomputers with petabytes of distributed memory) [20].

This section presents a complementary simulation approach that aims for overcoming this memory bottleneck (based on [21]). To this end, dedicated *Decision Diagrams* (DDs) are developed, which reduce the memory requirements by representing redundancies in the occurring vectors and matrices by means of shared nodes. This allows gaining significant improvements compared to straightforward realisations (relying on array-like representations) in many cases—often reducing the simulation time from several hours or days to seconds or minutes.[3]

3.1 General Idea

The general idea of the presented complementary approach is to exploit redundancies in the 2^n-dimensional vectors representing quantum states. To this end, decision diagram techniques (similar to those from the conventional realm) are employed. More precisely, a given state vector with entries being complex numbers is decomposed into sub-vectors. To this end, consider a quantum system with qubits $q_0, q_1, \ldots q_{n-1}$, whereby without loss of generality q_0 represents the most significant qubit. Then, the first 2^{n-1} entries of the corresponding state vector represent the amplitudes for the basis states with q_0 set to $|0\rangle$; the other entries represent the amplitudes for states with q_0 set to $|1\rangle$. This decomposition is represented in a decision diagram structure by a node labelled q_0 and two successors leading to nodes representing the sub-vectors. The sub-vectors are

[2] Note that different simulation approaches exist that do not compute the complete final state vector, and that it is usually not necessary to represent the exponentially large matrix explicitly. However, this does not decrease the exponential complexity.

[3] Note that previous DD-based simulators (e.g., *QuIDDPro* [22]) did not get established due to their limited applicability (i.e., they provide improvements in rather few cases).

recursively decomposed further until vectors of size 1 (i.e., a complex number) result. This eventually represents the amplitude α_i for the basis state and is given by a terminal node. During these decompositions, equivalent sub-vectors are represented by the same node—allowing for sharing and, hence, a reduction of the memory complexity. An example illustrates the idea.

Example 4. *Consider a quantum system with $n = 3$ qubits situated in a state given by the following vector:*

$$\varphi = \left[0, 0, \frac{1}{2}, 0, \frac{1}{2}, 0, -\frac{1}{\sqrt{2}}, 0\right]^T.$$

Applying the decompositions described above yields a decision diagram as shown in Fig. 2a. The left (right) outgoing edge of each node labelled q_i points to a node representing the sub-vector with all amplitudes for the basis states with q_i set to $|0\rangle$ ($|1\rangle$). Following a path from the root to the terminal node yields the respective entry. For example, following the path highlighted bold in Fig. 2a provides the amplitude for the basis state with $q_0 = |1\rangle$ (right edge), $q_1 = |1\rangle$ (right edge), and $q_2 = |0\rangle$ (left edge), i.e., $-\frac{1}{\sqrt{2}}$ which is exactly the amplitude for basis state $|110\rangle$ (seventh entry in the vector). Since some sub-vectors are equal (e.g., $\left[\frac{1}{2}, 0\right]^T$ represented by the left node labelled q_2), sharing is possible.

However, even more sharing is possible since sub-vectors often differ in a common factor only. This is additionally exploited in the proposed representation by denoting common factors of amplitudes as weights attached to the edges of the decision diagram. Then, the value of an amplitude for a basis state is determined by following the path from the root to the terminal, and additionally multiplying the weights of the edges along this path. Again, an example illustrates the idea.

Example 4 (continued)**.** *As can be seen, the sub-vectors represented by the nodes labelled q_2 (i.e., $\left[\frac{1}{2}, 0\right]^T$ and $\left[-\frac{1}{\sqrt{2}}, 0\right]^T$) differ in a common factor only.*

In the decision diagram shown in Fig. 2b, both sub-trees are merged. This is possible since the corresponding value of the amplitudes is now determined not by the terminals, but the weights on the respective paths. As an example, consider again the path highlighted bold representing the amplitude for the basis state $|110\rangle$. Since this path includes the weights $\frac{1}{2}$, 1, $-\sqrt{2}$, and 1, an amplitude of $\frac{1}{2} \cdot 1 \cdot (-\sqrt{2}) \cdot 1 = -\frac{1}{\sqrt{2}}$ results.

Note that, of course, various possibilities exist to factorise an amplitude. Hence, a normalisation is applied which assumes the left edge to inherit a weight of 1. More precisely, the weights w_l and w_r of the left and right edge are both divided by w_l and this common factor is propagated upwards to the parents of the node. If $w_l = 0$, the node is normalised by propagating w_r upwards to the parents of the node.

The idea used for representing state vectors by means of DDs can be extended to also represent unitary matrices. Here, each DD-node has four successors that

(a) Without edge weights.

(b) With edge weights.

Fig. 2. DD-based representation of state vectors.

represent the four quadrants of the sub-matrix. Having description means for state vectors and unitary matrices (describing the functionality of gates) it is left to provide algorithms for matrix-vector multiplication as well as for measurement. Fortunately, all these operations can be directly employed on the DDs and without the need of explicitly representing the underlying exponentially large entities. For further details we refer to [21].

3.2 Resulting Approaches

Following the general idea outlined above leads to a simulation approach that scales polynomially with the size of the DD representing the state vector. Since the DD often remains rather compact, significant improvements can be observed compared to straightforward Schrödinger-style simulators as well as to previous DD-based simulators in many cases—even though these techniques have been heavily optimised over the last decade and utilise multiple CPU-cores to reduce simulation time (while the proposed approach utilises a single core only). More precisely, the approach proposed in [21] is capable of (1) simulating quantum computations for more qubits than before, (2) in significantly less run-time, and (3) on a regular Desktop machine.

For further details on the basic ideas and required algorithms of the DD-based simulator we refer to [21,23]. Moreover, [21] shows that for many cases, the simulation time can be reduced from several days to just a few seconds or minutes. This initial version of a DD-based simulator did not only lead to a significant improvement compared to the current state of the art, but has also received significant acknowledgement by the community—triggering further optimisations as done for array-based Schrödinger-style simulators for more than a decade.

Using DDs for representing occurring vectors and matrices, the complexity of multiplications depends on the size (i.e, the number of nodes) of the respective operands in DD-based simulation. Together with the fact that the DDs for the usually considered gate matrices are linear in size (with respect to the number

of qubits), this implies that it might be beneficial to combine gate operations before applying them to the state vector. In [24], strategies are described for combining operations that allow improving the initial version of the proposed DD-based simulator significantly—up to several orders of magnitude when exploiting application-specific knowledge.

Enormous improvements compared to the state of the art as described above obviously require an efficient implementation of the underlying DD-package—especially for handling the occurring complex numbers. By providing such techniques—in joint consideration of implementation techniques for decision diagrams in the conventional domain developed decades ago—the development of a powerful DD-package for the quantum domain was leveraged in [25]. The evaluation conducted in [25] showed that complex numbers can be handled much more efficiently than in previous implementations and that decision diagrams for established quantum functionality is constructed in significantly less run-time (up to several orders of magnitude). Presumably, this performance boost can be easily passed to DD-based methods for other design automation tasks like synthesis [26,27] or verification [15–17], just by incorporating this new package.

Since handling complex numbers is crucial in DDs for quantum computation (especially when occurring as edge weights), the resulting trade-off between accuracy and compactness has been thoroughly discussed and evaluated in [28]. Since this trade-off requires fine-tuning of parameters on a case-by-case basis and might still yield useless results, an algebraic decision diagram is proposed in [28] to overcome this issue. The proposed algebraic representation guarantees perfect accuracy while remaining compact (all redundancies that are actually present are detected)—with moderate overhead in many cases.

All the endeavours listed above have been implemented in C/C++ and made publicly available at http://iic.jku.at/eda/research/quantum_simulation. Besides that, a stand-alone version of the developed DD-package is available at http://iic.jku.at/eda/research/quantum_dd. Together with the significant improvements gained compared to the state of the art, this did not only result in acknowledgement inside the academic community, but also received interest from big players in the field. More precisely, the developed simulation approach has been acknowledged with a *Google Research Faculty Award* and has recently been officially integrated into IBM's SDK *Qiskit*. This further emphasises the potential of DD-based design methods in the quantum domain—hopefully leading to as powerful DD-based methods as taken for granted in the conventional domain today. Questions on whether hybrid approaches are possible or whether concurrent approaches as well as approximation schemes can be exploited remain open issues for future work. First results towards these questions are provided in [29,30].

4 Design of Boolean Components for Quantum Circuits

Estimating resource requirements of quantum algorithms (i.e., the number of required qubits and run-time on quantum computers), their simulation, or their

execution on real hardware requires compiling quantum algorithms containing high-level operations (e.g., modular exponentiation in Shor's algorithm) into quantum circuits composed of elementary gates available on the considered target architecture. Thereby, quantum circuits composed of gates with multiple control qubits (multiple-controlled qubit gates) are usually considered since they (1) describe a rather low-level but still technology independent description of the algorithm, (2) can be directly handled by most simulators, and (3) are usually utilised as input for technology mapping algorithms (which will be covered in the next section).

For the "quantum part" of an algorithm, a decomposition into multiple-controlled qubit gates is usually inherently given by the algorithm, by using common building blocks like a *Quantum Fourier Transform* (QFT [31]), or determined by hand. However, this is different for large Boolean components that are contained in many quantum algorithms, e.g., the modular exponentiation in Shor's algorithm for integer factorisation [3] or a Boolean description of the database that is queried in Grover's algorithm [2].

Even though the functionality of the Boolean components can be described in the conventional domain, corresponding design methods cannot be utilised since the inherent reversibility of quantum computations has to be considered. In fact, determining circuits composed of reversible gates only, requires dedicated *reversible-circuit synthesis* approaches. To manage the complex functionality of Boolean components, they are usually split into several (non-)reversible parts [32]. However, these resulting non-reversible sub-functions have to be *embedded* into reversible ones to ensure the desired unique mapping from inputs to outputs—a task that can either be conducted explicitly or implicitly. This embedding process requires adding several so-called *ancillary qubits*, which shall be kept as small as possible since qubits are a highly limited resource. Besides that, T-count and T-depth of the synthesised reversible circuits serve as cost metric to compare different approaches that yield circuits with an equal (or at least a close-to equal) number of qubits.

This section focuses on the *functional* design flow for synthesising Boolean components (where the reversible function resulting from an explicit embedding step is passed to synthesis algorithm) since it yields circuits with a moderate number of qubits (often the minimum). Investigating this problem from a design automation perspective allows developing efficient methods utilising the decision diagrams introduced in the context of simulation (cf. Sect. 3) [33–35]. However, there is even more (yet) unused potential that allows synthesising cheaper circuits, yields better scalability, and even reduces the number of required qubits below what is currently considered as the minimum (for certain cases)—significantly improving the current state of the art.

4.1 One-Pass Design of Reversible Circuits

Despite using efficient description means like DDs for functional synthesis, the currently established design flow still suffers from the need to conduct embedding

and actual synthesis separately—a major drawback that prohibits the exploitation of a huge degree of freedom since embedding is not necessarily conducted in a fashion, which suits the following synthesis step. To overcome this drawback, the work [36,37] introduced a completely new design flow that combines functional synthesis and the embedding to a *one-pass design flow*. This generic flow is not bound to a certain functional synthesis approach and—for the first time—exploits the available degree of freedom to significantly increase scalability and to reduce the costs of the synthesised circuit while keeping the number of required qubits at the minimum.

In the established flow, an individual step is required that embeds the non-reversible function to be synthesised into a reversible one. Thereby, $k = \lceil \log_2 \mu(p_1) \rceil$ further so-called garbage outputs are added (assuming that the most frequent output pattern p_1 occurs $\mu(p_1)$ times) and the additional rows and columns of the truth table are assigned such that a unique mapping from inputs and outputs results [33]. Passing a non-reversible function directly to a functional reversible-circuit synthesis approach will fail, since several input combinations shall be mapped to the same output combination. This can be avoided in two ways:

- Following the *exact* solution guarantees to result in a circuit requiring the minimum number of qubits. The general idea is to add k further variables to the function description (e.g., a DD), but keep all additional entries in the function *don't care*—allowing to exploit the available degree of freedom of their assignment (which does not matter as long as a reversible function results). Having these additional variables allows conducting synthesis (almost) as usual. During synthesis, the *don't cares* are inherently assigned (1) in a way that suits best to the synthesis algorithm, and (2) such that a reversible function results (since only reversible gates are added to the circuit).
- Following the *heuristic* solution does not necessarily result in a circuit requiring the minimum number of qubits, but still bounded. The general idea is to conduct synthesis without embedding. Whenever an error is encountered during synthesis (i.e., synthesis cannot proceed due to the missing embedding step), the function to be synthesised is modified such that the algorithms can continue. Since this obviously results in a circuit different to the intended one, the modifications of the function are stored on so-called buffer-lines (at most one buffer line is required for each variable of the function). After synthesis finishes, these modifications are reverted by a single CNOT gate for each buffer line.

  The advantage of the heuristic approach is that no additional variables are added to the function description (as done in the usual functional design flow and the exact one-pass design). Hence, this heuristic approach is even more scalable that the exact solution since the function description remains smaller.

Example 5. *Consider a function* $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ *with* n *inputs and* m *outputs and assume that the most frequent output pattern occurs* $\mu(p_1)$ *times. Then,*

following the exact solution, the f is enriched by $k = \lceil \log_2 \mu(p_1) \rceil$ further outputs to make all output patterns distinguishable. Hence, the synthesis is conducted on a function with $\max(n, m + k)$ *variables—like in the established design flow. However, the additional entries in the truth table remain* don't care *initially and are assigned 0 or 1 during synthesis as suitable.*

Instead, the heuristic solution conducts synthesis directly on f and, hence $\max(n, m)$ *variables. The modifications made to f during synthesis require at most* $\min(n, m)$ *buffer lines—resulting in a quantum circuit with at most $n + m$ qubits.*

The evaluations provided in [36] show the advantages of the one-pass design flow (which can be also applied to other functional synthesis approaches) compared to the conventional two-stage design flow. Besides substantial speedups compared to the state-of-the-art design flow, the T-count is reduced by several orders of magnitude in most cases—clearly outperforming the currently established functional design flow for reversible circuits where embedding and synthesis are conducted separately. For further details, we refer to [36].

4.2 Exploiting Coding Techniques

The proposed one-pass design flow can be enriched with the idea of exploiting coding techniques in order to reduce the number of variables that have to be considered during synthesis [38].[4] This idea is based on the fact, that the output patterns in non-reversible functions are not uniformly distributed—leading to a situation where some patterns require many additional outputs while others require only a few. Hence, several garbage outputs are required only for certain output patters. Avoiding this overhead provides significant potential for improving synthesis. In fact, employing a variable-length code allows realising any non-reversible function with a single ancillary qubit only—allowing conducting synthesis on significantly fewer variables than before [39]. The key idea is to represent frequently occurring output patterns (which require more garbage outputs) with a smaller number of variables. Vice versa, less frequently occurring patterns (which require less garbage outputs) are represented with a larger number of variables. In other words, coding techniques are utilised in order to encode the desired function with a variable-length code in which the length of the code word for an output pattern p_i is indirectly proportional to the number $\mu(p_i)$ of times the pattern occurs. An example illustrates that.

Example 6. *Consider the Boolean function shown in Table 1a and its distribution of the output patterns as shown in Table 1b. Following, e.g., the exact one-pass design flow outlined above results in a function with 5 inputs/outputs since the most frequent output pattern $p_1 = 010$ occurs four times and, thus, requires two garbage outputs. However, using a variable-length code as shown in*

[4] Note that exploiting coding techniques is also possible in the original design flow composed of an embedding and a synthesis step.

Table 1. Variable-length encoding for one-pass design.

(a) Orig. function

x_0	x_1	x_2	y_0	y_1	y_2
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	0	1	1
1	0	1	0	1	0
1	1	0	0	1	0
1	1	1	0	0	1

(b) Output patterns

i	p_i	$\mu(p_i)$
1	010	4
2	100	2
3	001	1
4	011	1
5	000	0
6	101	0
7	110	0
8	111	0

(c) Encoding

i	p_i	$c(p_i)$
1	010	0 - -
2	100	1 0 -
3	001	1 1 0
4	011	1 1 1

(d) Encoded function

x_0	x_1	x_2	y_0	y_1	y_2
0	0	0	0	-	-
0	0	1	0	-	-
0	1	0	1	0	-
0	1	1	1	0	-
1	0	0	1	1	1
1	0	1	0	-	-
1	1	0	0	-	-
1	1	1	1	1	0

Table 1c allows reducing the number of required qubits. There, the most frequent output pattern is encoded by $c(p_1) = 0$*. Since this pattern requires two garbage outputs, in total* $1+2 = 3$ *outputs are required.*[5] *The second most frequent output pattern* $p_2 = 100$ *is encoded by* $c(p_2) = 10$*. Since this pattern occurs only twice, one garbage output is required—again resulting in* $2 + 1 = 3$ *outputs. The patterns* p_3 *and* p_4 *are encoded by* $c(p_3) = 110$ *and* $c(p_4) = 111$*, respectively. Here, no garbage outputs are required. The remaining patterns (*p_5 *to* p_8*) do not have to be encoded, since they never occur. Overall, this yields an (encoded) reversible function which embeds* f *as shown in Table 1d and is composed of a total of 3 inputs/outputs only—two qubits fewer than without using coding.*

The code is computed by generating a *Pseudo-Huffman* tree: Starting with terminal nodes—one for each output pattern with $\mu(p_i) > 0$ (no code has to be assigned to output patterns that do not occur)—with attached weights representing the number of respectively required garbage outputs (i.e., $\lceil \log_2 \mu(p_i) \rceil$), the *Pseudo-Huffman* tree is then generated by repeatedly combining the two nodes a and b with the smallest attached weights $w(a)$ and $w(b)$ to a new node c with weight $w(c) = \max(w(a), w(b)) + 1$ until a single node results. The weight of such a node $w(c)$ then gives the number of outputs required to represent all combined output patterns uniquely, i.e., one additional variable is required (aside from $\max(w(a), w(b))$) to distinguish between a and b.

Example 7. *Consider the distribution of the output patterns as shown in Table 1b. Determining the Pseudo-Huffman code starts with the nodes* v_1*,* v_2*,* v_3*, and* v_4*—one for each output pattern* p_i *with* $\mu(p_i) > 0$*. These nodes are shown at the bottom of Fig. 3. The weights are drawn inside the respective nodes. The weight of node* v_1 *is* $w_1 = k_1 = 2$*, because output pattern* $p_1 = 010$ *requires two garbage outputs. The weights of the nodes representing* p_2*,* p_3*, and* p_4 *are 1, 0, and 0, respectively. In a first step, the nodes* v_3 *and* v_4 *(both have weight 0) are combined. The resulting node* v_5 *has a weight of* $w_5 = \max(0, 0) + 1 = 1$*. Next, the two nodes with weight 1 (i.e.,* v_2 *and* v_5*) are combined. The resulting node*

[5] The garbage outputs are represented by a dash, since they represent *don't care* values (as long as it is ensured that the resulting function is reversible).

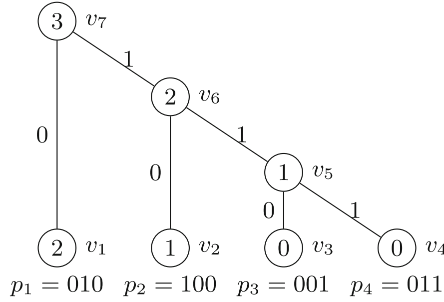


Fig. 3. Huffman tree for the function from Table 1a.

v_6 has a weight of $w_6 = \max(1,1) + 1 = 2$. Finally, the two remaining nodes are combined to a new node v_7 with weight $w_7 = \max(2,2) + 1 = 3$—eventually resulting in the tree shown in Fig. 3.

After generating the Pseudo-Huffman tree, the overall number of variables that are required to realise the encoded function is given by the weight of the root node of the tree. The resulting code is inherently given by the structure of the Pseudo-Huffman tree. In fact, each path from the root node to a leaf node represents a code word, where taking the left (right) edge implies a 0 (1).

Example 7 (continued)**.** *Since the root node has a weight of 3, three variables are required to realise the encoded function (without encoding,* $\max(3, 3+2) = 5$ *variables would be required). The path from the root node to the leaf node v_2 (which represents output pattern p_2) traverses the right edge of the root node v_7 as well as the left edge of v_6. Consequently, $c(p_2) = 10$ encodes $p_2 = 100$. Since v_2 has weight $w_2 = 1$, one output is used as garbage output in this case. Accordingly, code words for all other output patterns are determined—eventually resulting in the code shown in Table 1c. Dashes again represent* don't cares.

Following this idea, at most $n + 1$ qubits—instead of $\max(n, m + \lceil \log_2 \mu(p_1) \rceil)$—are required to embed any non-reversible function with n inputs. Concerning the design of Boolean components contained in quantum algorithms, the encoded outputs can be handled (1) *locally* where decoders are required for each sub-component that again increase the number of qubits to $\max(n, m + \lceil \log_2 \mu(p_1) \rceil)$, or (2) *globally* where subsequent components that are capable of handling encoded inputs allow remaining at $n + 1$ qubits.

Incorporating the idea of utilising coding techniques into the one-pass design flow introduced above unveils even more potential. In fact, it allows exploiting an even larger degree of freedom since the values of the garbage outputs are basically *don't care* (except the restriction that a reversible function has to be realised)—while still guaranteeing to synthesise a circuit that uses the minimum number of qubits (or even below that minimum if no decoding is required afterwards). This degree of freedom allows for synthesising circuits with significantly smaller T-count [38].

5 Mapping Quantum Circuits to NISQ Devices

In order to use currently developed *Noisy Intermediate-Scale Quantum* (NISQ) devices, the quantum algorithm to be executed has to be properly mapped to these devices such that their underlying physical constraints are satisfied (this is one part of the overall compilation task). To this end, it is assumed that the considered quantum algorithm has already been translated into a quantum circuit composed of multiple-controlled one-qubit gates. For the "quantum part" of the algorithm, this is often inherently given (e.g., by using components for which such translations are known) or done by hand. For the "Boolean part" of the algorithm, a gate-level description is often gained by *reversible circuit synthesis*, as discussed in the previous section.

Then, mapping quantum circuits to NISQ devices requires the consideration of two aspects. First, the occurring gates have to be decomposed into elementary operations provided by the target device—usually a single two-qubit gate as well as a broader variety of one-qubit gates to gain a universal gate set. Second, the *logical* qubits of the quantum circuit have to be mapped to the *physical* qubits of the target device while satisfying the so-called *coupling-constraints* given by the respective device. Since not all physical qubits are coupled directly with each other (due to missing physical connections), two-qubit gates can only be applied to selected pairs of physical qubits. Since it is usually not possible to determine a mapping such that all coupling-constraints are satisfied throughout the whole circuit, the mapping has to change dynamically. This is achieved by inserting additional gates, e.g., realising SWAP operations, in order to "move" the logical qubits to other physical ones.

While there exist several methods to address the first issue, i.e., how to efficiently decompose multiple-controlled one-qubit gates into elementary operations (see [40,41]), there is only few work on how to efficiently satisfy the coupling-constraints of real devices. Although there are similarities with recent work on nearest-neighbour optimisation of quantum circuits as proposed in [42–45], they are not applicable since simplistic architectures with 1-dimensional or 2-dimensional layouts are assumed which have a fixed coupling (all adjacent qubits are coupled) that does not allow modelling all current NISQ devices.

This section covers the mapping of the logical qubit of a quantum circuit to the physical ones of a NISQ device from a design automation perspective. Thereby, *IBM Q devices* are considered as representatives for NISQ devices to discuss the occurring challenges in detail, as well as to describe the proposed solutions. IBM's approach has been chosen, since it provides the first publicly available quantum devices (available since 2017) that can be accessed by everyone (not only academics) through cloud access. Moreover, their coupling-constraints are described more flexibly than those of other companies—allowing to map their coupling-constraints to IBM's model as well.

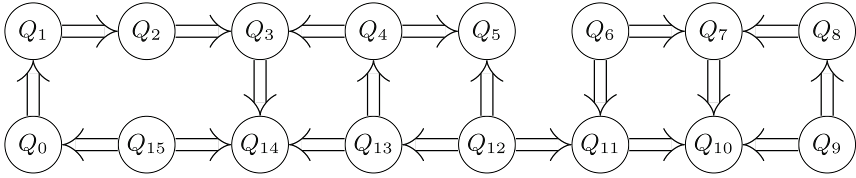


Fig. 4. IBM Q 16 Rueschlikon V1.0.0 (IBM QX3) [46].

5.1 Considered Problem

While one-qubit gates can be applied without limitations in IBM's devices, the physical architecture of the respectively developed quantum computers—usually a linear or rectangular arrays of qubits—limits two-qubit gates to neighbouring qubits that are connected by a superconducting bus resonator. In IBM's devices that use cross-resonance interaction as the basis for CNOT gates, the frequencies of the qubits also determine the direction of the gate (i.e., determining which qubit is the control and which is the target). The possible CNOT gates are captured by so-called coupling maps [46], giving a very flexible description means to specify the *coupling-constraints* of a certain quantum device. Figure 4 shows the coupling map of the IBM QX3 device. Physical qubits are visualised with nodes and a directed edge from physical qubit Q_i to physical qubit Q_j indicates that a CNOT with control qubit Q_i and target qubit Q_j can be applied.

To satisfy the coupling-constraints, one has to map the n logical qubits $q_0, q_1, \ldots, q_{n-1}$ of the decomposed circuit to the $m \geq n$ physical qubits $Q_0, Q_1, \ldots, Q_{m-1}$ of the considered quantum device such that all coupling-constraints given by the corresponding coupling map are satisfied. Unfortunately, it is usually not possible to find a mapping such that the coupling-constraints are satisfied throughout the whole circuit (this is already impossible if the number of other qubits, a logical qubit interacts with, is larger than the maximal degree of the coupling map). More precisely, the following problems—using $CNOT(q_c, q_t)$ to describe a CNOT gate with control qubit q_c and target qubit q_t, and CM to describe the edges of the device's coupling map—may occur:

- A CNOT gate $CNOT(q_c, q_t)$ shall be applied while q_c and q_t are mapped to physical qubits Q_i and Q_j, respectively, and $(Q_i, Q_j) \notin CM$ as well as $(Q_j, Q_i) \notin CM$.
- A CNOT gate $CNOT(q_c, q_t)$ shall be applied while q_c and q_t are mapped to physical qubits Q_i and Q_j, respectively, and $(Q_i, Q_j) \notin CM$ while $(Q_j, Q_i) \in CM$.

To overcome these problems, one strategy is to insert additional gates into the circuit to be mapped. More precisely, to overcome the first issue, one can insert so-called SWAP operations into the circuit that exchange of the states of two physical qubits and, by this, "move" around the logical ones—changing the mapping dynamically.

Example 8. *Figure 5 shows the effect of a SWAP gate as well as its decomposition into elementary gates supported by the IBM Q devices. Assume that the logical qubits q_0 and q_1 are initially mapped to the physical ones Q_0 and Q_1, respectively (indicated by $\rightarrow$). Then, by applying a SWAP gate, the states of Q_0 and Q_1 are exchanged—eventually yielding a mapping where q_0 and q_1 are mapped to Q_1 and Q_0, respectively.*

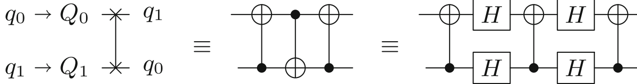
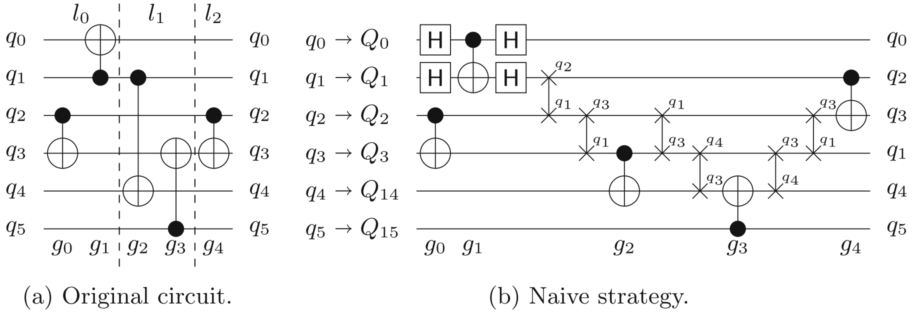


Fig. 5. Decomposition of a SWAP operation.

The second issue may also be solved by inserting SWAP operations. However, it is cheaper (fewer overhead is generated) to insert four Hadamard operations (labelled by H) as they switch the direction of the CNOT gate (i.e., they change the target and the control qubit). This can also be observed in Fig. 5, where H gates switch the direction of the middle CNOT in order to satisfy all coupling-constraints given by the coupling map (assuming that only CNOTs with control qubit Q_1 and target qubit Q_0 are possible).

However, inserting additional gates in order to satisfy the coupling-constraints drastically increases the number of operations—a significant drawback, which affects the fidelity of the quantum circuit since each gate has a certain error rate. Since each SWAP operation is composed of 7 elementary gates (cf. Fig. 5), particularly their number shall be kept as small as possible. Accordingly, this raises the question of how to derive a proper mapping of logical qubits to physical qubits while, at the same time, minimising the number of added SWAP and H operations—an $\mathcal{NP}$-complete problem as recently proven in [47,48].

Example 9. *Consider the quantum circuit composed of 5 CNOT gates shown in Fig. 6a and assume that the logical qubits q_0, q_1, q_2, q_3, q_4, and q_5 are respectively mapped to the physical qubits Q_0, Q_1, Q_2, Q_3, Q_{14}, and Q_{15} of* IBM QX3 *shown in Fig. 4 on Page 16. The first gate can be directly applied, because the coupling-constraints are satisfied. For the second gate, the direction has to be changed because a CNOT with control qubit Q_0 and target Q_1 is valid, but not vice versa. This can be accomplished by inserting Hadamard gates as shown in Fig. 6b. For the third gate, the mapping has to change. To this end, SWAP operations $SWAP(Q_1, Q_2)$ and $SWAP(Q_2, Q_3)$ are inserted to move logical qubit q_1 to become a neighbour of logical qubit q_4 (see Fig. 6b). Afterwards, q_1 and q_4 are mapped to the physical qubits Q_3 and Q_{14}, respectively, which allows applying the desired CNOT gate. Following this procedure for the remaining qubits eventually results in the circuit shown in Fig. 6b. The mapped circuit is composed of 51 elementary operations and has a depth of 36 when using a naive algorithm—a significant overhead that motivates research on improved approaches.*



(a) Original circuit.

(b) Naive strategy.

Fig. 6. Mapping of a quantum circuit to IBM QX3.

5.2 Existing Approaches and Results

There exist only very few algorithms that explicitly tackle the mapping problem for IBM Q devices, and, thus, serve as alternative to IBM's own solution provided within its SDK Qiskit [49].[6] To encourage further development in this area, IBM even launched the *IBM Qiskit Developer Challenge* seeking for the best possible solution [50]. This led to the development of several approaches that explicitly consider design automation techniques to tackle the mapping problem.

The work [51] provides—for the first time—an exact approach (using a formal description of the mapping problem that is passed to a powerful reasoning engine) to solve the mapping problem by inserting the minimum number of additional H and SWAP operations. By this, a lower bound on the overhead is provided (when neglecting pre- and post-mapping optimisations), which is required to satisfy the coupling-constraints given by the quantum hardware—allowing to show that IBM's own solution often exceeds the minimal overhead by more than 100 % (even for small instances). However, the exponential nature of the mapping problem (it has been proven to be $\mathcal{NP}$-complete [47]) makes the exact approach applicable for small instances only.

This limitation—together with the fact that IBM's approach generates mapping that are far above the minimum—motivates the development of heuristic approaches. The heuristic methods presented in [52] are heuristic solution that utilises the A* search method to determine proper mappings. This allows reducing the overhead compared to Qiskit by approximately one fourth on average.[7] This difference in quality is mainly because IBM's solution randomly searches for a mapping that satisfies the coupling-constraints—leading to a rather small exploration of the search space so that only rather poor solutions are usually found. In contrast, the proposed approach aims for an optimised solution by exploring more suitable parts of the search space and additionally exploiting

[6] Note that IBM's solution randomly searches (guided by heuristics) for mappings of the qubits at a certain point of time.

[7] Note that the proposed approach has additionally been integrated into Qiskit to allow a fair comparison by utilising the same post-mapping optimisations.

information of the circuit. More precisely, a look-ahead scheme is employed that considers gates that are applied in the near future and, thus, allows determining mappings which aim for a global optimum (instead of local optima) with respect to the number of SWAP operations.

Even though this heuristic approach allows outperforming Qiskit's mapping algorithm, it has some scalability issues when used for mapping certain random circuits for validating quantum computers [19], which also served as benchmarks in the *IBM Qiskit Developer Challenge* (a challenge for writing the best quantum-circuit compiler to encourage development). These circuits provide a worst-case scenario that heavily affects the efficiency of the proposed heuristic approach. Therefore, a dedicated approach is proposed in [53], which explicitly considers their structure by using dedicated pre- and post-mapping optimisations. The resulting methodology has been declared as winner of the IBM Qiskit Developer Challenge, since it generated mapped/compiled circuits with at least 10 % lower costs than the other submissions while generating them at least 6 times faster, and is currently being integrated into Qiskit by researchers from IBM. Besides that, all mapping approaches developed in context of this thesis are publicly available at http://iic.jku.at/eda/research/ibm_qx_mapping.

6 Conclusion

This chapter has shown the great potential of bringing knowledge gained from the design automation of conventional circuits and systems into the quantum realm. More precisely, quantum-circuit simulation, the design of Boolean components for quantum algorithms, as well as technology mapping have been considered from a design automation perspective—leading to improvements of several orders of magnitude (with respect to runtime or other design objectives) in many cases. For further information on the developed algorithms we refer to the cited papers. In the future, this development shall continue on a larger scale—eventually providing the foundation for design automation methods that accomplish for quantum computing what the design automation community realised for conventional (electronic) circuits.

Acknowledgments. This work has partially been supported by the European Union through the COST Action IC1405 and the LIT Secure and Correct System Lab funded by the State of Upper Austria.

References

1. Nielsen, M.A., Chuang, I.: Quantum computation and quantum information. AAPT **70**, 558 (2002)
2. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Symposium on the Theory of Computing, pp. 212–219 (1996)
3. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. **26**(5), 1484–1509 (1997)

4. Montanaro, A.: Quantum algorithms: an overview. npj Quantum Inf. **2**, 15023 (2016)
5. Preskill, J.: Quantum computing in the NISQ era and beyond. Quantum **2**, 79 (2018)
6. Coles, P.J., et al.: Quantum algorithm implementations for beginners. arXiv preprint arXiv:1804.03719 (2018)
7. Gambetta, J.M., Chow, J.M., Steffen, M.: Building logical qubits in a superconducting quantum computing system. npj Quantum Inf. **3**(1), 2 (2017)
8. Kelly, J.: A preview of Bristlecone, Google's new quantum processor (2018). https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html
9. Hsu, J.: CES 2018: Intel's 49-qubit chip shoots for quantum supremacy. IEEE Spectrum Tech Talk (2018). https://spectrum.ieee.org/tech-talk/computing/hardware/intels-49qubit-chip-aims-for-quantum-supremacy
10. Sete, E.A., Zeng, W.J., Rigetti, C.T.: A functional architecture for scalable quantum computing. In: International Conference on Rebooting Computing (ICRC), pp. 1–6 (2016)
11. IonQ: IonQ: trapped ion quantum computing. https://ionq.co. Accessed 15 June 2019
12. Nay, C.: IBM unveils world's first integrated quantum computing system for commercial use. https://newsroom.ibm.com/2019-01-08-IBM-Unveils-Worlds-First-Integrated-Quantum-Computing-System-for-Commercial-Use. Accessed 15 June 2019
13. Horsman, C., Fowler, A.G., Devitt, S., Van Meter, R.: Surface code quantum computing by lattice surgery. New J. Phys. **14**(12), 123011 (2012)
14. Gottesman, D.: An introduction to quantum error correction and fault-tolerant quantum computation. In: Quantum Information Science and Its Contributions to Mathematics, Proceedings of Symposia in Applied Mathematics, vol. 68, pp. 13–58 (2010)
15. Yamashita, S., Markov, I.L.: Fast equivalence-checking for quantum circuits. In: International Symposium on Nanoscale Architectures. pp. 23–28. IEEE Press (2010)
16. Niemann, P., Wille, R., Drechsler, R.: Equivalence checking in multi-level quantum systems. In: International Conference of Reversible Computation, pp. 201–215 (2014)
17. Burgholzer, L., Wille, R.: Improved DD-based equivalence checking of quantum circuits. In: Asia and South Pacific Design Automation Conference (ASP-DAC) (2020)
18. Boixo, S., et al.: Characterizing quantum supremacy in near-term devices. Nat. Phys. **14**(6), 595 (2018)
19. Cross, A.W., Bishop, L.S., Sheldon, S., Nation, P.D., Gambetta, J.M.: Validating quantum computers using randomized model circuits. arXiv preprint arXiv:1811.12926 (2018)
20. Smelyanskiy, M., Sawaya, N.P.D., Aspuru-Guzik, A.: qHiPSTER: the quantum high performance software testing environment. arXiv preprint arXiv:1601.07195 (2016)
21. Zulehner, A., Wille, R.: Advanced simulation of quantum computations. IEEE Trans. CAD Integr. Circuits Syst. **38**, 848–859 (2019)
22. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Quantum Circuit Simulation. Springer, Dordrecht (2009). https://doi.org/10.1007/978-90-481-3065-8

23. Niemann, P., Zulehner, A., Wille, R., Drechsler, R.: Efficient construction of QMDDs for irreversible, reversible, and quantum functions. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 214–231. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_17
24. Zulehner, A., Wille, R.: Matrix-vector vs. matrix-matrix multiplication: potential in DD-based simulation of quantum computations. In: Design, Automation and Test in Europe, European Design and Automation Association (2019)
25. Zulehner, A., Hillmich, S., Wille, R.: How to efficiently handle complex values? Implementing decision diagrams for quantum computation. In: International Conference on CAD (2019)
26. Niemann, P., Datta, R., Wille, R.: Logic synthesis for quantum state generation. In: International Symposium on Multi-Valued Logic, pp. 247–252. IEEE (2016)
27. Niemann, P., Wille, R., Drechsler, R.: Improved synthesis of Clifford+T quantum functionality. In: Design, Automation and Test in Europe, pp. 597–600 (2018)
28. Zulehner, A., Niemann, P., Drechsler, R., Wille, R.: Accuracy and compactness in decision diagrams for quantum computation. In: Design, Automation and Test in Europe (2019)
29. Hillmich, S., Zulehner, A., Wille, R.: Concurrency in DD-based quantum circuit simulation. In: Asia and South Pacific Design Automation Conference (ASP-DAC) (2020)
30. Zulehner, A., Hillmich, S., Markov, I., Wille, R.: Approximation of Quantum States Using Decision Diagrams. Asia and South Pacific Design Automation Conference (ASP-DAC) (2020)
31. Ekert, A., Jozsa, R.: Quantum computation and Shor's factoring algorithm. Rev. Mod. Phys. **68**(3), 733 (1996)
32. Soeken, M., Roetteler, M., Wiebe, N., De Micheli, G.: LUT-based hierarchical reversible logic synthesis. IEEE Trans. CAD Integr. Circuits Syst. **38**, 848–859 (2018)
33. Zulehner, A., Wille, R.: Make it reversible: efficient embedding of non-reversible functions. In: Design, Automation and Test in Europe, European Design and Automation Association, pp. 458–463 (2017)
34. Soeken, M., Wille, R., Hilken, C., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: Asia and South Pacific Design Automation Conference, pp. 85–92 (2012)
35. Zulehner, A., Wille, R.: Improving synthesis of reversible circuits: exploiting redundancies in paths and nodes of QMDDs. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 232–247. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_18
36. Zulehner, A., Wille, R.: One-pass design of reversible circuits: combining embedding and synthesis for reversible logic. IEEE Trans. CAD Integr. Circuits Syst. **37**(5), 996–1008 (2018)
37. Zulehner, A., Wille, R.: Skipping embedding in the design of reversible circuits. In: International Symposium on Multi-Valued Logic, pp. 173–178. IEEE (2017)
38. Zulehner, A., Wille, R.: Exploiting coding techniques for logic synthesis of reversible circuits. In: Asia and South Pacific Design Automation Conference, pp. 670–675. IEEE Press (2018)
39. Zulehner, A., Niemann, P., Drechsler, R., Wille, R.: One additional qubit is enough: encoded embeddings for Boolean components in quantum circuits. In: International Symposium on Multi-Valued Logic (2019)

40. Amy, M., Maslov, D., Mosca, M., Roetteler, M.: A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **32**(6), 818–830 (2013)
41. Miller, D.M., Wille, R., Sasanian, Z.: Elementary quantum gate realizations for multiple-control Toffoli gates. In: International Symposium on Multi-Valued Logic, pp. 288–293. IEEE (2011)
42. Wille, R., Keszocze, O., Walter, M., Rohrs, P., Chattopadhyay, A., Drechsler, R.: Look-ahead schemes for nearest neighbor optimization of 1D and 2D quantum circuits. In: Asia and South Pacific Design Automation Conference, pp. 292–297 (2016)
43. Shafaei, A., Saeedi, M., Pedram, M.: Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures. In: Design Automation Conference, pp. 41–46 (2013)
44. Wille, R., Quetschlich, N., Inoue, Y., Yasuda, N., Minato, S.I.: Using πDDs for nearest neighbor optimization of quantum circuits. In: International Conference of Reversible Computation, pp. 181–196 (2016)
45. Zulehner, A., Gasser, S., Wille, R.: Exact global reordering for nearest neighbor quantum circuits using A^*. In: Phillips, I., Rahaman, H. (eds.) RC 2017. LNCS, vol. 10301, pp. 185–201. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59936-6_15
46. IBM Q team: IBM Q 16 Rueschlikon backend specification v1.0.0. https://ibm.biz/qiskit-rueschlikon. Accessed 15 June 2019
47. Botea, A., Kishimoto, A., Marinescu, R.: On the complexity of quantum circuit compilation. In: Symposium on Combinatorial Search (2018)
48. Siraichi, M., Dos Santos, V.F., Collange, S., Pereira, F.M.Q.: Qubit allocation. In: International Symposium on Code Generation and Optimization (CGO), pp. 1–12 (2018)
49. Cross, A.: The IBM Q experience and QISKit open-source quantum computing software. Bull. Am. Phys. Soc. **63**(1) (2018)
50. IBM Q team: QISKit Developer Challenge. https://qx-awards.mybluemix.net/#qiskitDeveloperChallengeAward. Accessed 15 June 2019
51. Wille, R., Burgholzer, L., Zulehner, A.: Mapping quantum circuits to IBM QX architectures using the minimal number of SWAP and H operations. In: Design Automation Conference (2019)
52. Zulehner, A., Paler, A., Wille, R.: An efficient methodology for mapping quantum circuits to the IBM QX architectures. IEEE Trans. CAD Integr. Circuits Syst. **38**, 1226–1236 (2018)
53. Zulehner, A., Wille, R.: Compiling SU(4) quantum circuits to IBM QX architectures. In: Asia and South Pacific Design Automation Conference, pp. 185–190. ACM (2019)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

Research on Reversible Functions Having Component Functions with Specified Properties: An Overview

Paweł Kerntopf[1(✉)], Claudio Moraga[2], Krzysztof Podlaski[3], and Radomir Stanković[4]

[1] Institute of Computer Science, Warsaw University of Technology, Warsaw, Poland
pawel.kerntopf@gazeta.pl
[2] Faculty of Computer Science, Technical University of Dortmund, Dortmund, Germany
claudio.moraga@tu-dortmund.de
[3] Faculty of Physics and Applied Informatics, University of Łódź, Łódź, Poland
podlaski@uni.lodz.pl
[4] Department of Computer Science, Faculty of Electronic Engineering, University of Niš, Niš, Serbia
radomir.stankovic@gmail.com

Abstract. In the traditional logic synthesis, different classifications of non-reversible Boolean functions have found many applications. Recently, some attempts to deal with classifications of reversible functions have been published. In this paper, an overview of our results towards constructing a new classification of reversible functions is presented. These results were obtained due to our discussions during two Short Term Scientific Missions (STSMs) as well as during our further research in the framework of COST Action IC1405 "Reversible Computation - Extending Horizons of Computing" and were published in five papers.

Keywords: Reversible functions · Component functions · Classification

1 Introduction

Recent advances in nanotechnology, low-power design, and quantum computing have renewed interest in reversible logic synthesis since they allow for reducing the power dissipation in related circuits and the potential speed-up in quantum computations. More details can be found in [4,25] and references therein.

A reversible function is defined as a bijective mapping $f : A^n \rightarrow A^n$, where A is any finite set of elements which can be conveniently identified with non-negative integers $\{0, 1, \ldots, p-1\}$. In particular, for $p = 2$ and $p = 3$, we speak about binary or Boolean and ternary reversible functions, respectively. Therefore, an n-variable reversible function is actually a permutation on A^n, and can

© The Author(s) 2020
I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 83–107, 2020.
https://doi.org/10.1007/978-3-030-47361-7_4

be viewed as a vector of n functions called the *component functions* (CFs), i.e., $F = (f_1, f_2, \ldots, f_n)$. In [27], the term *components* is applied in the similar meaning, meanwhile in the literature on cryptography the term *coordinate functions* is used, see e.g., [3,30]. However, in [30] the term *component function* means a linear combination of *coordinate functions*.

Correspondingly, a reversible circuit is a circuit that realises a reversible function, i.e., performs a bijective mapping of n input signals onto n output signals in a manner specified by the function to be realised.

Recently in [13,14], we discussed the question if it is possible to extend a Boolean function $f : \{0,1\}^n \to \{0,1\}$ into a reversible function $F : \{0,1\}^n \to \{0,1\}^n$, *under the condition that all its component functions have a homogeneous property*. The term homogeneous property means that all component functions express the same particular property Boolean functions might have, e.g. all the component functions belong to the same equivalence class in a particular classification of Boolean functions. The motivation was that if such an embedding of a Boolean function into a reversible function is possible, then new classes of reversible functions can be defined. In [15,17] the same question is explored for ternary functions $F : \{0,1,2\}^n \to \{0,1,2\}^n$ and we have shown that there are significant differences in the theory of binary and ternary reversible functions in the case of linear component functions.

As homogeneous properties, we have chosen typical ones considered in classical logic synthesis: symmetry, affinity, linearity, nonlinearity, self-duality, self-complementarity, monotonicity, unateness (see, e.g. [31]). In papers [13–15,17] the exemplary functions used in proofs of the results were obtained in a constructive manner. In [16] the results on properties of component functions of Boolean reversible functions obtained by the extrapolation approach were demonstrated. An overview of the most relevant of these results in the binary case is presented here. Because of lack of space we have omitted our results on reversible multiple-valued functions. The reader can find them in [15,17].

The presentation is organised in the following way. For the sake of completeness, necessary definitions and basic results from the theory of standard Boolean as well as from reversible Boolean functions are provided in Sect. 2. In Sect. 3, a brief overview of related and background work is presented. Section 4 demonstrates our theoretical results on properties of component functions of reversible functions. Section 5 describes the results of our research on the existence of Boolean reversible functions with all component functions belonging to different equivalence classes while considering well-known and newly constructed reversible functions defined for any number of variables. Section 6 presents our numerical calculations of all equivalence classes of balanced Boolean functions up to $n = 4$ and all reversible functions up to $n = 3$. Finally, Sect. 7 describes our results on the existence of Boolean reversible functions with specified properties of all component functions obtained by extrapolating some properties of reversible functions. The presented research is summarised in Sect. 8.

2 Preliminaries

In this section, the basic definitions and known results are provided for the convenience of the reader. Let us first briefly survey fundamental notions related to standard Boolean functions and reversible Boolean functions.

First we present notation and terminology for fundamental notions. The symbols $+, -, \cdot$ denote ordinary addition, subtraction, and multiplication, respectively. For arbitrary elements x and y in the set $\{0, 1\}$ basic operations in this set (one unary and three binary operations) are defined in the usual way:

Negation $x' = 1 - x$, i.e. if the argument x is 0, then the result is 1, otherwise it is 0;
Product $xy = x \cdot y$, i.e. its value is 1 if and only if both arguments are 1;
Sum $x \vee y = x + y - x \cdot y$, i.e. its value is 0 if and only if both arguments are 0;
EXOR $x \oplus y = x + y(mod2) = x + y - 2 \cdot x \cdot y$, i.e. its value is 1 if and only if exactly one argument is 1.

In classical logic synthesis, the basic representation of a Boolean function is the Sum-of-Products expression (SOP). In the field of reversible circuit synthesis two other representations are commonly used. Any Boolean function $f : \{0, 1\}^n \rightarrow \{0, 1\}$ can be described using an EXOR-sum of products (ESOP) expression. In ESOPs each variable may appear in both uncomplemented and complemented forms. The Positive Polarity Reed-Muller (PPRM) expression is an ESOP expression which uses only uncomplemented variables. It is a canonical expression and for small functions can be easily generated from a truth table or other representations of the Boolean function.

A Boolean function $f(x_1, x_2, \ldots, x_n)$ depends essentially on its variable x_i if and only if $f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \neq f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$.

Definition 1. *A Boolean function depending essentially on all its variables is called* non-degenerate*, otherwise it is called* degenerate.

Example 1. *There are 16 functions of two variables x and y:* 0, 1, x, x', y, y', xy, $x'y$, xy', $x'y'$, $x \vee y$, $x' \vee y$, $x \vee y'$, $x' \vee y'$, $x \oplus y = x' \oplus y'$, $x' \oplus y = x \oplus y'$. *The first six of them are degenerate: the first two depend essentially on none of the variables, the next four depend essentially on only one of the variables.* ∎

Let us define an order relation in the set $\{0, 1\}$ in the usual way: $0 < 1$ and a partial order relation in the set $\{0, 1\}^n$: for any two vectors $\mathbf{a} = (a_1, a_2, \ldots, a_n)$, $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ in $\{0, 1\}^n$ $\mathbf{a} \leq \mathbf{b}$ if and only if $a_i \leq b_i$ for $1 \leq i \leq n$.

Definition 2. *A Boolean function f is* monotone increasing *if and only if* $\mathbf{a} \leq \mathbf{b}$ *implies* $f(\mathbf{a}) \leq f(\mathbf{b})$ *which will simply be called a* monotone *function. By changing the inequalities into inverse ones we obtain a definition of* monotone decreasing *function.*

Example 2. *Both the constant functions 0 and 1 are monotone increasing and monotone decreasing. There are six monotone increasing functions of two variables x and y:* 0, 1, x, y, xy, $x \vee y$. *Similarly, there are six monotone decreasing functions of two variables x and y:* 0, 1, x', y', $x'y'$, $x' \vee y'$. ∎

Definition 3. *A Boolean function $f(x_1, x_2, \ldots, x_n)$ is called* unate *(or mixed monotone) if and only if it is a constant or there exists its SOP representation using either uncomplemented or complemented literals for each variable.*

Example 3. *There are 14 unate functions of two variables x and y: only functions $x \oplus y$ and $x' \oplus y$ are not unate.* ∎

Definition 4. *A Boolean function $f(x_1, x_2, \ldots, x_n)$ is called* threshold *(or* linearly separable*) if and only if there exist real numbers $a_1, a_2, \ldots, a_n$, and b such that $f = 1$ if the sum of all $a_i x_i$, $1 \leq i \leq n$, is greater than or equal to b, and $f = 0$ otherwise.*

Example 4. *All unate functions of up to three variables are threshold functions. Thus, for two variables there are 14 threshold functions (i.e., all except $x \oplus y$ and $x' \oplus y$), in particular,*

$$\begin{aligned}
&\textit{when} \quad a_1 = a_2 = 1 \quad \textit{and} \quad b = 1.5 \quad \textit{then} \quad f = xy, \\
&\textit{when} \quad a_1 = a_2 = 1 \quad \textit{and} \quad b = 0.5 \quad \textit{then} \quad f = x \vee y, \\
&\textit{when} \quad a_1 = a_2 = -1 \quad \textit{and} \quad b = -0.5 \quad \textit{then} \quad f = x'y', \\
&\textit{when} \quad a_1 = a_2 = -1 \quad \textit{and} \quad b = -1.5 \quad \textit{then} \quad f = x' \vee y'.
\end{aligned}$$

The 4-variable function $f(x_1, x_2, x_3, x_4) = x_1x_2 \vee x_3x_4$ is an example of a monotone increasing function which is not a threshold function. ∎

Definition 5. *A Boolean function f on an odd number of arguments is called* majority function *if and only if $f = 1$ when more than half of the arguments are* 1.

Example 5. *The 3-variable majority function $f(x, y, z) = xy \oplus xz \oplus yz$ is a threshold function, where $a_1 = a_2 = a_3 = 1$ and $b = 2$.* ∎

It is well known that the following result holds.

Lemma 1.

(1) Every threshold function is a unate function.
(2) Every majority function is a threshold function.

Definition 6. *A Boolean function f is* linear *with respect to a variable x_i if it can be expressed in the form $f = x_i \oplus g$, where $\oplus$ denotes XOR operation and g is a function independent of x_i (then the variable x_i is called* linear *in f). A function has property LV if it contains at least one linear variable. A function f is called* affine *if and only if each of variable x_i is either linear in f, or f does not depend on x_i, i.e. $f(x_1, x_2, \ldots, x_n) = a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n$, where $a_0, a_1, a_2, \ldots, a_n \in \{0, 1\}$. If $a_0 = 0$ then it is called* linear. *Any affine function which is not linear can be obtained by negating an appropriate linear function. A Boolean function which is not affine is called* nonlinear.

Example 6. *$f_1(x, y, z) = x \oplus y \oplus yz$ is linear with respect to x as then $g = y \oplus yz$ is independent of x, but f_1 is not linear with respect to y as then $g = x \oplus yz$ is dependent of y. Similarly, $f_2(x, y) = x \oplus y \oplus xy$ is neither linear with respect to x, nor to y.* ∎

Definition 7. *A Boolean function is (*totally*)* symmetric *if any permutation of all its variables does not change the function.*

There are 2^{n+1} symmetric Boolean functions.

Definition 8. *If any permutation of a proper subset S of the variables of cardinality at least 2 does not change the Boolean function f, then f is called a* partially symmetric function with respect to S *and S is called* a partial symmetry of variables of f. *The collection of maximal partial symmetry subsets of variables of f is called* a partial symmetry profile *and is denoted by S_f. The partial symmetry profile of a totally symmetric Boolean function $f(x_1, x_2, \ldots, x_3)$ is equal to $\{\{x_1, x_2, \ldots, x_n\}\}$. Let partial symmetry profiles of Boolean functions $f_1, f_2, \ldots, f_n$ be denoted by $S_1, S_2, \ldots, S_n$, respectively. The intersection of such profiles is the collection of subsets of variables obtained by taking all possibilities of performing intersection operation on an element in S_1, an element in S_2, ..., and an element in S_n. If the intersection operation on $S_1, S_2, \ldots, S_n$ does not contain an element with at least two variables, then there does not exist a partial symmetry subset of all functions $f_1, f_2, \ldots, f_n$.*

Example 7. *Let $f(u, v, w, x, y, z) = u \oplus vw \oplus xyz$ and $g(u, v, w, x, y, z) = uv \oplus w \oplus xyz$. Then the partial symmetry profile of f is $S_f\{\{v, w\}, \{x, y, z\}\}$, the partial symmetry profile of g is $S_g = \{\{u, v\}, \{x, y, z\}\}$ and the intersection of S_f and S_g is equal to $\{\{v\}, \phi, \{x, y, z\}\}$, i.e. both functions f and g are partially symmetric with respect to $\{\{x, y, z\}\}$ as well as this subset is the only one partial symmetry subset of both functions f and g.* ∎

Definition 9. *A Boolean function $f : \{0, 1\}^n \to \{0, 1\}$ is called balanced if it takes value 1 the same number of times as value 0.*

Example 8. *There are 70 balanced Boolean functions on 3 arguments, including degenerate ones. Only four of them are totally symmetric, namely parity and majority functions and their negations:*

$$\begin{array}{lll} \textit{parity} & x \oplus y \oplus z & 1 \oplus x \oplus y \oplus z, \\ \textit{majority} & xy \oplus xz \oplus yz & 1 \oplus xy \oplus xz \oplus yz. \end{array}$$

Eight of the balanced functions, including degenerate ones, are partially symmetric with respect to each 2-element subset of variables, for instance, functions

$$\begin{array}{cc} x \oplus y, & 1 \oplus x \oplus y, \\ xy \oplus z, & 1 \oplus xy \oplus z, \\ x \oplus y \oplus xy \oplus z, & 1 \oplus x \oplus y \oplus xy \oplus z, \\ x \oplus y \oplus xy \oplus xz \oplus yz, & 1 \oplus x \oplus y \oplus xy \oplus xz \oplus yz. \end{array}$$

are partially symmetric with respect to $\{x, y\}$. ∎

Definition 10. *Two Boolean functions are:*

(1) P-equivalent *if they can be converted to each other by the permutation of variables,*

(2) NP-equivalent *if they can be converted to each other by the negation and/or permutation of variables,*
(3) NPN-equivalent *if they can be converted to each other by negation of variables, permutation of variables and negation of the function.*

Definition 11. *A Boolean function f is* self-complementary *(SC) if f and f' are NP-equivalent.*

Definition 12. *A Boolean function f is* self-dual *(SD) if*

$$f(x_1, x_2, \dots, x_n) = f'(x_1', x_2', \dots, x_n').$$

The following results are well-known:

Lemma 2. *(1) All self-complementary functions are balanced,*
(2) All self-dual functions are self-complementary,
(3) All functions having property LV are self-complementary,
(4) If a Boolean function f is linear with respect to a variable x_i then

$$f(x_i = 1) = f'(x_i = 0).$$

In the case of Boolean functions, depending on the operations allowed in a particular classification, the P-equivalent, NP-equivalent, and NPN-equivalent functions are distinguished. In some applications, equivalence classes defined with respect to a restricted set of operations are of a particular interest, as for example, in [5,6]. Here, we are particularly interested in P-equivalent functions when studying the properties of component functions.

Definition 13. *A mapping $F : \{0,1\}^n \rightarrow \{0,1\}^n$ is called an $n{*}n$* reversible function *if it is bijective. Functions which are not reversible are called irreversible. An $n{*}n$ reversible function F can be considered as a vector of standard Boolean functions called* component functions $f_i : \{0,1\}^n \rightarrow \{0,1\}, 1 \leq i \leq n$, *which are defined at every $x \in \{0,1\}^n$ by $F(x) = (f_1(x), \dots, f_n(x))$.*

In the truth table of a reversible $n{*}n$ Boolean function there are n input columns and n output columns. The output rows of such a truth table form a permutation of the input rows. From the bijectivity of reversible functions it follows that all component functions have to be balanced Boolean functions.

By an analogy with the definition of NPN-equivalence classes for standard Boolean functions, the following definition of equivalence classes for Boolean reversible functions can be given.

Definition 14. *Two reversible Boolean functions are NPNP-equivalent if they can be transformed to each other by the following operations (including the combinations that do not use all of these operations):*

(1) Negation of variables,
(2) Permutation of variables,
(3) Negation of component functions, and
(4) Permutation of component functions.

Each reversible function can be treated as a permutation. This is why we also recall basic notions connected with permutations. Let A be any finite set. A permutation on a set A is a bijective mapping from A to itself. Every permutation can be considered as a collection of disjoint cycles. Here such a collection will be called a cycle structure. We will write a cycle in the form $< a_1, a_2, \ldots, a_k >$, meaning that a_1 is mapped onto $a_2, \ldots, a_k$ is mapped onto a_1. It could be written in different ways, e.g. $< a_2, a_3, \ldots, a_k, a_1 >$. The number of elements in a cycle is called the length of the cycle. A cycle with the length k is called a *k-cycle*. A 2-cycle is also called a *transposition*.

3 Previous Work

The motivation for our studies of reversible functions toward constructing their classifications is borrowed from the classical logic synthesis by referring to an analogy with related problems. For example, in classical logic synthesis, the equivalence of two functions under permutation of the variables is an important problem due to applications in the synthesis of multiplexer-based field-programmable gate arrays [5,6]. The problem is called Boolean matching, and two functions match if they have the same P-representative. The extension to NP-representatives is done in [7,8] in solving the Boolean matching problem in cell-library binding.

Classification of Boolean functions is a classical problem in logic synthesis due to its various applications, with fast prototyping and unification of testing procedures being just two of them [28]. However, a considerably smaller amount of work has been done in the classification of reversible functions. In [18,19] it is presented an approach to enumerate equivalence classes of reversible functions with the equivalence classes defined as follows. Denote by G and H the groups of permutations acting on the inputs and outputs of Boolean reversible functions, respectively. Two functions $f_1(x)$ and $f_2(x)$ are equivalent if for each n-tuple x, there is a $g \in G$ and an $h \in H$ such that $f_1(x) = h(f_2(g(x)))$. It is also provided a list of all NPNP-equivalence classes of 3-variable reversible functions as well as a classification based on properties of the inverses of the representative functions for the equivalence classes considered. The lists consist of triples of balanced Boolean functions specified by ESOPs. Unfortunately, using "prime" for negation led to a number of typographical errors which has been discovered by us recently [13] (see Sect. 6).

A technical report from 1962 by C. S. Lorens [18] and an article by the same author [19] can be viewed as a starting point of subsequent work on an enumeration of equivalence classes of reversible functions by several authors [10,20–23,29]. With the exception of [22], these publications consider the classification of binary reversible functions. These publications were discussed mainly by researchers in combinatorial mathematics and cryptography but hardly used and correspondingly rarely if at all referred within the reversible functions community, the main reason probably being that the term *invertible* instead of *reversible*

functions has been used. A classification scheme for reversible functions was the subject of a profound study in [24], however, without a concrete solution proposed.

Recently, certain aspects of the classification problem have been addressed. In [26], the list of all NPNP-equivalence classes for three variable reversible functions from [19] is presented in the context of a study of the complexity of reversible circuits with the representative functions for equivalence classes given in the form of permutations (i.e. without considering individual component functions). The minimal number of nonlinear gates needed in the implementation of reversible functions is used as a classification criterion in [9]. The structure of closed classes of reversible functions is described in [1]. Enumeration of equivalence classes under the action of permutation of the inputs and outputs on the domain and the range is presented in [2].

For the first time in the literature, we solved in [13,14] several problems of the existence of binary reversible functions with all component functions having the same known property (e.g., symmetry, affinity, linearity, nonlinearity, self-duality, self-complementarity, monotonicity, unateness). Solutions of such problems for ternary reversible functions are presented by us in [15]. In [17] we presented results on the existence of ternary/multiple-valued reversible functions with all component functions belonging to different P-equivalence classes. In [16] it is shown how we discovered solutions of some problems by extrapolating properties of previously found reversible functions of 3 and 4 variables.

4 Theoretical Results

This section presents basic theoretical results on properties of component functions of reversible functions. We begin with the following general result:

Theorem 1. *If $f(x_1, x_2, \ldots, x_n) = (f_1, f_2, \ldots, f_n)$ is an $n{*}n$ reversible (irreversible) Boolean function, then the function obtained from f by any of the following transformations*

- *negation of variables,*
- *permutation of variables,*
- *negation of a component function,*
- *permutation of component functions,*

is also reversible (irreversible).

Proof. It is sufficient to notice that any of the above transformations corresponds to a permutation of rows in the truth table, i.e. preserves the property of bijectivity. □

The following result follows directly from Theorem 1.

Corollary 1. *$n{*}n$ functions belonging to an NPNP-equivalence class either are all reversible or none of them is reversible.*

There are constraints on using totally and partially symmetric functions as component functions of an $n*n$ reversible function $f(x_1, x_2, \ldots, x_n) = (f_1, f_2, \ldots, f_n)$. Let partial symmetry profiles of the component functions $f_1, f_2, \ldots, f_n$ be denoted by $S_1, S_2, \ldots, S_n$, respectively (see Definition 8).

Theorem 2. *A necessary condition for an $n*n$ function $f(x_1, x_2, \ldots, x_n) = (f_1, f_2, \ldots, f_n)$ to be reversible is as follows: intersection of all profiles $S_1, S_2, \ldots, S_n$, has to be equal to the collection of results each of which has no more than one element.*

Proof. Let us assume that two variables x_i and x_j belong to one subset being an element of the intersection of the profiles $S_1, S_2, \ldots, S_n$, i.e. appear in one subset in all these profiles. It is equivalent to the equation:

$$f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_n).$$

However, because any reversible function f is a bijective mapping then $f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_{j-1}, 1, x_{j+1}, \ldots, x_n)$ differs from $f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_{j-1}, 0, x_{j+1}, \ldots, x_n)$. □

Thus, the following theorem holds.

Theorem 3. *$n*n$ reversible Boolean functions, $n > 1$, with all totally symmetric CFs being non-degenerate do not exist.*

On the other hand, component functions of a reversible function can be totally or partially symmetric if at least two of them are partially symmetric.

Example 9. *It is easy to show that the following function is reversible*

$$f_1 = x_1 \oplus x_2 \oplus x_3, \qquad f_2 = x_1 \oplus x_2, \qquad f_3 = x_1 \oplus x_3,$$

where f_1 is a totally symmetric function and both f_2 and f_3 are partially symmetric functions The simple generalization of the above reversible function to the case of any n can be defined as follows:

$$f_1 = \bigoplus_{i=1}^{n} x_i, \qquad f_k = \bigoplus_{i \neq k} x_i, k \in \{2, \ldots, n\},$$

where the symbol $\bigoplus$ denotes summing modulo 2. ▮

In some papers, algorithms for synthesis of reversible circuits for (totally) symmetric functions are considered. However, symmetric functions in these papers are first embedded in reversible specifications with additional inputs and/or outputs.

Now let us consider linear and affine CFs. For any n there is only one non-degenerate linear Boolean function:

$$x_1 \oplus x_2 \oplus \cdots \oplus x_n.$$

Hence, by Theorem 1 the following result is true:

Theorem 4. *For $n > 1$ $n{*}n$ reversible Boolean functions with all linear or affine CFs being non-degenerate do not exist.*

However, reversible Boolean functions having as CFs one of non-degenerate linear (affine) functions and the other functions depending essentially on $k < n$ variables do exist as is shown in Example 9.

Let us consider the following property of monotone Boolean functions.

Lemma 3. *Every monotone Boolean function which is balanced, except projection functions P-equivalent to the identity, cannot be equal to* 1 *for an assignment with weight* 1 *(i.e. with only one non-zero entry).*

Proof. Assume that the lemma is not true. Then there exists a balanced monotone Boolean function f, not being a projection function, and an assignment $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ with weight 1 for which $f(\mathbf{a})$ is not equal to zero. Without loss of generality let $\mathbf{a} = (1, 0, \ldots, 0, 0)$, i.e. $x_1 = 1$, $x_i = 0$ for $2 \leq i \leq n$. Because f is monotone so $f(\mathbf{b}) = 1$ for all assignments $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ with $b_1 = 1$. The number of such assignments is equal to 2^{n-1}. As the number of all binary assignments is 2^n hence the number of assignments $\mathbf{c}$ not compatible with assignments $\mathbf{b}$, i.e. having $c_1 = 0$, is equal to $2^n - 2^{n-1} = 2^{n-1}$. A balanced Boolean function takes values 0 and 1 the same number of times so $f = 0$ for all those assignments with $c_1 = 0$. Thus,

$$
\begin{aligned}
f(\mathbf{a}) &= 0 \qquad \text{for all binary assignments} \qquad \mathbf{a} = (0, a_2, \ldots, a_{n-1}, a_n),\\
f(\mathbf{a}) &= 1 \qquad \text{for all binary assignments} \qquad \mathbf{a} = (1, a_2, \ldots, a_{n-1}, a_n),
\end{aligned}
$$

i.e., f is a projection function what is in contradiction with the initial assumption. □

Theorem 5. *An $n{*}n$ reversible Boolean function, $n \geq 3$, with all component functions being non-degenerate monotone does not exist.*

Proof. Lemma 3 states that for any monotone balanced Boolean function F and any input assignment $\mathbf{a}$ with weight 1

$$F(\mathbf{a}) = 0.$$

Thus, for any $n{*}n$ reversible Boolean function G, $n \geq 3$, with all CFs being monotone and any input assignment $\mathbf{a}$ with weight 1

$$G(0, 0, \ldots, 0, 0, 1) = G(0, 0, \ldots, 0, 1, 0) = (0, 0, \ldots, 0, 0, 0),$$

what contradicts the reversibility constraint as G takes value $(0, 0, \ldots, 0, 0, 0)$ more than once. Thus, any $n{*}n$ Boolean function, $n \geq 3$, with all component functions being monotone, is not reversible.

By Definition 3, Definition 4, Lemma 1, Theorem 1 and Theorem 5 the following result holds. □

Corollary 2. *An $n{*}n$ reversible Boolean function, $n \geq 3$, with all component functions being non-degenerate and threshold, does not exist.*

5 Results Based on Newly Constructed Functions

Let us first introduce simple notions related to Positive Polarity Reed-Muller expressions for Boolean functions. The number of literals in a term will be called its rank. Denote by $T_{i,j}$ the exclusive-or sum of all terms having a rank not smaller than i and not greater than j ($T_{i,i}$ will denote all terms with rank i).

In [13] we introduced the following $n{*}n$ reversible function, for arbitrary n, which will be called Negation with Preservation of Constants (in short NPC$n{*}n$):

Definition 15. *The reversible function NPC$n{*}n(x_1, x_2, \ldots, x_n)$, $n \geq 3$, is defined in such a manner that its component functions NPCn are defined as follows:*

$$\begin{aligned}
f_1 &= x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_{n-1} \oplus T_{2,n-1},\\
f_2 &= x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_n \oplus T_{2,n-1},\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
f_{n-1} &= x_1 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_{n-1} \oplus x_n \oplus T_{2,n-1},\\
f_n &= x_2 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_{n-1} \oplus x_n \oplus T_{2,n-1},
\end{aligned}$$

i.e., in each of the above equations exactly one variable is missing, namely in the ith equation variable x_{n-i+1} is missing.

The formulas in Definition 15 can be transformed taking into account that

1. $x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_{n-1} \oplus x_n \oplus T_{2,n} = T_{1,n} = \\ = x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_{n-1} \vee x_n,$

(this transformation can be easily proved by induction starting from the well-known formula for $n = 2 : x_1 \vee x_2 = x_1 \oplus x_2 \oplus x_1x_2$),

2. $f_i = f_i \oplus x_{n-i+1} \oplus x_{n-i+1} \oplus x_1x_2 \ldots x_n \oplus x_1x_2 \ldots x_n = T_{1,n} \oplus x_{n-i+1} \oplus x_1x_2 \ldots x_n \\ = (x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_{n-1} \vee x_n) \oplus x_{n-i+1} \oplus x_1x_2 \ldots x_n,$

where we applied the following three obvious formulas:

$$f_i = f_i \oplus 0, \qquad x_{n-i+1} \oplus x_{n-i+1} = 0, \qquad x_1x_2 \ldots x_n \oplus x_1x_2 \ldots x_n = 0,$$

and the transformation used in case 1:

$$\begin{aligned}
f_i \oplus x_{n-i+1} \oplus x_1x_2 \ldots x_n &= x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_{n-1} \oplus x_n \oplus T_{2,n-1} \oplus x_1, x_2 \ldots x_n\\
&= x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_{n-2} \oplus x_{x-1} \oplus x_n \oplus T_{2,n} = T_{1,n}.
\end{aligned}$$

Using the above formulas we will show by example how the values of the function f_i can be calculated. Without loss of generality we will show this for f_1:
Step 1. Calculate $f_1^{(1)} = x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_{n-1} \vee x_n$ (see Table 1).
Step 2. Calculate $f_1^{(2)} = (x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_{n-1} \vee x_n) \oplus x_1$ (by negating the lower half of the truth table obtained in Step 1).
Step 3. Calculate $f_1 = (x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_{n-1} \vee x_n) \oplus x_1 \oplus x_1 x_2 \ldots x_n$ (by negating the output value in the last row of the truth table obtained in Step 2). These steps are performed for $n = 3$ in Table 1.

Thus, f_1 has the well-known property of preserving constants:

$$f_1(0,0,0) = 0 \quad \text{and} \quad f_1(1,1,1) = 1,$$

as well as is negating the input x_1 for all other vectors of input values. Similarly (see Table 2), the reversible function NPC3∗3 is preserving constants:

$$\text{NPC3}{*}3(0,0,0) = (0,0,0) \quad \text{and} \quad \text{NPC3}{*}3(1,1,1) = (1,1,1),$$

as well as negating all the other input vectors. This is why we gave this reversible function the name Negation with Preservation of Constants.

Table 1. Establishing values of the 3-variable function f_1 in three steps

$x_1x_2x_3$	$f_1^{(1)}$	$f_1^{(2)}$	f_1
0 0 0	0	0	0
0 0 1	1	1	1
0 1 0	1	1	1
0 1 1	1	1	1
1 0 0	1	0	0
1 0 1	1	0	0
1 1 0	1	0	0
1 1 1	1	0	1

By analogy with the above example it is easy to show that the following two results hold for any n:

Lemma 4. *Each function NPCn∗n is reversible.*

Lemma 5. *Each component function of NPCn∗n can be obtained from the Boolean function NPCn as a result of a permutation of its variables.*

Table 2. Truth table for the function NPC3∗3

$x_1x_2x_3$	$f_1f_2f_3$
0 0 0	0 0 0
0 0 1	1 1 0
0 1 0	1 0 1
0 1 1	1 0 0
1 0 0	0 1 1
1 0 1	0 1 0
1 1 0	0 0 1
1 1 1	1 1 1

Theorem 6. *All component functions of NPCn∗n, $n \geq 3$, are (1) nonlinear, (2) self-dual, (3) self-complementary, (4) P-equivalent, and (5) unate.*

Proof. (1) The function NPCn is nonlinear because its PPRM contains terms of rank 2 for any $n > 2$.

(2) Without loss of generality we write NPC$n(x_1, x_2, \dots, x_n) = (x_1 \vee x_2 \vee \dots \vee x_n) \oplus x_1 \oplus x_1 x_2 \dots x_n$ On the other hand, by De Morgan's laws the following two formulas hold:

$$(x_1' \vee x_2' \vee \dots \vee x_n') = (x_1 x_2 \dots x_n)' = 1 \oplus x_1 x_2 \dots x_n,$$
$$x_1' x_2' \dots x_n' = (x_1 \vee x_2 \vee \dots \vee x_n)' = 1 \oplus (x_1 \vee x_2 \vee \dots \vee x_n).$$

Thus,

$$\begin{aligned}(\mathrm{NPC}n)'(x_1', x_2', \dots, x_n') &= 1 \oplus [(x_1' \vee x_2' \vee \dots \vee x_n') \oplus x_1' \oplus x_1' x_2' \dots x_n'] \\ &= 1 \oplus [(x_1 x_2 \dots x_n)' \oplus x_1' \oplus (x_1 \vee x_2 \vee \dots \vee x_n)'] \\ &= 1 \oplus [1 \oplus x_1 x_2 \dots x_n \oplus 1 \oplus x_1 \oplus 1 \oplus (x_1 \vee x_2 \vee \dots \vee x_n)] \\ &= (x_1 \vee x_2 \vee \dots \vee x_n) \oplus x_1 \oplus x_1 x_2 \dots x_n \\ &= \mathrm{NPC}n(x_1, x_2, \dots, x_n).\end{aligned}$$

and by Definition 12 any NPCn is self-dual.

(3) From Lemma 2 it follows that it is self-complementary.

(4) P-equivalence follows from Lemma 5.

(5) Once again, without loss of generality we can write

$$\mathrm{NPC}n(x_1, x_2, \dots, x_n) = [(x_1 \vee x_2 \vee \dots \vee x_n) \oplus x_1] \oplus x_1 x_2 \dots x_n,$$

and transform it using well-known formulas $a \oplus b = ab' + a'b, aa' = 0$, $a = a \ \vee ab$, and De Morgan's laws:

$$\begin{aligned}&\mathrm{NPC}n(x_1, x_2, \dots, x_n) = [(x_1 \vee x_2 \vee \dots \vee x_n)x_1' \vee (x_1 \vee x_2 \vee \dots \vee x_n)' x_1] \oplus x_1 x_2 \dots x_n \\ &= [x_1' x_2 \vee x_1' x_3 \vee \dots \vee x_1' x_n \vee (x_1' x_2' \dots x_n') x_1] \oplus x_1 x_2 \dots x_n \\ &= (x_1' x_2 \vee x_1' x_3 \vee \dots \vee x_1' x_n)(x_1 x_2 \dots x_n)' \vee (x_1' x_2 \vee x_1' x_3 \vee \dots \vee x_1' x_n)'(x_1 x_2 \dots x_n) \\ &= x_1' x_2 \vee x_1' x_3 \vee \dots \vee x_1' x_n \vee x_2 \dots x_n.\end{aligned}$$

Thus, in the reduced SOP for NPCn the variable x_1 appears only as complemented and all the other variables are uncomplemented, i.e. NPCn is unate. □

Corollary 3. *For any $n \geq 3$ there exist reversible functions having all component functions being:*

(1) nonlinear,
(2) self-dual,
(3) self-complementary,
(4) P-equivalent,
(5) unate.

6 Computational Results

By running simple programs on a laptop we have obtained the computational results described in this section. The configuration of the laptop we used was standard: i7 processor and 4 GB of RAM. Each of the computational tasks took less than one hour. First we calculated in an exhaustive manner all NPN-equivalence classes of balanced Boolean functions of 1, 2, 3 and 4 variables. The results for $n = 1, 2, 3$ are shown in Table 3 and for $n = 4$ in Table 4 together with sizes and functional properties of all these classes. These results were published for the first time in [13]. Each row gives one equivalence class identified by its representative expressed in the form of PPRM expressions. For our purpose considering each component function separately is a more convenient form than permutation which is shorter but in which component functions are not shown explicitly. For each class, the table shows the number of variables (n), the name of the class (Class), the size of the equivalence class (Size), a *Representative* of the class, and the classical *Properties* the class possesses (the meanings of abbreviations L, LV, NL, SC and SD were introduced in Sect. 2). Equivalence classes are sorted first by the size of the number of terms in PPRM expression and in case of a tie by the sizes of the consecutive terms in the expression (the terms of the same size are given in the lexicographic order). To decrease the width of Tables 4 and 5 we used names a, b, c and d to denote variables (instead of x_1, x_2, x_3, x_4 which we use in the rest of the paper).

We have checked that only for the following 18 out of 58 classes of balanced Boolean functions up to 4 variables (B1.1-B4.52) it is *impossible* to find four functions belonging to the same class which would constitute a 4-variable reversible function: B2.1, B3.2, B4.2, B4.3, B4.4, B4.7 (this class includes only 2 functions), B4.13, B4.15, B4.27, B4.28, B.4.31, B4.33, B4.34, B4.35, B4.38, B4.42, B4.48, B4.51.

We used this result for extrapolation of some properties for a larger number of variables. We also expect that several interesting conjectures can be formulated on the basis of the above results.

We have also calculated all NPNP-equivalence classes of 3-variable reversible functions (see Table 5 organised under the same assumptions as Tables 3 and 4).

Table 3. NPN-equivalence classes of n-variable balanced Boolean functions for $n \leq 3$

Class	Size	Representative	Properties				
			L	LV	NL	SC	SD
$B1.1$	8	a	+	+		+	+
$B2.1$	12	$a \oplus b$	+	+		+	
$B3.1$	96	$a \oplus bc$		+	+	+	
$B3.2$	8	$a \oplus b \oplus c$	+	+		+	+
$B3.3$	96	$a \oplus ab \oplus bc$			+	+	
$B3.4$	32	$ab \oplus ac \oplus bc$			+	+	+

For the synthesis of reversible functions, NPNP-equivalence classes are interesting because permutations of component functions do not change values of cost functions of optimal reversible circuits implementing them. It is because a permutation of component functions leads to permutation of lines in the circuit which does not change the cost of the circuit.

As mentioned in Sect. 3 such a table was published in [19] but we were able to find (probably typographic) errors in it. One type of these errors consists in non-reversibility of two classes' representatives. To show precisely where the errors are located let us point that Lorens' Table VI is split into three parts based on properties of the inverses of the classes' representatives:

(A) 21 functions having their inverses identical to the function (called self-inverse functions),
(B) 3 classes of functions having their inverses in the same NPNP-equivalence class,
(C) 28 classes of functions having their inverses in a different NPNP-equivalence class.

It is easy to check that the following two classes' representatives from the Lorens' table are not reversible:

$$f_1 = x_1' \oplus x_2x_3, f_2 = x_2 \oplus x_1x_3', f_3 = x_3 \oplus x_1'\mathbf{x_2} \text{(Part A, row 16)},$$
$$f_1 = x_1x_2 \oplus x_2x_3 \oplus x_3x_1, f_2 = x_1 \oplus \mathbf{x_2x_3'}, f_3 = x_3 \oplus x_2x_1' \text{(Part C, column 1, row 13)}.$$

It seems that the correct expressions were supposed to be as follows:

$$f_1 = x_1' \oplus x_2x_3, f_2 = x_2 \oplus x_1x_3', f_3 = x_3 \oplus x_1'\mathbf{x_2'} \text{(adding a "prime" to the last literal)},$$
$$f_1 = x_1x_2 \oplus x_2x_3 \oplus x_3x_1, f_2 = x_1 \oplus \mathbf{x_2'x_3}, f_3 = x_3 \oplus x_2x_1' \text{(swapping the "prime" in the 2nd term in } f_2\text{)}.$$

The last two functions are reversible and belong to our classes R28 and R31, respectively, which are not covered by the other representatives in Table VI in [19].

In Lorens' Table VI we have also found two pairs of representatives that belong to the same class:

Table 4. NPN-equivalence classes of n-variable balanced Boolean functions for $n = 4$

Class	Size	Representative	Properties				
			L	LV	NL	SC	SD
$B4.1$	64	$a \oplus bcd$		+	+	+	
$B4.2$	48	$a \oplus b \oplus cd$		+	+	+	
$B4.3$	192	$a \oplus b \oplus acd$		+	+	+	
$B4.4$	96	$a \oplus bc \oplus bd$		+	+	+	
$B4.5$	768	$a \oplus bc \oplus abd$			+		
$B4.6$	192	$a \oplus abc \oplus bcd$			+	+	
$B4.7$	2	$a \oplus b \oplus c \oplus d$	+	+		+	
$B4.8$	192	$a \oplus b \oplus c \oplus abd$		+	+	+	
$B4.9$	96	$a \oplus b \oplus ac \oplus cd$		+	+	+	
$B4.10$	384	$a \oplus b \oplus cd \oplus abc$			+		
$B4.11$	384	$a \oplus b \oplus abc \oplus acd$			+	+	
$B4.12$	96	$a \oplus ab \oplus bc \oplus bd$			+	+	
$B4.13$	32	$a \oplus bc \oplus bd \oplus cd$		+	+	+	
$B4.14$	384	$a \oplus bc \oplus abc \oplus abd$			+	+	
$B1.15$	64	$a \oplus b \oplus c \oplus d \oplus abc$		+	+	+	
$B4.16$	192	$a \oplus b \oplus c \oplus abc \oplus abd$			+	+	
$B4.17$	32	$a \oplus b \oplus ac \oplus ad \oplus cd$		+	+	+	+
$B4.18$	384	$a \oplus b \oplus ab \oplus ac \oplus bcd$			+	+	
$B4.19$	384	$a \oplus b \oplus ac \oplus ad \oplus bcd$			+		
$B4.20$	384	$a \oplus b \oplus ac \oplus acd \oplus bcd$			+		
$B4.21$	384	$a \oplus b \oplus ac \oplus abd \oplus bcd$			+		
$B4.22$	384	$a \oplus b \oplus abc \oplus acd \oplus bcd$			+		
$B4.23$	48	$a \oplus ab \oplus ac \oplus bd \oplus cd$			+	+	+
$B4.24$	384	$a \oplus ab \oplus bc \oplus bd \oplus acd$			+		
$B4.25$	384	$a \oplus ab \oplus bc \oplus abd \oplus acd$			+		
$B4.26$	192	$a \oplus ab \oplus cd \oplus abc \oplus acd$			+	+	
$B4.27$	384	$a \oplus bc \oplus bd \oplus abd \oplus acd$			+	+	
$B4.28$	192	$a \oplus bc \oplus bd \oplus acd \oplus bcd$			+		
$B4.29$	192	$a \oplus ab \oplus abc \oplus abd \oplus bcd$			+	+	
$B4.30$	384	$a \oplus bc \oplus abc \oplus abd \oplus acd$			+		
$B4.31$	48	$ab \oplus ac \oplus ad \oplus bc \oplus bd$			+	+	
$B4.32$	384	$a \oplus b \oplus c \oplus ab \oplus ad \oplus bcd$			+	+	
$B4.33$	192	$a \oplus b \oplus c \oplus ad \oplus abc \oplus bcd$			+		
$B4.34$	384	$a \oplus b \oplus c \oplus ad \oplus abd \oplus bcd$			+	+	
$B4.35$	24	$a \oplus b \oplus ab \oplus ac \oplus bd \oplus cd$			+	+	
$B4.36$	384	$a \oplus b \oplus ac \oplus cd \oplus abc \oplus abd$			+	+	
$B4.37$	192	$a \oplus b \oplus ac \oplus cd \oplus abd \oplus acd$			+	+	
$B4.38$	192	$a \oplus b \oplus ab \oplus abc \oplus abd \oplus acd$			+	+	
$B4.39$	768	$a \oplus b \oplus ac \oplus abd \oplus acd \oplus bcd$			+		
$B4.40$	96	$a \oplus b \oplus abc \oplus abd \oplus acd \oplus bcd$			+		
$B4.41$	96	$a \oplus ab \oplus ac \oplus bc \oplus bd \oplus cd$			+	+	
$B4.42$	192	$a \oplus ab \oplus bc \oplus bd \oplus acd \oplus bcd$			+		
$B4.43$	384	$a \oplus ab \oplus bc \oplus cd \oplus abd \oplus acd$			+	+	
$B4.44$	384	$a \oplus ab \oplus cd \oplus abc \oplus abd \oplus acd$			+		
$B4.45$	384	$a \oplus bc \oplus bd \oplus abc \oplus acd \oplus bcd$			+	+	
$B4.46$	64	$a \oplus b \oplus c \oplus ad \oplus abd \oplus acd \oplus bcd$			+	+	
$B4.47$	384	$a \oplus b \oplus ac \oplus cd \oplus abc \oplus abd \oplus acd$			+	+	
$B4.48$	192	$a \oplus b \oplus ab \oplus abc \oplus abd \oplus acd \oplus bcd$			+		
$B4.49$	384	$a \oplus b \oplus ac \oplus abc \oplus abd \oplus acd \oplus bcd$			+	+	
$B4.50$	64	$a \oplus ab \oplus cd \oplus abc \oplus abd \oplus acd \oplus bcd$			+	+	+
$B4.51$	64	$a \oplus b \oplus c \oplus d \oplus ab \oplus abc \oplus abd \oplus acd$			+	+	
$B4.52$	64	$a \oplus b \oplus c \oplus ab \oplus cd \oplus abc \oplus abd \oplus acd \oplus bcd$			+	+	+

Table 5. Representatives of NPNP-equivalence classes of reversible Boolean functions for $n = 3$

Class	Size	$f_1(a,b,c)$	$f_2(a,b,c)$	$f_3(a,b,c)$	BF classes
$R1$	48	a	b	c	1.1 1.1 1.1
$R2$	288	a	b	$a \oplus c$	1.1 1.1 2.1
$R3$	576	a	b	$c \oplus ab$	1.1 1.1 3.1
$R4$	144	a	b	$a \oplus b \oplus c$	1.1 1.1 3.2
$R5$	144	a	$a \oplus b$	$a \oplus c$	1.1 2.1 2.1
$R6$	288	a	$a \oplus b$	$b \oplus c$	1.1 2.1 2.1
$R7$	1152	a	$a \oplus b$	$c \oplus ab$	1.1 2.1 3.1
$R8$	288	a	$a \oplus b$	$a \oplus b \oplus c$	1.1 2.1 3.2
$R9$	576	a	$b \oplus c$	$b \oplus ab \oplus ac$	1.1 2.1 3.3
$R10$	1152	a	$b \oplus ac$	$b \oplus c \oplus ab$	1.1 3.1 3.1
$R11$	576	a	$b \oplus ac$	$b \oplus c \oplus ac$	1.1 3.1 3.1
$R12$	2304	a	$b \oplus ac$	$c \oplus ab \oplus ac$	1.1 3.1 3.3
$R13$	576	a	$b \oplus ac$	$a \oplus b \oplus c \oplus ac$	1.1 3.2 3.1
$R14$	576	a	$a \oplus b \oplus c$	$b \oplus ab \oplus ac$	1.1 3.2 3.3
$R15$	288	a	$b \oplus ab \oplus ac$	$c \oplus ab \oplus ac$	1.1 3.3 3.3
$R16$	288	a	$b \oplus ab \oplus ac$	$a \oplus c \oplus ab \oplus ac$	1.1 3.3 3.3
$R17$	144	$a \oplus b$	$a \oplus c$	$a \oplus b \oplus c$	2.1 2.1 3.2
$R18$	576	$a \oplus b$	$a \oplus c$	$ab \oplus ac \oplus bc$	2.1 2.1 3.4
$R19$	576	$a \oplus b$	$c \oplus ab$	$a \oplus c \oplus ab$	2.1 3.1 3.1
$R20$	1152	$a \oplus b$	$c \oplus ab$	$a \oplus ac \oplus bc$	2.1 3.1 3.3
$R21$	1152	$a \oplus b$	$c \oplus ab$	$a \oplus c \oplus ab \oplus ac \oplus bc$	2.1 3.1 3.4
$R22$	576	$a \oplus b$	$a \oplus b \oplus c$	$a \oplus ac \oplus bc$	2.1 3.2 3.3
$R23$	288	$a \oplus b$	$a \oplus ac \oplus bc$	$a \oplus c \oplus ac \oplus bc$	2.1 3.3 3.3
$R24$	288	$a \oplus b$	$a \oplus ac \oplus bc$	$b \oplus c \oplus ac \oplus bc$	2.1 3.3 3.3
$R25$	1152	$a \oplus b$	$a \oplus ac \oplus bc$	$a \oplus c \oplus ab \oplus ac \oplus bc$	2.1 3.3 3.4
$R26$	576	$a \oplus b$	$ab \oplus ac \oplus bc$	$a \oplus c \oplus ab \oplus ac \oplus bc$	2.1 3.4 3.4
$R27$	2304	$a \oplus bc$	$a \oplus b \oplus ac$	$c \oplus ab \oplus ac$	3.1 3.1 3.3
$R28$	384	$a \oplus bc$	$a \oplus b \oplus ac$	$a \oplus b \oplus c \oplus ab$	3.1 3.1 3.1
$R29$	1152	$a \oplus bc$	$a \oplus b \oplus ac$	$a \oplus b \oplus c \oplus ac$	3.1 3.1 3.1
$R30$	1152	$a \oplus bc$	$a \oplus b \oplus ac$	$b \oplus c \oplus ac \oplus bc$	3.1 3.1 3.3
$R31$	1152	$a \oplus bc$	$a \oplus b \oplus ac$	$a \oplus c \oplus ab \oplus ac \oplus bc$	3.1 3.1 3.4
$R32$	576	$a \oplus bc$	$a \oplus b \oplus bc$	$a \oplus c \oplus bc$	3.1 3.1 3.1
$R33$	1152	$a \oplus bc$	$a \oplus b \oplus bc$	$c \oplus ab \oplus bc$	3.1 3.1 3.3
$R34$	576	$a \oplus bc$	$b \oplus ab \oplus ac$	$c \oplus ab \oplus ac$	3.1 3.3 3.3
$R35$	2304	$a \oplus bc$	$b \oplus ab \oplus ac$	$c \oplus ab \oplus bc$	3.1 3.3 3.3
$R36$	576	$a \oplus bc$	$b \oplus ab \oplus ac$	$a \oplus b \oplus c \oplus bc$	3.1 3.3 3.1
$R37$	1152	$a \oplus bc$	$b \oplus ab \oplus ac$	$a \oplus c \oplus ab \oplus ac$	3.1 3.3 3.3
$R38$	1152	$a \oplus bc$	$b \oplus ac \oplus bc$	$b \oplus c \oplus ab \oplus bc$	3.1 3.3 3.3
$R39$	1152	$a \oplus bc$	$b \oplus ac \oplus bc$	$b \oplus c \oplus ac \oplus bc$	3.1 3.3 3.3
$R40$	2304	$a \oplus bc$	$b \oplus ac \oplus bc$	$a \oplus c \oplus ab \oplus ac \oplus bc$	3.1 3.3 3.4
$R41$	576	$a \oplus bc$	$a \oplus b \oplus c \oplus bc$	$a \oplus b \oplus ab \oplus ac \oplus bc$	3.1 3.1 3.4
$R42$	576	$a \oplus bc$	$a \oplus b \oplus ab \oplus ac \oplus bc$	$a \oplus c \oplus ab \oplus ac \oplus bc$	3.1 3.4 3.4
$R43$	1152	$a \oplus b \oplus c$	$a \oplus ab \oplus bc$	$b \oplus ac \oplus bc$	3.2 3.3 3.3
$R44$	288	$a \oplus b \oplus c$	$a \oplus ab \oplus bc$	$c \oplus ab \oplus bc$	3.2 3.3 3.3
$R45$	288	$a \oplus b \oplus c$	$a \oplus ab \oplus bc$	$a \oplus b \oplus ab \oplus bc$	3.2 3.3 3.3
$R46$	384	$a \oplus ab \oplus bc$	$b \oplus ac \oplus bc$	$c \oplus ab \oplus ac$	3.3 3.3 3.3
$R47$	1152	$a \oplus ab \oplus bc$	$b \oplus ac \oplus bc$	$a \oplus c \oplus ac \oplus bc$	3.3 3.3 3.3
$R48$	1152	$a \oplus ab \oplus bc$	$b \oplus ac \oplus bc$	$b \oplus c \oplus ab \oplus ac \oplus bc$	3.3 3.3 3.4
$R49$	576	$a \oplus ab \oplus bc$	$c \oplus ab \oplus bc$	$a \oplus b \oplus ab \oplus bc$	3.3 3.3 3.3
$R50$	576	$a \oplus ab \oplus bc$	$c \oplus ab \oplus bc$	$a \oplus b \oplus ab \oplus ac \oplus bc$	3.3 3.3 3.4
$R51$	576	$a \oplus ab \oplus bc$	$a \oplus b \oplus ab \oplus ac \oplus bc$	$b \oplus c \oplus ab \oplus ac \oplus bc$	3.3 3.4 3.4
$R52$	192	$ab \oplus ac \oplus bc$	$a \oplus b \oplus ab \oplus ac \oplus bc$	$a \oplus c \oplus ab \oplus ac \oplus bc$	3.4 3.4 3.4

– one pair is in Part A, rows 13 and 14 (both representatives belong to our class R40),
– the other pair: Part C, column 1, row 10, and Part C, column 2, row 11 (both representatives belong to our class R45).

Thus, 4 out of 52 NPNP-equivalence classes of 3∗3 reversible functions are not represented in Lorens' Table VI.

In Table 4, in comparison with [19], we added sizes of the classes and information showing to which NPN-equivalence class of balanced functions each of the component functions belongs. The latter information was useful in our extrapolation of some properties of reversible functions [16] (for an example see Sect. 7).

7 Extrapolation Based on Cycle Structures

In [11,12] it has been demonstrated that it is possible to extrapolate some properties of reversible functions by considering their cycle structures. This is why we tried to exploit the same approach to discover infinite sequences of reversible functions with all their component functions being non-degenerate and belonging to different P-classes. We established that there are 26 NPNP-classes of 3-variable functions (R27-R52) that possess all component functions depending essentially on all three variables. Among them, there is only one class that consists of reversible functions all whose component functions belong to different NPN-classes. Below the PPRM expressions for a member of this class are shown:

NPNP-class R40

$$A = a \oplus c \oplus ab \oplus ac \oplus bc,$$
$$B = b \oplus ab \oplus ac,$$
$$C = c \oplus ab.$$

The above PPRM expressions show some regular features. However, our experience is so that extrapolation of such features of PPRMs is very difficult because: (1) usually a component function is obtained which is not balanced, (2) even if all PPRMs correspond to balanced functions then their collection does not constitute a reversible function. Therefore we have decided to apply extrapolation based on cycle structures. By considering the appropriate mappings $\{0,1\}^3 \rightarrow \{0,1\}^3$ it is easy to establish that the earlier defined member of the NPNP-class R40 has the following cycle structure:

$$< 000 > < 010 > < 011 > < 100 > < 001, 101, 111, 110 > .$$

Let us note that binary n-tuples in the unique cycle having more than one element form a regular pattern:

$$001,$$
$$101,$$
$$111,$$
$$110.$$

Namely, it is easy to note that

- the first and the second n-tuples differ only in the 1st bit position,
- the second and the third n-tuples differ only in the 2nd bit position,
- the third and the fourth n-tuples differ only in the 3rd bit position.

Thus, we observe here a certain periodicity which can be easily extrapolated leading to the desired infinite sequence of reversible functions as will be seen later. In this case, extrapolating was quite simple. Let us introduce additional notions.

Definition 16. *A set of variable assignments over* $\{0,1\}$ *with specified numbers of* p *0s and* r *1s is called a block and denoted by* $b_{p,r}$.

Example 10. *The set of all eight variable assignments for 3-variable Boolean functions can be partitioned into the following four blocks:*

$$b_{3,0} = \{000\}, \quad b_{2,1} = \{001, 010, 100\}, \quad b_{1,2} = \{011, 101, 110\}, \quad b_{0,3} = \{111\}.$$

∎

Definition 17. *For any Boolean function* f *let* $B^0(f)$ *and* $B^1(f)$ *denote the sets of blocks including all variable assignments for which* f *is equal 0 and 1, respectively.*

Example 11. *Let us consider the following Boolean projection functions:*

$$f(x_1, x_2, x_3) = x_1, \quad g(x_1, x_2, x_3) = x_2, \quad h(x_1, x_2, x_3) = x_3.$$

Then,

$$B^0(f) = \{\{000\}, \{001, 010\}, \{011\}\}, \quad B^1(f) = \{\{100\}, \{101, 110\}, \{111\}\},$$
$$B^0(g) = \{\{000\}, \{001, 100\}, \{101\}\}, \quad B^1(g) = \{\{010\}, \{011, 110\}, \{111\}\},$$
$$B^0(h) = \{\{000\}, \{010, 100\}, \{110\}\}, \quad B^1(h) = \{\{001\}, \{011, 101\}, \{111\}\}.$$

Notice that for each 3-variable Boolean reversible function k *the union of* $B^0(k)$ *and* $B^1(k)$ *is equal to the set of all 8 Boolean variable assignments. For each of the component functions of an arbitrary reversible function cardinalities of unions of their* B^i *sets are the same.* ∎

Example 12. *Let us consider a 3-variable Boolean reversible function $F(x_1, x_2, x_3) = (f_1, f_2, f_3)$ defined in such a manner that the only non-identical mappings of variable assignments in F are as follows*

$$\begin{aligned} 001 &\rightarrow 101, \\ 101 &\rightarrow 111, \\ 111 &\rightarrow 110, \\ 110 &\rightarrow 001. \end{aligned}$$

When we consider the reversible function F as a permutation of output assignments it is a single cycle of four elements:

$$< 001, 101, 111, 110 > .$$

Notice that in the above mappings

- *in the 1st row the leftmost bit is being negated,*
- *in the 2nd row the second bit is being negated*
- *in the 3rd row the third bit is being negated*
- *in the 4th row all bits are being negated.*

This observation will be generalised later to functions of any number of variables.

Now let us note what changes have been done in the sets B_i, $0 \leq i \leq 1$, for functions f_1, f_2, and f_3, in comparison with the sets for the function in Example 11 (the assignments moved to another block are shown bolded and underlined):

$$B^0(f_1) = \{\{000\}, \{010\}, \{011, \underline{\mathbf{110}}\}\}, \qquad B^1(f_1) = \{\{\underline{\mathbf{001}}, 100\}, \{101\}, \{111\}\},$$
$$B^0(f_2) = \{\{000\}, \{001, 100\}, \{\underline{\mathbf{110}}\}\}, \qquad B^1(f_2) = \{\{010\}, \{011, \underline{\mathbf{101}}\}, \{111\}\},$$
$$B^0(f_3) = \{\{000\}, \{010, 100\}, \{\underline{\mathbf{111}}\}\}, \qquad B^1(f_3) = \{\{001\}, \{011, 101\}, \{\underline{\mathbf{110}}\}\}.$$

Let us summarise the above observations.

The values of the function f_1 differ from the values of the projection function x_1 only for the assignments 001 and 110. Namely, we can notice that

$$f(0,0,1) = 0 \qquad f_1(0,0,1) = 1,$$
$$f(1,1,0) = 1 \qquad f_1(1,1,0) = 0.$$

As a result, the function f_1 can be obtained from the projection function x_1 by swapping its values for variable assignments 001 and 110.

Values of each of the other two component functions, f_2 and f_3, also differ from the values of the corresponding projection functions only for two assignments.

Swaps for f_2 in comparison with the projection function x_2 are as follows:

$$g(1,0,1)=0 \qquad f_2(1,0,1)=1,$$
$$g(1,1,0)=1 \qquad f_2(1,1,0)=0.$$

Swaps for f_3 in comparison with the projection function x_3 are as follows:

$$h(1,1,1)=1 \qquad f_3(1,1,1)=0,$$
$$h(1,1,0)=0 \qquad f_3(1,1,0)=1.$$

Let us show that component functions f_1 and f_2 belong to different P-equivalence classes. Assume that f_1 and f_2 belong to the same P-equivalence class. Then, since any permutation over the variable set $\{x_1, x_2, x_3\}$ does not change the assignment 111 *there should be $f_1(1,1,1) = f_2(1,1,1)$, however, $f_1(1,1,1) = 0$ and $f_2(1,1,1) = 1$. It is in contradiction with our assumption that f_1 and f_2 belong to the same P-equivalence class. Thus, f_1 and f_2 belong to different P-equivalence classes.*

In a similar manner it can be shown that the other two pairs of component functions of F, (f_1, f_3) and (f_2, f_3), belong to different P-equivalence classes.

Let us show that component functions f_1 and f_3 belong to different P-equivalence classes. Assume that f_1 and f_3 belong to the same P-equivalence class. Then, since any permutation over variable set $\{x_1, x_2, x_3\}$ does not change the assignment 111 *there should be $f_1(1,1,1) = f_3(1,1,1)$, however $f_1(1,1,1) = 0$ and $f_3(1,1,1) = 1$. It is in contradiction with our assumption that f_1 and f_3 belong to the same P-equivalence class. Thus, f_1 and f_3 belong to different P-equivalence classes.*

Let us show that component functions f_2 and f_3 belong to different P-equivalence classes. Assume that f_2 and f_3 belong to the same P-equivalence class. Then, let us consider the permutation of variables consisting in swapping variables x_2 and x_3. Then there should be $f_2(1,0,0) = f_3(1,0,0)$, however $f_2(1,0,0) = 0$ and $f_3(1,0,0) = 1$. It is in contradiction with our assumption that f_2 and f_3 belong to the same P-equivalence class. Thus, f_2 and f_3 belong to different P-equivalence classes. ∎

Now the presented above methodology of proving that two component functions of F belong to different P-equivalence classes will be extended to Boolean reversible functions of any number of variables. To prove that Boolean reversible functions with all component functions belonging to different P-equivalence classes exist for any number of variables $n \geq 3$, we will define the following infinite sequence of reversible functions:

Definition 18. *The reversible Boolean function $H^n(x_1, x_2, \ldots, x_n) = (f_1, f_2, \ldots, f_n)$, $n \geq 3$, is defined in such a manner that the only non-identical mappings of variable assignments in H^n are as follows:*

$$\begin{array}{lcl} a_1\ a_2 \ldots a_{n-1}\ a_n & \rightarrow & a'_1\ a_2 \ldots a_{n-1}\ a_n, \\ a'_1\ a_2 \ldots a_{n-1}\ a_n & \rightarrow & a'_1\ a'_2 \ldots a_{n-1}\ a_n, \\ \ldots & & \\ a'_1\ a'_2 \ldots a'_{n-1}\ a_n & \rightarrow & a'_1\ a'_2 \ldots a'_{n-1}\ a'_n, \\ a'_1\ a'_2 \ldots a'_{n-1}\ a'_n & \rightarrow & a_1\ a_2 \ldots a_{n-1}\ a_n, \end{array}$$

where the starting variable assignment is as follows:

$$a_1 a_2 a_3 \ldots a_{n-1} a_n = 000 \ldots 01.$$

Notice that in the ith row of the mappings in Definition 18, $1 \leq i \leq n$, the ith bit is being negated, and in the last mapping, all bits are being negated.

When we consider the function H^n as a permutation of variable assignments it is a cycle of $n + 1$ elements:

$$\begin{array}{l} < a_1, a_2, \ldots, a_{n-1}, a_n, \\ \quad a'_1, a_2, \ldots, a_{n-1}, a_n, \\ \quad a'_1, a'_2, \ldots, a_{n-1}, a_n, \\ \quad \ldots \\ \quad a'_1, a'_2, \ldots, a'_{n-1}, a_n, \\ \quad a'_1, a'_2, \ldots, a'_{n-1}, \ a'_n > . \end{array}$$

Theorem 7. *Each $n{*}n$ function H^n is reversible for any $n \geq 3$, where H^n is formulated in Definition 18.*

Proof. Because non-identical mappings of variable assignments in H^n form a cycle, this function is bijective for any $n \geq 3$. Hence, it is reversible. □

In a manner similar to Example 12 we proved in [16] that the following result holds.

Theorem 8. *Any two component functions of the Boolean reversible function H^n belong to different P-equivalence classes for $n \geq 3$.*

It is obvious that by Theorem 8 the following result holds:

Corollary 4. *For any $n \geq 3$ there exist binary reversible functions having all component functions that belong to different P-equivalence classes.*

8 Conclusions and Future Work

The chapter presents our results on properties of component functions of Boolean reversible functions. The solved problems were described briefly in Sects. 4, 5, 6 and 7. They can be summarised as follows:

(A) For any $n \geq 3$ there does not exist a Boolean reversible function with all component functions being non-degenerate and

- totally symmetric,
- linear/affine,
- monotone,
- majority,
- threshold.

(B) For any $n \geq 3$ there exists a Boolean reversible function with all component functions being nondegenerate and
- nonlinear,
- self-complementary,
- self-dual,
- unate.
- P-equivalent.

(C) For any $n \geq 3$ there exists a Boolean reversible function with all component functions being non-degenerate and belonging to different P-equivalence classes.

Our work has not been finished. We plan to continue efforts for constructing a classification of reversible Boolean functions which would be useful in the synthesis of reversible circuits.

Acknowledgements. The authors acknowledge partial support of COST Action IC1405 on “Reversible Computation - Extending Horizons of Computing”. They are grateful to Philipp Niemann, one of the reviewers, for many comprehensive remarks.

References

1. Aaronson, S., Grier, D., Schaeffer, L.: The classification of reversible bit operations. Preprint arXiv:1504.05155 [quant-ph] (2015)
2. Carić, M., Živković, M.: On the number of equivalence classes of invertible Boolean functions under action of permutation of variables on domain and range. Publications de l’Institut Mathématique **100**(114), 95–99 (2016)
3. Carlet, C.: Vectorial Boolean functions for cryptography. In: Crama, Y., Hammer, P. (eds.) Boolean Models and Methods in Mathematics, Computer Science, and Engineering, pp. 398–472. Cambridge University Press, Cambridge (2010)
4. De Vos, A.: Reversible Computing: Fundamentals, Quantum Computing, and Applications. Wiley-VCH Verlag, Weinheim (2010)
5. Debnath, D., Sasao, T.: Fast Boolean matching under variable permutation using representative. In: Proceedings of the Asia and South Pacific Design Automation Conference, pp. 359–362 (1999)
6. Debnath, D., Sasao, T.: Efficient computation of canonical form for Boolean matching in large libraries. In: Proceedings of the Asia and South Pacific Design Automation Conference, pp. 591–596 (2004)
7. Debnath, D., Sasao, T.: Fast Boolean matching under permutation by efficient computation of canonical form. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E87–A**(12), 3134–3140 (2004)
8. Debnath, D., Sasao, T.: Efficient computation of canonical form under variable permutation and negation for Boolean matching in large libraries. IEICE Trans. Fundam. Electron. Commun. Comput. Sci. **E89–A**(12), 3443–3450 (2006)

9. Draper, T.G.: Nonlinear complexity of Boolean permutations. Ph.D. thesis, Department of Mathematics, University of Maryland, College Park, Maryland, USA (2009)
10. Harrison, M.A.: The number of classes of invertible Boolean functions. J. ACM **10**, 25–28 (1963)
11. Jegier, J., Kerntopf, P.: Progress towards constructing sequences of benchmarks for quantum Boolean circuits synthesis. In: Proceedings of the 14th IEEE International Conference on Nanotechnology, pp. 250–255 (2014)
12. Jegier, J., Kerntopf, P., Szyprowski, M.: An approach to constructing reversible multi-qubit benchmarks with provably minimal implementations. In: Proceedings of the 13th IEEE International Conference on Nanotechnology, pp. 99–104 (2013)
13. Kerntopf, P., Moraga, C., Podlaski, K., Stanković, R.: Towards classification of reversible functions. In: Steinbach, B. (ed.) Proceedings of the 12th International Workshop on Boolean Problems, pp. 21–28 (2018)
14. Kerntopf, P., Moraga, C., Podlaski, K., Stanković, R.: Towards classification of reversible functions with homogeneous component functions. In: Steinbach, B. (ed.) Further Improvements in the Boolean Domain, pp. 386–406. Cambridge Scholars Publishing, Newcastle upon Tyne (2018)
15. Kerntopf, P., Podlaski, K., Moraga, C., Stanković, R.: Study of reversible ternary functions with homogeneous component functions. In: Proceedings of the 47th IEEE International Conference on Multiple-Valued Logic, pp. 191–196 (2017)
16. Kerntopf, P., Podlaski, K., Moraga, C., Stanković, R.: New results on reversible Boolean functions having component functions with specified properties. In: Drechsler, R., Soeken, M. (eds.) Advanced Boolean Techniques, pp. 217–236. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-20323-8_10
17. Kerntopf, P., Stanković, R., Podlaski, K., Moraga, C.: Ternary/MV reversible functions with component functions from different equivalence classes. In: Proceedings of the 48th IEEE International Conference on Multiple-Valued Logic, pp. 109–114 (2018)
18. Lorens, C.S.: Invertible Boolean functions. Tech. rep. 21, Space-General Corp., El Monte, CA, Research Memorandum (1962)
19. Lorens, C.S.: Invertible Boolean functions. IEEE Trans. Electron. Comput. **EC–13**(5), 529–541 (1964)
20. Primenko, E.A.: Invertible Boolean functions and fundamental groups of transformations of algebras of Boolean functions. Avtomatika & Vychislitelnaya Tekhnika **3**, 17–21 (1976)
21. Primenko, E.A.: On the number of types of invertible Boolean functions. Avtomatika & Vychislitelnaya Tekhnika **6**, 12–14 (1977)
22. Primenko, E.A.: On the number of types of invertible transformations in multivalued logic. Kibernetika **5**, 27–29 (1977)
23. Primenko, E.A.: Equivalence classes of invertible Boolean functions. Kibernetika **6**, 1–5 (1984)
24. Rice, J.E.: Considerations for determining a classification scheme for reversible Boolean functions. Tech. rep. TR-CSJR2-2007, Department of Mathematics and Computer Science, University of Lethbridge, Lethbridge, Alberta, Canada (2007)
25. Saeedi, M., Markov, I.L.: Synthesis and optimization of reversible circuits: a survey. ACM Comput. Surv. **45**(2), 21:1–21:34 (2013)
26. Soeken, M., Abdessaied, N., De Micheli, G.: Enumeration of reversible functions and its application to circuit complexity. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 255–270. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_19

27. Soeken, M., Wille, R., Keszocze, O., Miller, D.M., Drechsler, R.: Embedding of large Boolean functions for reversible logic. ACM J. Emerg. Technol. Comput. Syst. **12**(4), 41:1–41:26 (2015)
28. Stanković, R.S., Astola, J.T., Steinbach, B.: Former and recent work in classification of switching functions. In: Steinbach, B. (ed.) Proceedings of the 8th International Workshop on Boolean Problems, pp. 115–126 (2008)
29. Strazdins, I.E.: On the number of types of invertible binary networks. Avtomatika & Vychislitelnaya Tekhnika **1**, 30–34 (1974)
30. Tokareva, N.: Bent Functions, Results and Applications to Cryptography. Academic Press, London (2015)
31. Tsai, C.C., Marek-Sadowska, M.: Boolean functions classification via fixed polarity Reed-Muller forms. IEEE Trans. Comput. **46**(2), 173–186 (1997)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

A Case Study for Reversible Computing: Reversible Debugging of Concurrent Programs

James Hoey[1], Ivan Lanese[2], Naoki Nishida[3], Irek Ulidowski[1], and Germán Vidal[4](✉)

[1] School of Informatics, University of Leicester, Leicester, UK
{jbh11/iu3}@leicester.ac.uk

[2] Focus Team, University of Bologna/Inria, Bologna, Italy
ivan.lanese@gmail.com

[3] Graduate School of Informatics, Nagoya University, Nagoya, Japan
nishida@i.nagoya-u.ac.jp

[4] MiST, VRAIN, Universitat Politècnica de València, Valencia, Spain
gvidal@dsic.upv.es

Abstract. Reversible computing allows one to run programs not only in the usual forward direction, but also backward. A main application area for reversible computing is debugging, where one can use reversibility to go backward from a visible misbehaviour towards the bug causing it. While reversible debugging of sequential systems is well understood, reversible debugging of concurrent and distributed systems is less settled. We present here two approaches for debugging concurrent programs, one based on *backtracking*, which undoes actions in reverse order of execution, and one based on *causal consistency*, which allows one to undo any action provided that its consequences, if any, are undone beforehand. The first approach tackles an imperative language with shared memory, while the second one considers a core of the functional message-passing language Erlang. Both the approaches are based on solid formal foundations.

1 Introduction

Reversible computing has been attracting interest due to its applications in fields as different as, e.g., hardware design [12], computational biology [4], quantum computing [2], discrete simulation [6] and robotics [31].

One of the oldest and more explored application areas for reversible computing is *program debugging*. This can be explained by looking, on the one hand, to

This work has been partially supported by COST Action IC1405 on Reversible Computation - Extending Horizons of Computing. The second author has been partially supported by the French National Research Agency (ANR), project DCore n. ANR-18-CE25-0007. The third author has been partially supported by JSPS KAKENHI Grant Number JP17H01722. The last author has been partially supported by the EU (FEDER) and the *Spanish MICINN/AEI* under grant TIN2016-76843-C4-1-R and by the *Generalitat Valenciana* under grant Prometeo/2019/098 (DeepTrust).

© The Author(s) 2020
I. Ulidowski et al. (Eds.): RC 2020, LNCS 12070, pp. 108–127, 2020.
https://doi.org/10.1007/978-3-030-47361-7_5

the relevance of the problem, and, on the other hand, to how naturally reversible computing fits in the picture. Concerning the former, finding and fixing bugs inside software has always been a main activity in the software development life cycle. Indeed, according to a 2014 study [47], the cost of debugging amounts to $312 billions annually. Another recent study [3] estimates that the time spent in debugging is 49.9% of the total programming time. Concerning how naturally reversible computing fits in this context, consider that debugging means finding a bug, i.e., some wrong line of code, causing some visible misbehaviour, i.e., a wrong effect of a program, such as a wrong message printed on the screen. In general, the execution of the wrong line precedes the wrong visible effect. For instance, a wrong assignment to a variable may imply a misbehaviour later on, when the value of the variable is printed on the screen. Usually, the programmer has a very precise idea about which line of code makes the misbehaviour visible, but a non trivial debugging activity may be needed to find the bug. Indeed, debugging practice requires to put a breakpoint before the line of code where the programmer thinks the bug is, and use step-by-step execution from there to find the wrong line of code. However, the guess of the location of the bug is frequently wrong, causing the breakpoint to occur too late (after the bug) and a new execution with an updated guess is often needed. Reversible debugging practice is more direct: first, run the program and stop when the visible misbehaviour is reached; then, execute backwards (possibly step-by-step) looking for the causes of the misbehaviour until the bug is found.

With these premises, it is no surprise that reversible debugging has been deeply explored, as shown for instance by the survey in [11]. Indeed, many debuggers provide features for reversible execution, including popular open source debuggers such as GDB [8] as well as tools from big corporations such as Microsoft, the case of WinDbg [34].

However, the problem is far less settled for concurrent and distributed programs. We remark that nowadays most of the software is concurrent, either since the platform is distributed, the case of Internet or the Cloud, or to overcome the advent of the power wall [46]. Finding bugs in concurrent and distributed software is more difficult than in sequential software [33], since faults may appear or disappear according to the speed of the different processes and of the network communications. The bugs generating these faults, called Heisenbugs, are thus particularly challenging because they are rather difficult to reproduce. Two approaches to reversible debugging of concurrent systems have been proposed. Using *backtracking*,[1] actions are undone in reverse order of execution, while using *causal-consistent reversibility* [25] actions can be undone in any order, provided that the consequences of a given action, if any, are undone beforehand. Note that, by exploring a computation back and forth using either backtracking or causal-consistent reversibility one is guaranteed that Heisenbugs that occurred in the computation will not disappear.

[1] Backtracking sometimes refer to the exploration of a set of possibilities: this is not the case here, since backward execution is (almost) deterministic.

This paper will present two lines of research on debugging for concurrent systems developed within the European COST Action IC1405 on "Reversible Computation - Extending Horizons of Computing" [23]. They share the use of state saving to enable backward computation (this is called a Landauer embedding [24], and it is needed to tackle languages which are irreversible) and a formal approach aiming at supporting debugging tools with a theory guaranteeing the desired properties. The first line of research [20–22] (Sect. 3) supports backtracking (apart from some non relevant actions) for a concurrent imperative language with shared memory, while the second line of research [28–30,36] (Sect. 4) supports causal-consistent reversibility for a core subset of the functional message-passing language Erlang. We will showcase both the approaches on the same airline booking example (Sect. 2), coded in the two languages. Related work is discussed in Sect. 5 and final remarks are presented in Sect. 6.

2 Airline Booking Example

In this section we will introduce an example program that contains a bug, and discuss a specific execution leading to a corresponding misbehaviour. This example will be used as running example throughout the paper. We will show this example in the two programming languages needed for the two approaches mentioned above. We begin by introducing each of these languages.

2.1 Imperative Concurrent Language

Our first language is much like any while language, consisting of assignments, conditional statements and while loops. Support has also been added for block statements containing the declaration of local variables and/or procedures, as well as procedure call statements. Further to this, *removal statements* are introduced to "clean up" at the end of a block, where any variables or procedures declared within the block are removed. Our language also contains unique names given to each conditional, loop, block, procedure declaration and call statement, named *construct identifiers* (represented as `i1.0`, `w1.0`, `b1.0`, etc.), and sequences of block names in which a given statement resides named *paths* (represented as `pa`). Both of these are used to handle variable scope, allowing one to distinguish different variables with the same name. The final addition to our language is *interleaving parallel composition.* A parallel statement, written `P par Q` allows the execution of the programs `P` and `Q` to interleave. All statements except blocks contain a stack `A` that is used to store identifiers (see below). The syntax of our language follows, where ε represents an empty program. Note that ε is the neutral element of sequential and parallel composition. We write `(pa,A)?` to denote the fact that `(pa,A)` is optional. We also write `In, Wn, Bn, Cn` to range, respectively, over identifiers for conditionals, while loops, blocks and call statements. Also, `n` refers to the name of a procedure.

$$
\begin{aligned}
\texttt{P} &::= \varepsilon \mid \texttt{S} \mid \texttt{P; P} \mid \texttt{P par P}\\
\texttt{S} &::= \texttt{skip (pa,A)?} \mid \texttt{X = E (pa,A)} \mid \texttt{if In B then P else Q end (pa,A)}\\
&\quad\mid \texttt{while Wn B do P end (pa,A)} \mid \texttt{begin Bn BB end} \mid \texttt{call Cn n (pa,A)}\\
\texttt{BB} &::= \texttt{DV; DP; P; RP; RV}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{DV} &::= \varepsilon \mid \texttt{var X = v (pa,A); DV} & \texttt{DP} &::= \varepsilon \mid \texttt{proc Pn n is P end (pa,A); DP}\\
\texttt{RV} &::= \varepsilon \mid \texttt{remove X = v (pa,A); RV} & \texttt{RP} &::= \varepsilon \mid \texttt{remove Pn n is P end (pa,A); RP}
\end{aligned}
$$

Operational Semantics. Our approach (see [20] for a detailed explanation) to reversing programs starts by producing two versions of the original program. The first one, named the *annotated version*, performs forward execution and saves any information that would be lost in a normal computation but is needed for inversion (named *reversal information* and saved into our auxiliary store δ). Identifiers are assigned to statements as we execute them, capturing the interleaving order needed for correct inversion. The second one, named the *inverted version*, executes forwards but simulates reversal using the reversal information as well as the identifiers to follow backtracking order. We comment here that we use 'inversion' to refer to both the process of producing the program code of the inverted version (program inverter [1]), and to the process of executing the inverted version of a program. A reverse execution computes all parallel statements as in a forward execution, but it uses identifiers to determine which statement to invert next (instead of nondeterministically deciding). For programs containing many nested parallel statements, the overhead of determining the correct interleaving order increases, though we still deem this as reasonable [19]. Note that using a nondeterministic interleaving for the reverse execution is not possible, since it is not guaranteed to behave correctly (e.g., requiring information from the auxiliary store that is not there may cause an execution to be stuck). However, a small number of execution steps, including closing a block and removing a skip, do not use an identifier and can therefore be interleaved nondeterministically during an inverse execution. Forward and reverse execution are each defined in terms of a non-standard, small step operational semantics. Our semantics perform both the expected execution (forward and reverse respectively) and all necessary saving/using of the reversal information. Consider the example rule [D1a] for assignments, which is a reversibilisation of the traditional irreversible semantics of an assignment statement [51].

$$
[\text{D1a}] \quad \frac{\texttt{m} = \texttt{next()} \quad (\texttt{e pa} \mid \delta, \sigma, \gamma, \Box) \hookrightarrow^{*}_{\texttt{a}} (\texttt{v} \mid \delta, \sigma, \gamma, \Box) \quad \mathit{evalV}(\gamma\texttt{,pa,X}) = \texttt{l}}{(\texttt{X = e (pa,A)} \mid \delta, \sigma, \gamma, \Box) \xrightarrow{m} (\texttt{skip m:A} \mid \delta[(\texttt{m},\sigma(\texttt{l})) \rightharpoonup \texttt{X}], \sigma[\texttt{l} \mapsto \texttt{v}], \gamma, \Box)}
$$

As shown here, this rule consists of the evaluation of the expression `e` to the value `v`, evaluation of the variable `X` to a memory location `l` and finally the assigning of the value `v` to the memory location `l` as expected. Alongside this, the rule also pushes the old value of the variable (the current value held at the memory location, namely $\sigma(\texttt{l})$) onto the stack for this variable name within δ ($\delta[(\texttt{m},\sigma(\texttt{l})) \rightharpoonup \texttt{X}]$, where $\rightharpoonup$ denotes a push operation). This old value is saved alongside the next available identifier `m`, returned via the function `next()` and used within the rule to record interleaving order (represented using the labelled

$$\begin{aligned}
program &::= fun_1 \;\ldots\; fun_n \\
fun &::= a_1(p_{11},\;\ldots,\;p_{1n_1}) \text{ when } g_1 \to e_1; \\
&\quad\;\ldots \\
&\quad\; a_m(p_{m1},\ldots,p_{mn_m}) \text{ when } g_m \to e_n\;. \\
e \ni expr &::= X \mid literal \mid [e_1|e_2] \mid \{e_1,\ldots,e_n\} \mid a(e_1,\ldots,e_n) \mid p = e \mid e_1, e_2 \\
&\;\mid\; \text{receive } c_1;\ldots;c_n \text{ end} \mid \text{spawn}(mod, a, [e_1,\ldots,e_n]) \mid e_1 \,!\, e_2 \mid \text{self}() \\
c \ni clause &::= p \text{ when } g \to e \qquad\qquad p \ni pat ::= X \mid literal \mid [p_1|p_2] \mid \{p_1,\ldots,p_n\}
\end{aligned}$$

Fig. 1. Language syntax rules

arrow $\xrightarrow{m}$). This identifier `m` is also inserted into the stack `A` corresponding to this specific assignment statement, represented as `m:A`.

Now consider the rule [D1r] from our inverse semantics for reversing assignments (that executed forwards via [D1a]).

$$[\text{D1r}] \quad \frac{\texttt{A} = \texttt{m:A}' \quad \texttt{m} = \texttt{previous()} \quad \delta(\texttt{X}) = \texttt{(m,v):X}' \quad evalV(\gamma\texttt{,pa,X}) = \texttt{l}}{(\texttt{X = e (pa,A)} \mid \delta, \sigma, \Box) \overset{m}{\rightsquigarrow} (\texttt{skip A}' \mid \delta[\texttt{X/X}'], \sigma[\texttt{l} \mapsto \texttt{v}], \Box)}$$

This rule first ensures this is the next statement to invert using the identifier `m`, which must match the last used identifier (`previous()`) and be present in both the statements stack ($\texttt{A} = \texttt{m:A}'$) and the auxiliary store alongside the old value ($\delta(\texttt{X}) = \texttt{(m,v):X}'$). Provided this is satisfied, this rule then removes all occurrences of `m`, and assigns the old value `v` retrieved from δ to the corresponding memory location. Note that `e` appears exactly as in the original version but it is not evaluated, and that the functions `next()` and `previous()` both update the next and previous identifiers respectively as a side effect.

2.2 Erlang

Our second approach deals with a relevant fragment of the functional and concurrent language Erlang. We show in Fig. 1 the syntax of its main constructs, focusing on the ones needed in our running example. We drop from the syntax some declarations related to module management, which are orthogonal to our purpose in this paper.

A program is a sequence of function definitions, where each function has a name (an *atom*, denoted by a) and is defined by a number of equations of the form $a_i(p_{i1},\;\ldots,\;p_{in_i})$ when $g_i \to e_i$, where $p_{i1},\;\ldots,\;p_{in_i}$ are *patterns* (i.e., terms built from variables and data constructors), g_i is a *guard* (typically an arithmetic or relational expression only involving built-in functions), and e_i is an arbitrary expression. As is common, the variables in $p_{i1},\;\ldots,\;p_{in_i}$ are the only variables that may occur free in g_i and e_i. The body of a function is an *expression*, which can include variables, literals (i.e., atoms, integers, floating point numbers, the empty list [ ], etc.), lists (using Prolog-like notation, i.e., $[e_1|e_2]$ is a list with head e_1 and tail e_2), tuples (denoted by $\{e_1,\ldots,e_n\}$),[2]

[2] The only data constructors in Erlang (besides literals) are the predefined functions for lists and tuples.

function applications (we do not consider higher order functions in this paper for simplicity), pattern matching, sequences (denoted by comma), receive expressions, spawn (for creating new processes), "!" (for sending a message), and self. Note that some of these functions are actually built-ins in Erlang.

In contrast to expressions, *patterns* are built from variables, literals, lists, and tuples. Patterns can only contain fresh variables. In turn, *values* are built from literals, lists, and tuples (i.e., values are ground patterns). In Erlang, variables start with an uppercase letter.

Let us now informally introduce the semantics of Erlang constructions. In the following, *substitutions* are denoted by Greek letters σ, θ, etc. A substitution σ denotes a mapping from variables to expressions, where $\mathcal{D}om(\sigma)$ is its domain. Substitution application $\sigma(e)$ is also denoted by $e\sigma$.

Given the pattern matching $p = e$, we first evaluate e to a value, say v; then, we check whether v matches p, i.e., there exists a substitution σ for the variables of p with $v = p\sigma$ (otherwise, an exception is raised). Then, the expression reduces to v, and variables are bound according to σ. Roughly speaking, a sequence $(p = e_1,\ e_2)$ is equivalent to the expression $\mathsf{let}\ p = e_1\ \mathsf{in}\ e_2$ in most functional programming languages.

A similar pattern matching operation is performed during a function application $a(e_1, \ldots, e_n)$. First, one evaluates $e_1, \ldots, e_n$ to values, say $v_1, \ldots, v_n$. Then, we scan the left-hand sides of the equations defining the function a until we find one that matches $a(v_1, \ldots, v_n)$. Let $a(p_1, \ldots, p_n)\ \mathsf{when}\ g \rightarrow e$ be such equation, with $a(v_1, \ldots, v_n) = a(p_1, \ldots, p_n)\sigma$. Here, we should also check that the guard, $g\sigma$, reduces to true. In this case, execution proceeds with the evaluation of the function's body, $e\sigma$.

Let us now consider the concurrent features of our language. In Erlang, a running system can be seen as a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Received messages are stored in the queues of processes until they are consumed; namely, each process has one associated local (FIFO) queue. A process is uniquely identified by its *pid* (process identifier). Message sending is asynchronous, while receive instructions block the execution of a process until an appropriate message reaches its local queue (see below).

We consider the following functions with side-effects: self, "!", spawn, and receive. The expression self() returns the pid of a process, while p ! v evaluates to v and, as a side-effect, sends message v to the process with pid p, which will be eventually stored in p's local queue. New processes are spawned with a call of the form $\mathsf{spawn}(mod, a, [v_1, \ldots, v_n])$, where *mod* is the name of the module declaring function a, and the new process begins with the evaluation of the function application $a(v_1, \ldots, v_n)$. The expression $\mathsf{spawn}(mod, a, [v_1, \ldots, v_n])$ returns the (fresh) pid assigned to the new process.

Finally, an expression "$\mathsf{receive}\ p_1\ \mathsf{when}\ g_1 \rightarrow e_1; \ldots; p_n\ \mathsf{when}\ g_n \rightarrow e_n\ \mathsf{end}$" should find the *first* message v in the process' queue (if any) such that v matches some pattern p_i (with substitution σ) and the instantiation of the corresponding guard $g_i\sigma$ reduces to true. Then, the receive expression evaluates to $e_i\sigma$, with

the side effect of deleting the message v from the process' queue. If there is no matching message in the current queue, the process *suspends* until a matching message arrives.

2.3 Airline Code

We are now ready to describe the example. Consider a model of an airline booking system, where multiple agents sell tickets for the same flight. In order to keep the example concise, we consider only two agents selling tickets in parallel, with three seats initially available. The code of the example is shown in Listing 1.1, written in the concurrent imperative programming language described in Sect. 2.1.

The code contains two while loops operating in parallel (lines 10–16 and 18–24), where each loop models the operation of a single agent. Let us consider the first loop. For each iteration, the agent checks whether any seat remains (line 11). As long as the number of currently available seats is greater than zero, the agent is free to sell a ticket via the procedure named `sell` (called at line 12). Once the number of available tickets has reached zero, each agent will then close, terminating its loop.

As previously mentioned, this program can show a misbehaviour under certain execution paths. Recall the simplified setting of three initially available seats. Consider an execution that begins with each agent selling a single ticket (allocating one seat) via one full iteration of each while loop (the interleaving among the two iterations is not relevant). At this point, both agents remain open (since `agent1` = `1` and `agent2` = `1`), and the current number of seats is `1`. Now assume that the execution continues with the following interleaving. The condition of each while loop is checked, both of which will evaluate to true as each agent is open. Next, the execution of each loop body begins with the evaluation of the guard of each conditional statement. They will both evaluate to true, as there is at least one seat available. At this point, each agent is committed to selling one more ticket, even if only one seat is available. The rest of the execution can then be finished under any interleaving. The important thing to note here is that the final number of free seats is `-1`. This is an obvious misbehaviour, as the two agents allocated four tickets when only three seats were available. This misbehaviour occurs since the programmer assumed that the checking for an available seat and its allocation were atomic, but there is no mechanism enforcing this.

Listing 1.2 shows the same example coded in Erlang. A call to the initial function, `main`, spawns two processes (the *agents*) that start with the execution of function calls `agent(1,Main)` and `agent(2,Main)`, respectively. Here, `Main` is a variable with the *pid* of the main process, which is obtained via a call to the predefined function `self`.

Then, at line 8, the main process calls to function `seats` with argument `3` (the initial number of available seats). From this point on, the main process behaves as a *server* that executes a potentially infinite loop that waits for requests and replies to them. Here, the *state* of the process is given by the argument `Num` which

```
1  seats = 3;
2  begin b0.0
3   var agent1 = 1;
4   var agent2 = 1;
5   proc p0.0 sell is
6    seats = seats - 1;
7   end;
8
9   par {
10   while w0.0 (agent1 == 1) do
11    if i0.0 (seats > 0) then
12     call c0.0 sell;
13    else
14     agent1 = 0;
15    end;
16   end;
17  } {
18   while w1.0 (agent2 == 1) do
19    if i1.0 (seats > 0) then
20     call c1.0 sell;
21    else
22     agent2 = 0;
23    end;
24   end;
25  }
26  remove proc p0.0 sell end;
27  remove var agent2 = 1;
28  remove var agent1 = 1;
29 end
```

Listing 1.1. Airline booking example in a concurrent imperative language. All paths and identifier stacks are omitted as these are inserted automatically.

represents the current number of available seats. The server accepts two kinds of messages: {`numOfSeats,Pid`}, a request to know the current number of available seats, and {`sell,Pid`}, to decrease the number of available seats (analogously to the procedure `sell` in Listing 1.1). In the first case, the number of available seats is sent back to the agent that performed the request (`Pid ! Num`); in the second case, the number of the booked seat is sent.[3] The behaviour of the agents (lines 17–23) is simple. An agent first sends a request to know the number of available seats, `Pid ! {numOfSeats,self()}`, where `self()` is required for the main process to be able to send a reply back to the sender. Then, the agent suspends its execution waiting for an answer {`seats,Num`}: if `Num` is greater than zero, the agent sends a new message to sell a seat (`Pid ! {sell,self()}`) and

[3] We note that the number of the booked seat, `Num`, is not used by function `agent` in our example, but might be used in a more realistic program. We keep this value anyway since it will ease the understanding of the trace in Sect. 4.

```
1  -module(airline).
2  -export([main/0,agent/2]).
3  8
4  main() ->
5      Main = self(),
6      spawn(?MODULE, agent, [1,Main]),
7      spawn(?MODULE, agent, [2,Main]),
8      seats(3).
9
10 seats(Num) ->
11     receive
12         {numOfSeats,Pid} -> Pid ! {seats,Num}, seats(Num);
13         {sell,Pid} -> io:format("Seat sold!~n"),
14                       Pid ! {booked,Num},seats(Num-1)
15     end.
16
17 agent(NAg,Pid) ->
18     Pid ! {numOfSeats,self()},
19     receive
20         {seats,Num} when Num > 0 -> Pid ! {sell,self()},
21                       receive {booked,_} -> agent(NAg,Pid) end;
22         _ -> io:format("Agent~p done!~n",[NAg])
23     end.
```

Listing 1.2. Airline booking example, in Erlang.

receives the confirmation ({`booked,_`});[4] otherwise, it terminates the execution with the message "`AgentN done!`", where `N` is either `1` or `2`.

3 Backtracking in a Concurrent Imperative Language

In this section we describe a state-saving approach to reversibility in the concurrent imperative programming language described in Sect. 2.1. We begin by discussing our approach and its use within the debugging of the airline example (see Sect. 2.3), along with our simulation tool [20,21].

As described in more detail in [21], we have produced a simulator implementing the operational semantics of our approach. This simulator is capable of parsing a program, automatically inserting removal statements, construct identifiers and paths, and simulating both forward and reverse execution. Each execution can be either end-to-end, or step-by-step.

We first execute the forward version of our airline example completely. This execution produces the annotated version in Fig. 2a, where the identifier stack for each statement has been populated capturing an interleaving order that experiences the bug as outlined in Sect. 2.3. The inverted version of the airline example is shown in Fig. 2b, where the overall statement order has been inverted. Note that some annotations are omitted to keep this source code concise (e.g., no paths

[4] Anonymous variables are denoted by an underscore "_".

```
1  seats = 3 [0];
2  begin b0.0
3   var agent1 = 1 [1];
4   var agent2 = 1 [2];
5   proc p0.0 sell is
6    seats = seats - 1 [7,13,22,23];
7   end [3];
8
9   par {
10   while w0.0 (agent1 == 1) do
11    if i0.0 (seats > 0) then
12     call c0.0 sell [6,8,20,24];
13    else
14     agent1 = 0 [31];
15    end [5,9,18,26,30,32];
16   end [4,17,29,33];
17  } {
18   while w1.0 (agent2 == 1) do
19    if i1.0 (seats > 0) then
20     call c1.0 sell [12,14,21,25];
21    else
22     agent2 = 0 [35];
23    end [11,15,19,27,34,36];
24   end [10,16,28,37];
25  }
26  remove proc p0.0 sell end [38];
27  remove var agent2 = 1 [39];
28  remove var agent1 = 1 [40];
29 end
30 //Finishes with seats = -1
```

(a) Annotated program (executed)

```
1  //Expect seats=0, not seats=-1
2  begin b0.0
3   var agent1 = 1 [40];
4   var agent2 = 1 [39];
5   proc p0.0 sell is
6    seats = seats - 1 [7,13,22,23];
7   end [38];
8
9   par {
10   while w0.0 (agent1 == 1) do
11    if i0.0 (seats > 0) then
12     call c0.0 sell [6,8,20,24];
13    else
14     agent1 = 0 [31];
15    end [5,9,18,26,30,32];
16   end [4,17,29,33];
17  } {
18   while w1.0 (agent2 == 1) do
19    if i1.0 (seats > 0) then
20     call c1.0 sell [12,14,21,25];
21    else
22     agent2 = 0 [35];
23    end [11,15,19,27,34,36];
24   end [10,16,28,37];
25  }
26  remove proc p0.0 sell end [3];
27  remove var agent2 = 1 [2];
28  remove var agent1 = 1 [1];
29 end
30 seats = 3 [0];
```

(b) Inverted program (not yet executed)

Fig. 2. Final annotated and inverted versions of the airline example, with paths omitted

are shown). We start the debugging process at the beginning of the execution of the inverted version (line 1 of Fig. 2b). Recall that all expressions or conditions are not evaluated or used during an inverse execution. Using the final program state showing the misbehaviour (produced via the annotated execution with `seats` = `-1`), the simulator begins by opening the block and re-declaring both local variables and the procedure, using identifiers 40–38. From here, the execution continues with the parallel statement. The final iteration of each while loop is reversed (simulating the inversion of the closing of each agent) using identifiers 37–28. Now the penultimate iteration of each while loop must be inverted. The consecutive identifiers 27 and 26 are then used to ensure that each of the conditional statements (lines 11 and 19) are opened, using two true values retrieved from the reversal information saved.

The execution then continues using identifiers 25–20, where each loop almost completes the current iteration, reversing the last time each of them allocated a

```
Simulator: Rev-Ex> display loops
|--Displaying the current while loop environment
-----------------------------------------------------------------
currently executing a parallel statement
Loop Name       | Program
-----------------------------------------------------------------
w0.0            | while w0.0 (agent1Open == 1) do
                |   if i0.2 (numOfSeats > 0) then
                |     call c0.2 sellTicket(b0.0;) [6, 8]
                |   else
                |     agent1Open = 0 (b0.0;) []
                |   fi (b0.0;) [5, 9, 18] <--------
                | elihw (b0.0;) [4, 17]
-----------------------------------------------------------------
-----------------------------------------------------------------
w1.0            | while w1.0 (agent2Open == 1) do
                |   if i1.2 (numOfSeats > 0) then
                |     call c1.2 sellTicket(b0.0;) [12, 14]
                |   else
                |     agent2Open = 0 (b0.0;) []
                |   fi (b0.0;) [11, 15, 19] <--------
                | elihw (b0.0;) [10, 16]
-----------------------------------------------------------------
-----------------------------------------------------------------
```

Fig. 3. Stopping position of the inverse execution (containing paths automatically inserted by the simulator)

seat. This produces the state where `seats` = `1`, and where the next available step is to close either of the inverse conditional statements. Though the identifiers ensure we must start by closing the conditional with identifier `19`, the fact that both can be closed implies that both are open at the same time. This current position within the inverse execution is shown in Fig. 3, where the command 'display loops' outputs all current while loops (agents) with arrows indicating the next statement to be executed. It is clear from our semantics (see [20]) that the closing of an inverted conditional is the reverse of opening its forward version. Since the two conditionals have been opened using consecutive identifiers, one can see that each committed to selling a ticket. Given that the current state has `seats` = `1`, this execution commits to selling two tickets when only one remains. It is therefore clear that this is an atomicity violation, since interleaving of other actions is allowed between the checking for at least one free seat and the allocation of it. We have therefore shown how the simulator implementing our approach to reversibility can be used during the debugging process of an example bug.

4 Causal-Consistent Reversibility in Erlang

In this section we will discuss how to apply causal-consistent reversible debugging to the airline booking example in Sect. 2.3. Our approach to reversible debugging is based on the following principles [29,30]:

- First, we consider a *reduction* semantics for the language (a subset of Core Erlang [5], which is an intermediate step in Erlang compilation). Our semantics includes two transition relations, one for expressions (which is mostly a call-by-value semantics for a functional language) and one for *systems*, i.e., collections of processes, possibly interacting through message passing. An advantage of this modular design is that only the transition relation for systems needs to be modified in order to produce a reversible semantics.
- Then, we instrument the standard semantics in different ways. On the one hand, we instrument it to produce a *log* of the computation; namely, by recording all actions involving the sending and receiving of messages, as well as the spawning of new processes (see [30] for more details). On the other hand, one can instrument the semantics so that the configurations now carry enough information to undo any execution step, i.e., a typical Landauer embedding. Producing then a *backward* semantics that proceeds in the opposite direction is not difficult. Here, the configurations may include both a *log*—to drive forward executions—and a *history*—to drive backward executions.
- It is worthwhile to note that forward computations need not follow exactly the same steps as in the recorded computation (indeed the log does not record the total order of steps). However, it is guaranteed that the admissible computations are *causally equivalent* to the recorded one; namely, they differ only for swaps of concurrent actions. Analogously, backward computations need not be the exact inverse of the considered forward computation, but ensuring that backward steps are *causal-consistent* suffices. This degree of freedom is essential to allow the user to focus on the process and/or actions of interest during debugging, rather than inspecting the complete execution (which is often impractical).
- Finally, we define another layer on top of the reversible semantics in order to *drive* it following a number of *requests* from the user, e.g., rolling back up to the point where a given process was spawned, going forward up to the point where a message is sent, etc. This layer essentially implements a stack of requests that follows the causal dependencies of the reversible semantics.

In the following, we consider the causal-consistent reversible debugger CauDEr [27,28] which follows the principles listed above.

CauDEr first translates the airline example into Core Erlang [5]. Then one can execute the program, either using a built-in scheduler, or using the log of an actual execution [30].

Here, if we compile the program in the standard environment and execute the call `main()`, we get the following output:

```
Seat sold!
Seat sold!
Seat sold!
Seat sold!
Agent1 done!
Agent2 done!
```

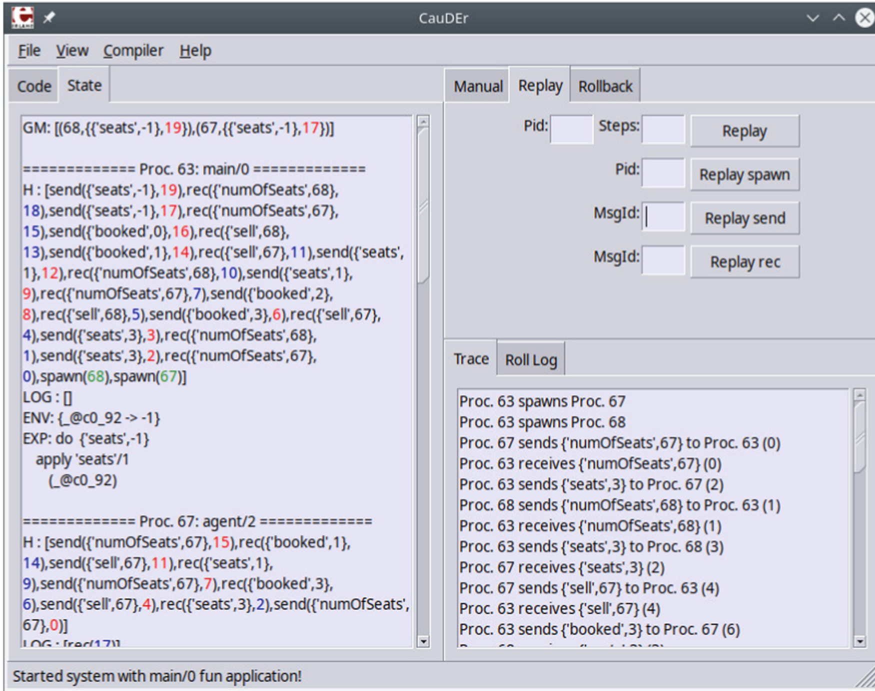


Fig. 4. CauDEr debugging session

which is clearly incorrect since we only had three seats available.

By using the logger and, then, loading both the program and the log into CauDEr (as described in [30]), we can replay the entire execution and explore the sequence of concurrent actions. Figure 4 shows the final state (on the left) and the sequence of concurrent actions (on the right), where process `63` is the main process, and processes `67` and `68` are the agents.

Now, we can look at the sequence of concurrent actions, where messages are labelled with a unique identifier, added by CauDEr, which is shown in brackets to the right of the corresponding line:

```
Proc. 63 spawns Proc. 67
Proc. 63 spawns Proc. 68
Proc. 67 sends {'numOfSeats',67} to Proc. 63 (0)

... 19 lines ...

Proc. 63 receives {'numOfSeats',68} (10)
Proc. 63 sends {'seats',1} to Proc. 68 (12)
Proc. 67 receives {'seats',1} (9)
Proc. 67 sends {'sell',67} to Proc. 63 (11)
Proc. 63 receives {'sell',67} (11)
```

```
Proc. 63 sends {'booked',1} to Proc. 67 (14)
Proc. 68 receives {'seats',1} (12)
Proc. 68 sends {'sell',68} to Proc. 63 (13)
Proc. 63 receives {'sell',68} (13)
Proc. 63 sends {'booked',0} to Proc. 68 (16)
Proc. 67 receives {'booked',1} (14)
Proc. 67 sends {'numOfSeats',67} to Proc. 63 (15)
Proc. 63 receives {'numOfSeats',67} (15)
Proc. 63 sends {'seats',-1} to Proc. 67 (17)
Proc. 68 receives {'booked',0} (16)
Proc. 68 sends {'numOfSeats',68} to Proc. 63 (18)
Proc. 63 receives {'numOfSeats',68} (18)
Proc. 63 sends {'seats',-1} to Proc. 68 (19)
```

One can see that seat number `0` (which does not exist!) has been booked by process `68`, and the notification has been provided via message number `16`.

A good state to explore is the one where message number `16` has been sent. Here a main feature of causal-consistent reversible debugging comes handy: the possibility of going to the state just before a relevant action has been performed, by undoing it, including all and only its consequences. This is called a causal-consistent rollback. CauDEr provides causal-consistent rollbacks for various actions, including send actions. Thus, the programmer can invoke a `Roll send` command with message identifier `16` as a parameter.

In this way, one discovers that the message has been sent by process `63` (as expected, since process `63` is the `main` process). By exploring its state one understands that, from the point of view of process `63`, sending message `16` is correct, since it is the only possible answer to a `sell` message. The bug should be thus before.

From the program code, the programmer knows that whether seat `Num` is available or not is checked by a message of the form {`numOfSeats,Pid`}, which is answered with a message of the form {`seats,Num`}, where `Num` is the number of available seats.

Looking again at the concurrency actions, the programmer can see that process number `68` was indeed notified of the availability of a seat by message number `12`.

We can use again `Roll send`, now with parameter `12`, to check whether this send is correct or not. We discover that indeed the send is correct since, when the message is sent, there is one available seat. However, here, another window comes handy: the `Roll log` window that shows which actions (causally dependent on the one undone) have been undone during a rollback, which shows:

```
Roll send from Proc. 63 of {'booked',1} to Proc. 67 (14)
Roll send from Proc. 67 of {'numOfSeats',67} to Proc. 63 (15)
Roll send from Proc. 63 of {'seats',1} to Proc. 68 (12)
Roll send from Proc. 68 of {'sell',68} to Proc. 63 (13)
```

By checking it the programmer sees that also the interactions between process `67` and process `63` booking seat `1` are undone. Hence the problem is that, in between the check for availability and the booking, another process may interact with `main`, stealing the seat; thus, the error is an atomicity violation.

Of course, given the simplicity of the system, one could have spotted the bug directly by looking at the code or at the full sequence of message exchanges, but the technique above is quite driven by the visible misbehaviour, hence it will better scale to larger systems (e.g., with more seats and agents, or with additional functionalities).

We remark that, while the presentation above concentrates on the debugger and its practical use, this line of research also deeply considered its theoretical underpinning, as briefly summarised at the beginning of the section. Thanks to this, relevant properties have been proved, e.g., that if a misbehaviour occurs in a computation then the same misbehaviour will occur also in each replay [30].

5 Related Work

Reversible computation in general, and reversible debugging in particular, have been deeply explored in the literature.

A line of research considers naturally reversible languages, that is languages where only reversible programs can be written. Such approaches include the imperative languages Janus [49,50], R-CORE [17] and R-WHILE [16], and the object-oriented languages Joule [43] and ROOPL [18]. These approaches require dedicated languages, and cannot be applied to mainstream languages like Erlang or a classic imperative language, as we do in this paper.

The backtracking approach has been applied, e.g., in the Reverse C Compiler (RCC) defined by Perumalla et al. [6,37]. It supports the entire programming language C, but lacks a proof of correctness, which is instead provided by our approaches. The Backstroke framework [48] is a further example, supporting the vast majority of the programming language C++. This framework has been used to provide reverse execution in the field of Parallel Discrete Event Simulation (PDES) [13], as described in more recent works by Schordan et al. [40–42]. Similar approaches have been used for debugging, e.g., based on program instrumentation techniques [7]. Identifiers and keys are used to control execution in the work by Phillips and Ulidowski [38,39]. Another related work is omniscient debugging, where each assignment and method call is stored in an execution history, which can be used to restore any desired program state. An example of such a debugger written for Java was proposed by Lewis [32].

Causal-consistent reversibility has been mainly studied in the area of foundational process calculi such as CCS [10] and its variants [35,38], π-calculus [9], and higher-order π-calculus [26] and coordination languages such as Klaim [15]. The application to debugging has been first proposed in [14] in the context of the toy functional language μOz. A related approach is Actoverse [44], for Akka-based applications. It provides many relevant features complementary to ours, such as a partial-order graphical representation of message exchanges. On the

other side, Actoverse allows one to explore only some states of the computation, such as the ones corresponding to message sending and receiving. We also mention Causeway [45], which however is not a full-fledged debugger, but just a post-mortem traces analyser.

6 Conclusion

We presented two approaches to reversible debugging of concurrent systems, we will now briefly compare them. Beyond the language they consider, the main difference between the two approaches is in the order in which execution steps can be reversed. The backtracking approach undoes them in reverse order of execution. This means that there is no need to track dependencies, and the user of the debugger can easily anticipate which steps will be undone by looking at identifiers. The causal-consistent approach instead allows independent steps of an execution to be reversed in any order, hence tracking dependencies between steps is crucial. This offers the benefit that only the steps strictly needed to reach the desired point of an execution need to be reversed, and steps which happened in between but were actually independent are disregarded.

Debugging is a relevant application area for reversible computation, but reversible debugging for concurrent and distributed systems is still in its infancy. While different techniques have been put forward, they are not yet able to deal with real, complex systems. A first reason is that they do not tackle mainstream languages (Erlang could be considered mainstream, but only part of the language is currently covered). When this first step will be completed, then runtime overhead and size of the logs will become relevant problems, as they are now in the setting of sequential reversible debugging.

References

1. Abramov, S., Glück, R.: Principles of inverse computation and the universal resolving algorithm. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H. (eds.) The Essence of Computation. LNCS, vol. 2566, pp. 269–295. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-36377-7_13
2. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005), pp. 249–258. IEEE Computer Society (2005). https://doi.org/10.1109/LICS.2005.1
3. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T.: Reversible debugging software - quantify the time and cost saved using reversible debuggers (2012). http://www.roguewave.com
4. Cardelli, L., Laneve, C.: Reversible structures. In: Fages, F. (ed.) Proceedings of the 9th International Conference on Computational Methods in Systems Biology (CMSB 2011), pp. 131–140. ACM (2011). https://doi.org/10.1145/2037509.2037529
5. Carlsson, R., et al.: Core Erlang 1.0.3. language specification (2004). https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf

6. Carothers, C.D., Perumalla, K.S., Fujimoto, R.: Efficient optimistic parallel simulations using reverse computation. ACM Trans. Model. Comput. Simul. **9**(3), 224–253 (1999)
7. Chen, S., Fuchs, W.K., Chung, J.: Reversible debugging using program instrumentation. IEEE Trans. Softw. Eng. **27**(8), 715–727 (2001). https://doi.org/10.1109/32.940726
8. Conrod, J.: Tutorial: reverse debugging with GDB 7 (2009). http://jayconrod.com/posts/28/tutorial-reverse-debugging-with-gdb-7
9. Cristescu, I., Krivine, J., Varacca, D.: A compositional semantics for the reversible π-calculus. In: Proceedings of the 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2013), pp. 388–397. IEEE Computer Society (2013). https://doi.org/10.1109/LICS.2013.45
10. Danos, V., Krivine, J.: Reversible communicating systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28644-8_19
11. Engblom, J.: A review of reverse debugging. In: Morawiec, A., Hinderscheit, J. (eds.) Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference (S4D), pp. 28–33. IEEE (2012)
12. Frank, M.P.: Introduction to reversible computing: motivation, progress, and challenges. In: Bagherzadeh, N., Valero, M., Ramírez, A. (eds.) Proceedings of the Second Conference on Computing Frontiers, pp. 385–390. ACM (2005). https://doi.org/10.1145/1062261.1062324
13. Fujimoto, R.: Parallel discrete event simulation. Commun. ACM **33**(10), 30–53 (1990). https://doi.org/10.1145/84537.84545
14. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In: Gnesi, S., Rensink, A. (eds.) FASE 2014. LNCS, vol. 8411, pp. 370–384. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_26
15. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent rollback in a tuple-based language. J. Log. Algebraic Meth. Program. **88**, 99–120 (2017)
16. Glück, R., Yokoyama, T.: A linear-time self-interpreter of a reversible imperative language. Comput. Softw. **33**(3), 108–128 (2016)
17. Glück, R., Yokoyama, T.: A minimalist's reversible while language. IEICE Trans. **100-D**(5), 1026–1034 (2017)
18. Haulund, T.: Design and implementation of a reversible object-oriented programming language. Master's thesis, Faculty of Science, University of Copenhagen (2017). https://arxiv.org/abs/1707.07845
19. Hoey, J.: Reversing an imperative concurrent programming language. Ph.D. thesis, University of Leicester (2020)
20. Hoey, J., Ulidowski, I., Yuen, S.: Reversing imperative parallel programs with blocks and procedures. In: 2018 Proceedings of Express/SOS (2018)
21. Hoey, J., Ulidowski, I.: Reversible imperative parallel programs and debugging. In: Thomsen, M.K., Soeken, M. (eds.) RC 2019. LNCS, vol. 11497, pp. 108–127. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21500-2_7
22. Hoey, J., Ulidowski, I., Yuen, S.: Reversing parallel programs with blocks and procedures. In: Pérez, J.A., Tini, S. (eds.) Proceedings of the Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics (EXPRESS/SOS 2018), EPTCS, vol. 276, pp. 69–86 (2018). https://doi.org/10.4204/EPTCS.276.7
23. European COST actions IC1405 on "reversible computation - extending horizons of computing". http://www.revcomp.eu/

24. Landauer, R.: Irreversibility and heat generated in the computing process. IBM J. Res. Dev. **5**, 183–191 (1961)
25. Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent reversibility. Bull. EATCS **114**, 121–139 (2014)
26. Lanese, I., Mezzina, C.A., Stefani, J.B.: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. **625**, 25–84 (2016)
27. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr. https://github.com/mistupv/cauder
28. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: CauDEr: a causal-consistent reversible debugger for Erlang. In: Gallagher, J.P., Sulzmann, M. (eds.) FLOPS 2018. LNCS, vol. 10818, pp. 247–263. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-90686-7_16
29. Lanese, I., Nishida, N., Palacios, A., Vidal, G.: A theory of reversibility for Erlang. J. Log. Algebraic Meth. Program. **100**, 71–97 (2018)
30. Lanese, I., Palacios, A., Vidal, G.: Causal-consistent replay debugging for message passing programs. In: Pérez, J.A., Yoshida, N. (eds.) FORTE 2019. LNCS, vol. 11535, pp. 167–184. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21759-4_10
31. Laursen, J.S., Schultz, U.P., Ellekilde, L.: Automatic error recovery in robot assembly operations using reverse execution. In: Proceedings of the 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2015), pp. 1785–1792. IEEE (2015). https://doi.org/10.1109/IROS.2015.7353609
32. Lewis, B.: Debugging backwards in time. In: Ronsse, M., Bosschere, K.D. (eds.) Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), pp. 225–235 (2003). https://arxiv.org/abs/cs/0310016
33. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Eggers, S.J., Larus, J.R. (eds.) Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), pp. 329–339. ACM (2008). https://doi.org/10.1145/1346281.1346323
34. McNellis, J., Mola, J., Sykes, K.: Time travel debugging: root causing bugs in commercial scale software. CppCon talk (2017). https://www.youtube.com/watch?v=l1YJTg_A914
35. Mezzina, C.A.: On reversibility and broadcast. In: Kari, J., Ulidowski, I. (eds.) RC 2018. LNCS, vol. 11106, pp. 67–83. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99498-7_5
36. Nishida, N., Palacios, A., Vidal, G.: A reversible semantics for Erlang. In: Hermenegildo, M.V., Lopez-Garcia, P. (eds.) LOPSTR 2016. LNCS, vol. 10184, pp. 259–274. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63139-4_15
37. Perumalla, K.: Introduction to Reversible Computing. CRC Press, Boca Raton (2014)
38. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebraic Program. **73**(1–2), 70–96 (2007)
39. Phillips, I., Ulidowski, I., Yuen, S.: A reversible process calculus and the modelling of the ERK signalling pathway. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 218–232. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36315-3_18
40. Schordan, M., Jefferson, D., Barnes, P., Oppelstrup, T., Quinlan, D.: Reverse code generation for parallel discrete event simulation. In: Krivine, J., Stefani, J.-B. (eds.) RC 2015. LNCS, vol. 9138, pp. 95–110. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20860-2_6

41. Schordan, M., Oppelstrup, T., Jefferson, D.R., Barnes Jr., P.D.: Generation of reversible C++ code for optimistic parallel discrete event simulation. New Gener. Comput. **36**(3), 257–280 (2018). https://doi.org/10.1007/s00354-018-0038-2
42. Schordan, M., Oppelstrup, T., Jefferson, D.R., Barnes Jr, P.D., Quinlan, D.J.: Automatic generation of reversible C++ code and its performance in a scalable kinetic Monte-Carlo application. In: Fujimoto, R., Unger, B.W., Carothers, C.D. (eds.) Proceedings of the 2016 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS 2016), pp. 111–122. ACM (2016). https://doi.org/10.1145/2901378.2901394
43. Schultz, U.P., Axelsen, H.B.: Elements of a reversible object-oriented language. In: Devitt, S., Lanese, I. (eds.) RC 2016. LNCS, vol. 9720, pp. 153–159. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40578-0_10
44. Shibanai, K., Watanabe, T.: Actoverse: a reversible debugger for actors. In: Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2017), pp. 50–57. ACM (2017). https://doi.org/10.1145/3141834.3141840
45. Stanley, T., Close, T., Miller, M.S.: Causeway: a message-oriented distributed debugger. Technical report, HP Labs tech report HPL-2009-78 (2009). http://www.hpl.hp.com/techreports/2009/HPL-2009-78.html
46. Sutter, H.: The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobb's J. **30**(3), 202–210 (2005)
47. Undo Software: Increasing software development productivity with reversible debugging (2014). http://undo-software.com/wp-content/uploads/2014/10/Increasing-software-development-productivity-with-reversible-debugging.pdf
48. Vulov, G., Hou, C., Vuduc, R.W., Fujimoto, R., Quinlan, D.J., Jefferson, D.R.: The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In: Jain, S., Creasey Jr, R.R.., Himmelspach, J., White, K.P., Fu, M.C. (eds.) Proceedings of the Winter Simulation Conference (WSC 2011), pp. 2965–2979. IEEE (2011). https://doi.org/10.1109/WSC.2011.6147998
49. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Ramalingam, G., Visser, E. (eds.) Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2007), pp. 144–153. ACM (2007)
50. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Ramírez, A., Bilardi, G., Gschwind, M. (eds.) Proceedings of the 5th Conference on Computing Frontiers, pp. 43–54. ACM (2008). https://doi.org/10.1145/1366230.1366239
51. Yokoyama, T., Axelsen, H.B., Glück, R.: Fundamentals of reversible flowchart languages. Theor. Comput. Sci. **611**, 87–115 (2016). https://doi.org/10.1016/j.tcs.2015.07.046

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.