Robert Glück
Tetsuo Yokoyama (Eds.)

LNCS 7581

Reversible Computation

4th International Workshop, RC 2012
Copenhagen, Denmark, July 2012
Revised Papers

Springer

Robert Glück Tetsuo Yokoyama (Eds.)

Reversible Computation

4th International Workshop, RC 2012
Copenhagen, Denmark, July 2-3, 2012
Revised Papers

Springer

Volume Editors

Robert Glück
University of Copenhagen
DIKU, Department of Computer Science
DK-2100 Copenhagen, Denmark
E-mail: glueck@acm.org

Tetsuo Yokoyama
Nanzan University
Department of Software Engineering
489-0863 Seto, Japan
E-mail: tyokoyama@acm.org

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-36314-6 e-ISBN 978-3-642-36315-3
DOI 10.1007/978-3-642-36315-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012956181

CR Subject Classification (1998): B.6.1, F.3.1-2, F.1.1-2, J.2, F.2.2, D.2.11-13, D.4.7

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Reversible computation has emerged as a very promising research area in recent years. Applications include low-power design, decoding, program debugging, testing, database recovery, discrete event simulation, reversible algorithms, reversible specification formalisms, reversible programming languages, process algebras and the modeling of biochemical systems. The first reversible circuits have recently been implemented and are seen as promising alternatives to conventional technologies.

The International Workshop on Reversible Computation series provides a platform for the presentation and discussion of new trends and recent work within the area of reversible computation. We are proud to present a comprehensive collection of revised and selected papers covering various aspects of reversible computation originally accepted and presented at the fourth workshop held in Copenhagen, Denmark, during July 2–3, 2012 (RC 2012). The papers in this volume cover theoretical considerations, reversible software and reversible hardware, as well as physical realizations and applications in quantum computing. In addition, the workshop program included an invited talk by Eric Lutz entitled "The Physics of Information: from Maxwell's Demon to Landauer."

The call for papers attracted 46 submissions by 101 authors from 18 countries, the largest number since the first workshop was held at the University of York, UK, in 2009. All submissions were reviewed by at least three experts and selected by a Program Committee. These proceedings include 16 full papers, three work-in-progress reports, and a tutorial paper. We would like to thank all authors for their submissions: their research is the justification for this volume.

Furthermore, this event would not have been possible without the dedication of many people who worked hard to make it successful and enjoyable. In particular, we would like to thank the members of the Program Committee and the external reviewers for their time and expertise in evaluating the submissions and providing extensive feedback to the authors. Special thanks go to Holger Bock Axelsen, Dina Riis Egholm, Susan Ibsen, Inge Hviid Jensen, Sarah Nøhr, Martin Nyborg Thomsen, and Michael Kirkedal Thomsen from the University of Copenhagen for their invaluable support prior to and during the workshop. We would also like to thank Robert Wille and Lisa Jungmann from the University of Bremen for their efforts in organizing the event.

Selected papers from previous workshops were published in *Electronic Notes in Theoretical Computer Science* Volume 253, Issue 6, in the *Journal of Multiple-Valued Logic and Soft Computing* Volume 18, Number 1, and in *Lecture Notes in Computer Science* Volume 7165.

Finally, we would like to thank all authors, participants, and organizers for making the fourth workshop in Copenhagen both successful and enjoyable.

September 2012

Robert Glück
Tetsuo Yokoyama

Organization

Program Committee Chairs

Robert Glück	University of Copenhagen, Denmark
Tetsuo Yokoyama	Nanzan University, Japan

Program Committee

Stéphane Burignat	Ghent University, Belgium
Vincent Danos	Université Paris Diderot, France
Gerhard W. Dueck	University of New Brunswick, Canada
Nate Foster	Cornell University, USA
Luca Gammaitoni	University of Perugia, Italy
Simon Gay	University of Glasgow, UK
Markus Grassl	Centre for Quantum Technologies, Singapore
Jarkko J. Kari	University of Turku, Finland
Martin Kutrib	University of Giessen, Germany
Per Larsson-Edefors	Chalmers University of Technology, Sweden
Kazutaka Matsuda	University of Tokyo, Japan
D. Michael Miller	University of Victoria, Canada
Shin-ichi Minato	Hokkaido University, Japan
Kenichi Morita	Hiroshima University, Japan
Ilia Polian	University of Passau, Germany
Michel Schellekens	University College Cork, Ireland
Irek Ulidowski	University of Leicester, UK
Janis Voigtländer	University of Bonn, Germany
Robert Wille	University of Bremen, Germany
Paolo Zuliani	Carnegie Mellon University, USA

Organizing Committee

Holger Bock Axelsen	University of Copenhagen, Denmark
Dina Riis Egholm	University of Copenhagen, Denmark
Lisa Jungmann	University of Bremen, Germany
Robert Wille	University of Bremen, Germany

Additional Referees

Sponsors

RC 2012 was sponsored by the *MicroPower* research project funded by The Danish Council for Strategic Research. Further, we gratefully acknowledge the local support provided by the Department of Computer Science (DIKU), University of Copenhagen.

Table of Contents

Theoretical Considerations

Tutorial: Graphical Calculus for Quantum Circuits 1
Bob Coecke and Ross Duncan

One-Way Reversible Multi-head Finite Automata 14
Martin Kutrib and Andreas Malcher

A Deterministic Two-Way Multi-head Finite Automaton Can Be Converted into a Reversible One with the Same Number of Heads 29
Kenichi Morita

Undecidability of the Surjectivity of the Subshift Associated to a Turing Machine 44
Rodrigo Torres, Nicolas Ollinger, and Anahí Gajardo

Reversible Software and Languages

Isomorphic Interpreters from Logically Reversible Abstract Machines ... 57
Roshan P. James and Amr Sabry

Synthesizing Loops for Program Inversion 72
Cong Hou, Daniel Quinlan, David Jefferson, Richard Fujimoto, and Richard Vuduc

Frugal Encoding in Reversible $\mathcal{MOQA}$: A Case Study for Quicksort 85
Diarmuid Early, Ang Gao, and Michel Schellekens

Towards a General-Purpose, Reversible Language for Controlling Self-reconfigurable Robots 97
Ulrik Pagh Schultz

Reversible and Quantum Circuits

Reversible and Quantum Circuit Optimization: A Functional Approach 112
Zahra Sasanian and D. Michael Miller

Properties of Quantum Templates 125
Md. Mazder Rahman and Gerhard W. Dueck

Optimal 4-bit Reversible Mixed-Polarity Toffoli Circuits 138
Marek Szyprowski and Paweł Kerntopf

Design of an Online Testable Ternary Circuit from the Truth Table 152
Noor M. Nayeem and Jacqueline E. Rice

Physical Realizations and Design

Garbageless Reversible Implementation of Integer Linear Transformations 160
Stéphane Burignat, Kenneth Vermeirsch, Alexis De Vos, and Michael Kirkedal Thomsen

Garbage-Free Reversible Integer Multiplication with Constants of the Form $2^k \pm 2^l \pm 1$ 171
Holger Bock Axelsen and Michael Kirkedal Thomsen

Property Checking of Quantum Circuits Using Quantum Multiple-Valued Decision Diagrams 183
Julia Seiter, Mathias Soeken, Robert Wille, and Rolf Drechsler

Using πDDs in the Design of Reversible Circuits (Work-In-Progress) 197
Mathias Soeken, Robert Wille, Shin-ichi Minato, and Rolf Drechsler

Distributed Systems

A Verification Technique for Reversible Process Algebra 204
Jean Krivine

A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway 218
Iain Phillips, Irek Ulidowski, and Shoji Yuen

Controlled Reversibility and Compensations 233
Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani

Author Index 241

Tutorial: Graphical Calculus for Quantum Circuits

Bob Coecke and Ross Duncan

Oxford University and Université Libre de Bruxelles
{coecke,rwd}@comlab.ox.ac.uk

Abstract. We explain the graphical ZX-calculus for reasoning about qubits without any reference to the underlying categorical semantics, and illustrate its use on quantum circuits.

The ZX-calculus is a graphical language for describing quantum systems. It was introduced in [2] and has been used by several authors in a variety of application areas e.g. for reasoning about a variety of quantum computational models including measurement-based quantum computation [3,7,8,11], quantum cryptography [10,12], and quantum-non-locality [5,4].

The ZX-calculus is an equational theory, based on rewriting the diagrams which comprise its syntax. Re-writing can be automated by means of the `quantomatic` software[1], developed (and still being improved) by L. Dixon, R. Duncan, B. Frot, A. Kissinger, A. Merry, D. Quick and M. Soloviev. Each diagram has an interpretation as a linear map, connecting the graphical language to the usual Hilbert space formalism of quantum mechanics. Of course, this interpretation is sound with respect to the equational rules.

The ZX-calculus describes qubit behaviors $\mathcal{Q} := \mathbb{C}^2$ in terms of a graphical representation of the Z- and X-observables, with respective bases of eigenvectors:

$$\{|0\rangle, |1\rangle\} \quad \text{and} \quad \left\{|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\ ,\ |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right\}\ .$$

A graphical representation of the phases relative to these observables results in a language that is universal for representing arbitrary linear maps between tensor powers of qubits, as we show in Section 4.

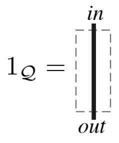
Recently, Miriam Backens showed that the ZX-calculus, augmented with the Euler angle decomposition of the Hadamard gate, is complete for pure stabilizer quantum mechanics [1]. This is the area of quantum mechanics concerned with stabilizer states [9] and transformations between these, where stabilizer states are those states that are stabilized by a subgroup of the Pauli group, which is the closure of the Pauli operations under composition. For example, the qubit has 6 stabilizer states, namely:

$$|0\rangle, |1\rangle, |+\rangle, |-\rangle, \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle) \text{ and } \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle)\,.$$

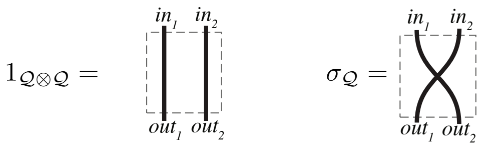
[1] `http://sites.google.com/site/quantomatic/`

1 The Graphical Language: Networks of Wires and Dots

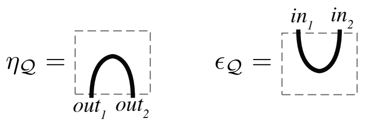
The ZX-calculus consists of components joined by wires, similar to electronic circuit diagrams or flow charts. The simplest non-trivial diagram in the language is simply a wire running from top to bottom:



We think of diagrams as being enclosed in a box with a certain number of points through which wires enter and leave; that is, each diagram has a fixed *interface*. Exactly one wire must be present at each point of the interface, and we must distinguish which wire is connected to which point. Indeed, this is the only difference between the two diagrams below.



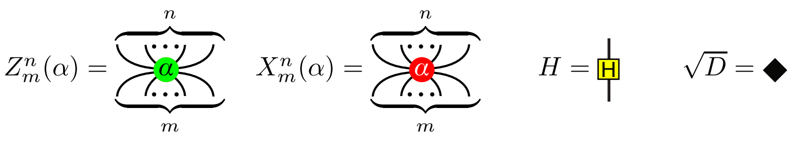
It is not important whether crossing wires pass over or under. Wires may bend, linking two outputs to form a cap, or two inputs to form a cup.



From here on, the inputs and outputs will not be named, and are distinguished simply by their ordering from left to right. We write $\mathfrak{D} : m \to n$ to indicate that the diagram $\mathfrak{D}$ has m inputs and n outputs.

Aside from wires, the ZX-calculus contains four kinds of component:

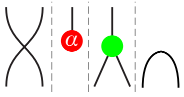
- Z vertices (green dots), labelled by an angle $\alpha \in [0, 2\pi)$, called the *phase*. These can have any number of inputs or outputs (including none).
- X vertices (red dots), labelled by a phase. These too can have any number of inputs or outputs (including none).
- H vertices (yellow squares). These have one input and one output.
- $\sqrt{D}$ vertices (black diamonds). These have no inputs nor outputs.



We refer to the Z and X vertices as "spiders", and make the convention that if $\alpha = 0$ the angle is omitted.

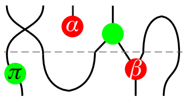
Diagrams are built from these generators—straight, crossing, and bending wires, and Z, X, H, and $\sqrt{D}$ vertices—in two manners.

– Placing them side-by-side:



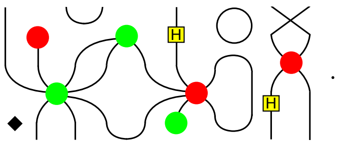
Notation: Given $\mathfrak{D} : m \to n$ and $\mathfrak{D}' : m' \to n'$ their *tensor product* is denoted $\mathfrak{D} \otimes \mathfrak{D}' : m + m' \to n + n'$.

– Connecting outputs to inputs:



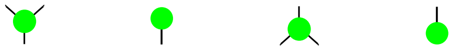
Notation: Given $\mathfrak{D}_1 : m \to n$ and $\mathfrak{D}_2 : n \to k$ their *composition* is denoted $\mathfrak{D}_2 \circ \mathfrak{D}_1 : m \to k$.

Therefore the terms of graphical language are networks of vertices of each type, straight, crossing, and bent wires:



In such a network, there can be no “loose wires”: every wire must terminate at a vertex, or else be an input or output.

Important examples are those spiders with 2 inputs and 1 output (cf. a binary operation), with no input and 1 output (cf. initiation of a value) which we will call a *point*, with 1 input and 2 outputs (cf. copying) and with 1 input and no output (cf. erasing):



As we will see shortly, these unlabelled spiders play a special role in the calculus, as do those labelled by π.

2 The Equational Rules

In addition to the rules for constructing diagrams, the calculus consists of a set of equations which specify how one diagram may be transformed into another. These rules are presented in Figure 1—it is not known whether ZX-calculus equality is decidable. We now expand on these rules and give some examples of their use.

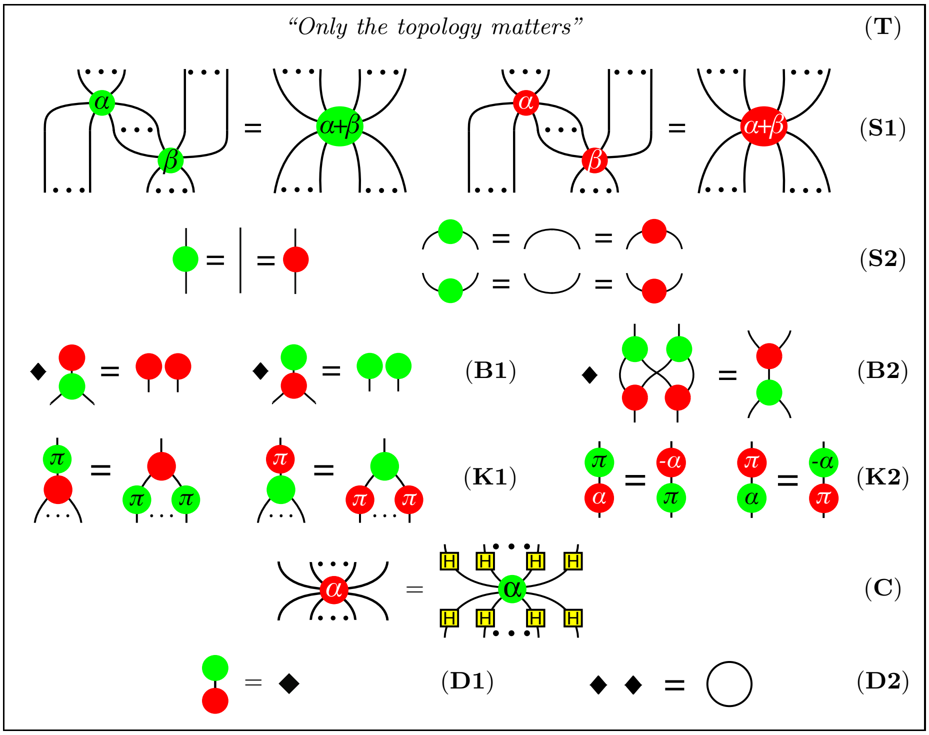
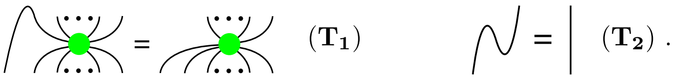


Fig. 1. Rules for the ZX-calculus

2.1 The T-Rule

The informally stated **T**-rule can be made more precise, but for practical purposes, the intuitive reading of "only the topology matters" suffices: the wires of the diagram may be arbitrarily stretched, bent, twisted, tied in knots, etc., without altering the meaning of the diagram, provided the connections are maintained. More precisely, after identifying (e.g. by enumerating) the inputs and the outputs, any topological deformation of the internal structure of the network yields a network that is equal to the given one.

Two important examples of such 'homotopic rewrites' are:



In fact, these two rules can also be seen as consequences of the **S**-rules, when introducing a green dot on the caps and cups as in (**S2**); see Example 2 below.

Remark 1. Since wires can be stretched without consequence, adding a straight length of wire to the input or output of diagram has strictly no effect. Hence bundles of straight wires act as *identity elements* in the algebra of diagrams.

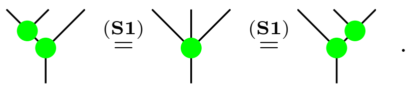
Remark 2. While the slogan says “only the topology matters”, this does not imply that the topology is always preserved. The other rules may change the topology of the diagram in various ways, for example to remove loops, or to disconnect previously connected vertices.

2.2 The S-Rules

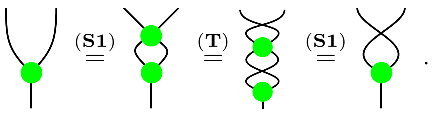
The “spider” rules govern how dots of the same colour interact. Rule (**S1**) states that connected dots of the same color can be merged, summing the phases; conversely, a dot can be ‘decomposed’ along one or more connecting wires. Notice that the number of connecting wires is irrelevant.

The equations (**S2**) specify when spiders are trivial: dots of degree 2 with phase $\alpha = 0$ can be removed, or conversely, introduced.

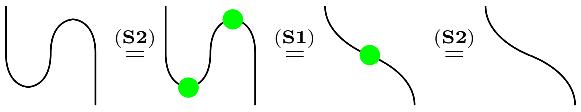
Example 1. If we view the dot $Z_1^2 : 2 \to 1$ as a binary operation, (**S1**) tells us that it is associative:



Less obviously, (**S1**) implies that this operation is commutative:



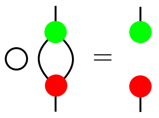
Example 2. The ($\mathbf{T_2}$) rule can be derived using the **S**-rules:



2.3 The B-Rules

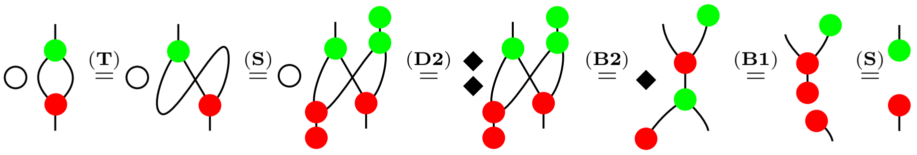
The **B1**-rule can be read loosely as “green copies red points” and “red copies green points”, in both cases “up to a diamond”. The **B2**-rule is a powerful commutation principle, and generates a whole family of equations, allowing alternating cycles of red and green dots to be replaced with simpler graphs.

Example 3. An important equation derivable from the **B**-rules is the following:

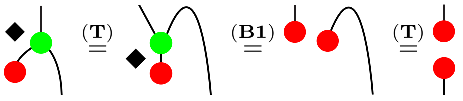


($\mathbf{B'}$)

This equation is obtained as follows:



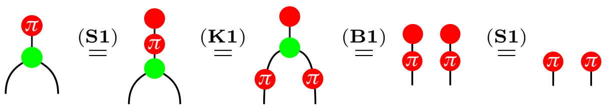
Note that the step labelled (**B1**) in fact applies a version of that rule deformed by (**T**), without altering the topology. We could do this more explicitly using the $\mathbf{T}_1$ and $\mathbf{T}_2$ examples as follows:



2.4 The K-Rules

These rules are concerned with special properties of spiders with phase $\alpha = \pi$. Rule (**K1**) states that dots labelled by π commute with spiders of the other colour; $X_1^1(\pi)$ is a homomorphism of the comultiplication $Z_2^1(0)$, and vice versa.

Example 4. Thanks to rule (**K1**), points with phase π can also be copied just like points with phase zero:



Since the points labelled with π or 0 can be copied we call these *classical points*; then $Z_1^1(\pi)$ and $X_1^1(\pi)$ are called *classical maps*. (Of course, **K** stands for "*k*lassical".) In the next section, we will see that $Z_1^1(\pi)$ and $X_1^1(\pi)$ are interpreted by the familiar Z and X gates respectively.

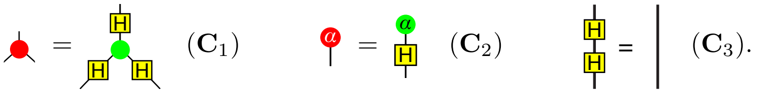
Rule (**K2**) states that π-labelled dots invert phases of dots of the other colour.

Example 5. By rules (**S1**) and (**S2**), the degree 2 spiders $Z_1^1(\alpha)$ form an abelian group, and by (**K2**), conjugation by $X_1^1(\pi)$— note here that $X_1^1(\pi)$ is self-inverse since $\pi + \pi = 0$ —sends each element to its inverse.

2.5 The C-Rule

This rule allows the H vertex to function as an explicit colour changing operation which transforms "green structures" into "red structures" and vice versa. In the next section, we will see that the H vertex is interpreted by the familiar Hadamard gate, exchanging the X and Z bases.

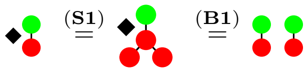
Example 6. Some special cases of this rule are:



2.6 The D-Rules

The (**D2**) rule states that two black diamonds are equal to a loop of wire, itself the result of composing a cup and a cap. We will see in the next section that the loop represents the dimension of the underlying Hilbert space, and spacial juxtaposition is a form of multiplication, justifying the name $\sqrt{D}$.

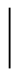
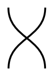
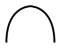
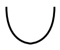
The (**D1**) rule 'almost follows' from the other rules:

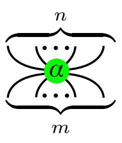


which would yield the desired result if could be cancelled.

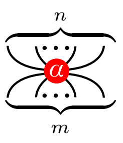
3 Interpreting the zx-calculus in Hilbert Space

Given a diagram $\mathfrak{D}$ with n inputs and m outputs, we construct a corresponding linear map $D : \mathcal{Q}^n \to \mathcal{Q}^m$ as follows. If $\mathfrak{D}$ consists of just a single generator—that is, one of $1_\mathcal{Q}$, $\sigma_\mathcal{Q}$, $\eta_\mathcal{Q}$, $\epsilon_\mathcal{Q}$, $Z^n_m(\alpha)$, $X^n_m(\alpha)$, H, or $\sqrt{D}$[2]—then its corresponding linear map is as shown below:

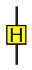
 $= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$  $= \begin{pmatrix} 1&0&0&0 \\ 0&0&1&0 \\ 0&1&0&0 \\ 0&0&0&1 \end{pmatrix}$  $= |00\rangle + |11\rangle$  $= \langle 00| + \langle 11|$



$$:: \begin{cases} \overbrace{|0 \ldots 0\rangle}^{n} \mapsto \overbrace{|0 \ldots 0\rangle}^{m} \\ |1 \ldots 1\rangle \mapsto e^{i\alpha} |1 \ldots 1\rangle \\ \text{others} \mapsto 0 \end{cases}$$



$$:: \begin{cases} \overbrace{|+ \ldots +\rangle}^{n} \mapsto \overbrace{|+ \ldots +\rangle}^{m} \\ |- \ldots -\rangle \mapsto e^{i\alpha} |- \ldots -\rangle \\ \text{others} \mapsto 0 \end{cases}$$

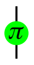


$= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ $= \sqrt{2}$

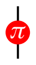
If $\mathfrak{D}$ consists of several generators there are two cases:

– if $\mathfrak{D} = \mathfrak{D}_1 \otimes \mathfrak{D}_2$ then $D = D_1 \otimes D_2$;
– if $\mathfrak{D} = \mathfrak{D}_1 \circ \mathfrak{D}_2$, then $D = D_1 \circ D_2$.

Example 7. The generators $Z^1_1(\pi)$ and $X^1_1(\pi)$ are the Pauli Z and X matrices:

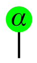


$= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$

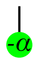


$= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

The generator $Z^0_1(\alpha)$ represents the preparation of a fresh qubit in the state $|+_\alpha\rangle = |0\rangle + e^{i\alpha} |1\rangle$, while $Z^0_1(-\alpha)$ is the projection onto that state.



$= \begin{pmatrix} 1 \\ e^{i\alpha} \end{pmatrix}$



$= \begin{pmatrix} 1 & e^{-i\alpha} \end{pmatrix}$.

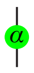
[2] Recall these have been introduced in Section 1.

The order in which we divide the diagram into pieces does not matter to the final result, so long as the "cuts" do not pass through any vertices, nor any points where wires cross, nor any points of inflection of a wire.

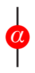
Proposition 1 (Soundness). *If diagrams $\mathfrak{D}_1$ and $\mathfrak{D}_2$ are equal according to the equational rules of the* ZX*-calculus then $D_1 = D_2$.*

4 Universality of the zx-calculus

We claim that we now have enough expressive power to write down any arbitrary linear map from n qubits to m qubits. The green and red phases, respectively:

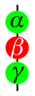


$$= Z_1^1(\alpha) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{pmatrix}$$



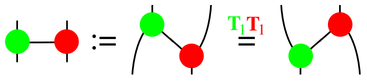
$$= X_1^1(\alpha) = e^{-i\alpha/2} \begin{pmatrix} \cos\frac{\alpha}{2} & i\sin\frac{\alpha}{2} \\ i\sin\frac{\alpha}{2} & \cos\frac{\alpha}{2} \end{pmatrix}$$

correspond with rotations of angle α respectively around the Z- and X-axis. Combining both the 'green' and the 'red' phases allows us to write down any arbitrary one-qubit unitary in terms of its Euler-angle decomposition:



$$= Z_1^1(\gamma) \circ X_1^1(\beta) \circ Z_1^1(\alpha) \tag{1}$$

The controlled-NOT gate is defined by



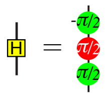
$$= \begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \\ 0\,0\,0\,1 \\ 0\,0\,1\,0 \end{pmatrix} = \wedge\mathsf{X}\,. \tag{2}$$

Standard results in quantum computing state that $\wedge\mathsf{X}$ gates and arbitrary one-qubit unitaries suffice to construct any n-qubit unitary map. As equations 1 and 2 show, the ZX-calculus contains this universal gate set, and hence:

Proposition 2. *Let $A : \mathcal{Q}^n \to \mathcal{Q}^m$ be a unitary map*[3]*; then there exists a diagram $\mathfrak{A}$ in the* ZX*-calculus whose Hilbert space interpretation is A.*

5 Completeness of the zx-calculus

If we augment the ZX-calculus with the explicit decomposition of the Hadamard gate in its Euler angles:



$$\tag{3}$$

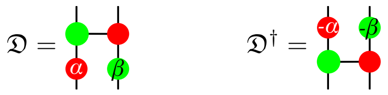
then, given two diagrams that have the same interpretation in Hilbert space within the stabilizer fragment of quantum theory—see introduction—, it is possible to show that these diagrams are equal using the ZX-calculus [1].

Theorem 1 (Backens, 2012). *The* ZX*-calculus augmented with equation 3 is complete for stabilizer quantum mechanics.*

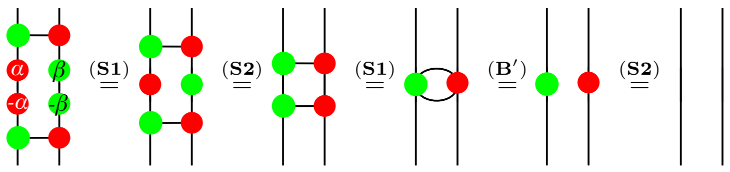
[3] In fact, by exploiting *map-state* duality any linear map may be represented; see [3].

6 The zx-calculus in Use

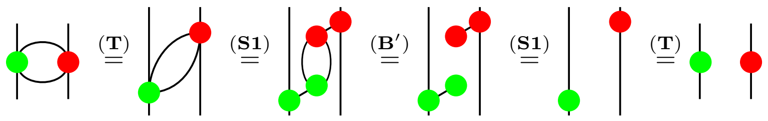
Example 8 (Adjoints). Given the diagram $\mathfrak{D}$, we form its adjoint $\mathfrak{D}^\dagger$ by flipping the diagram vertically and negating the labelling angles, as shown:



We claim that $\mathfrak{D}$ is unitary. Half of the required proof is shown below.

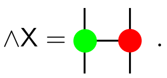


The 'horizontal application' of the $\mathbf{B}'$-rule can be decomposed as follows:

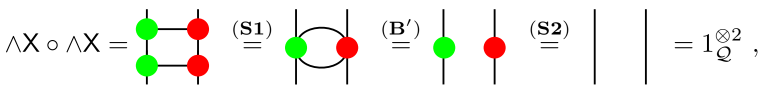


from which it follows that pairs of wires between green and red dots can be eliminated. It remains to show that $\mathfrak{D} \circ \mathfrak{D}^\dagger = 1_{\mathcal{Q}^2}$.

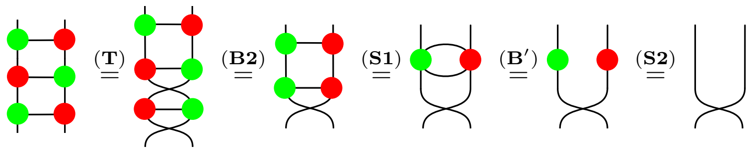
Example 9 (The $\wedge\mathsf{X}$ *gate).* We have already seen the controlled-NOT gate:



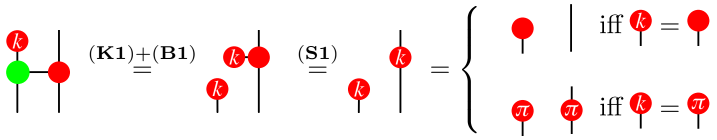
It is manifestly self-adjoint. We can prove that it is also unitary:



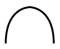
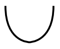
An elementary exercise is to show that a sequence of three $\wedge\mathsf{X}$ gates can be used to swap to qubits. A graphical proof of this fact is given below.



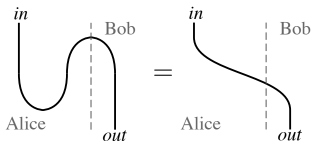
We can describe $\wedge\mathsf{X}$ by the following "behavioural specification": *when the control input is* $|0\rangle$*, the target qubit is left unchanged; when the control qubit is* $|1\rangle$*, the target qubit is flipped*. Letting represent one of the two red classical points, that is, either $= |0\rangle$ or $= |1\rangle$, we can supply a qubit to the control input (the left input, connected to the green dot), and obtain the following proof:



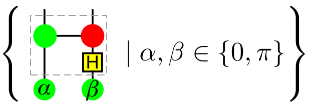
Example 10 (The teleportation protocol). The teleportation protocol consists of two main components: the preparation of the Bell state, and the Bell basis measurement. As described in Section 3, the (unnormalised) Bell state is represented by a cap, and its corresponding projection by a cup:

$$|00\rangle + |11\rangle =$$  $,\qquad \langle 00| + \langle 11| =$  .

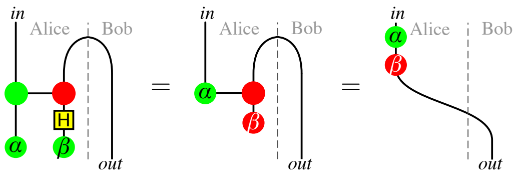
Combining these two elements, we obtain an almost trivial proof of the correctness of teleportation, in the case where Alice observes $|00\rangle + |11\rangle$.



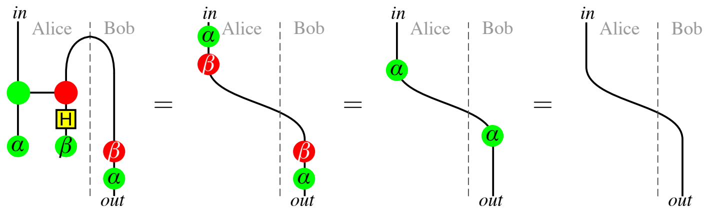
The role of classical communication is hidden in this picture, but it is revealed by a more detailed look at the Bell basis measurement. Let $\alpha, \beta \in \{0, \pi\}$. Ranging over the 4 possible (α, β) pairs in the diagram below gives the 4 possible outcomes of a Bell basis measurement:

$$\{\ \langle\mathbf{\Psi}_+|, \langle\mathbf{\Psi}_-|, \langle\mathbf{\Phi}_+|, \langle\mathbf{\Phi}_-|\ \} =$$ 

(Notice that the boxed part of the diagram is simply the circuit which rotates the Bell basis onto the X-basis.) This description of the protocol displays the Pauli errors that are introduced if Alice observes the other possible outcomes.

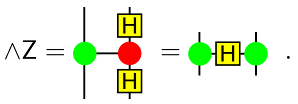


From this we can derive a complete description of the protocol, and show, including Bob's corrections, which are classically correlated to Alice's observations.

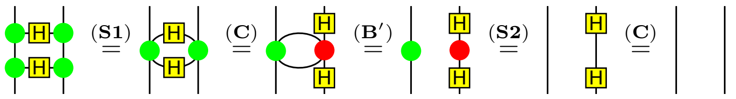


The first equation is the preceding derivation collapsed into one step, while the last two equations use the spider rules and the fact that $2\alpha = 2\beta = 0$.

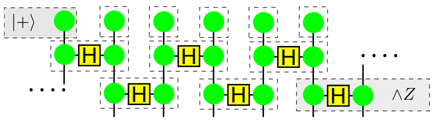
Example 11 (The $\wedge\mathsf{Z}$ gate). Since $\mathsf{Z} = H\mathsf{X}H$ we can obtain the $\wedge\mathsf{Z}$ gate from the $\wedge\mathsf{X}$ gate by conjugating the target qubit with H gates, as shown below:



We can immediately read off two properties of this gate from its graphical representation: it is self-adjoint, and it is symmetric in its inputs. It is also unitary:



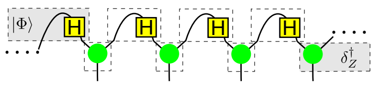
Example 12 (Cluster states). Cluster states, which are used in measurement-based quantum computing, can be prepared in several ways and the ZX-calculus provides short proofs of their equivalence. For example, the original scheme describes a $\wedge\mathsf{Z}$ interaction between qubits initially prepared in the state $|+\rangle$; in our notation this is Z_1^0, or ● . Hence a one-dimensional cluster state can be presented diagrammatically as:



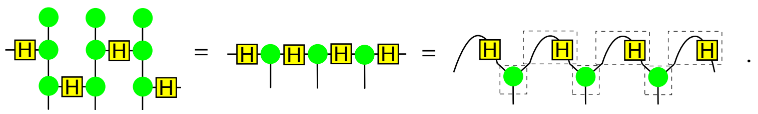
where the boxes delineate the individual $|+\rangle$ preparations and $\wedge\mathsf{Z}$ operations. Alternatively, the cluster state can be prepared by applying a Hadamard gate to one part of a Bell pair to obtain states of the form $|\Phi\rangle = |0+\rangle + |1-\rangle$, and then "fusing" these entangled pairs. The required fusion operation is exactly

$$: \mathcal{Q} \otimes \mathcal{Q} \to \mathcal{Q} :: \begin{cases} |00\rangle & \mapsto |0\rangle \\ |11\rangle & \mapsto |1\rangle \\ |01\rangle, |10\rangle & \mapsto 0 \end{cases}, \tag{4}$$

and a 1D cluster prepared with this method looks like:



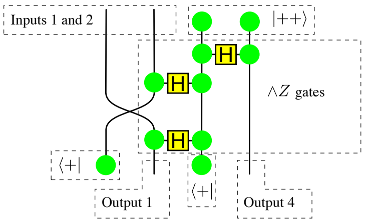
Again, dashed lines indicate the individual components. While conventional methods require some calculation to show that these methods of preparation produce the same state, using the spider theorem, the two diagrammatic forms are immediately equivalent:



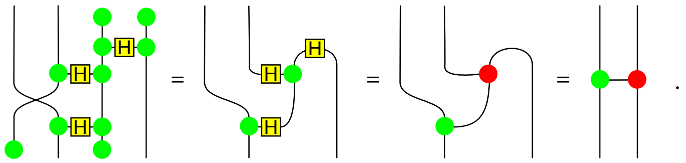
Example 13 (Measurement-based quantum computing). Consider a measurement-based program involving 4 qubits, which computes a $\wedge\mathsf{X}$ gate upon its inputs. In the syntax of the measurement calculus [6] this pattern is written:

$$M_2^0 M_4^0 E_{13} E_{23} E_{34} N_3 N_4 .$$

Reading from right to left, this specifies that qubits 3 and 4 should be prepared in a $|+\rangle$ state, then $\wedge\mathsf{Z}$ operations should be applied pairwise between qubits 1 and 3, 2 and 3, and 3 and 4; finally X basis measurements should be performed upon qubits 2 and 4. Qubits 1 and 2 are the inputs and qubits 1 and 4 are the outputs. We represent this pattern diagrammatically as:



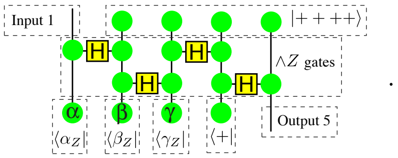
The spider theorem allows this one-way program to be rewritten to a $\wedge\mathsf{X}$ gate in three steps:



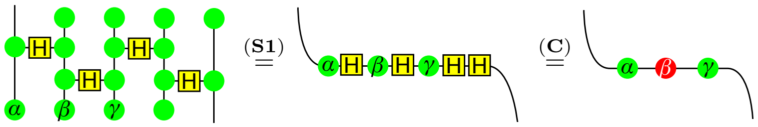
Our next example is a one-way program implementing an arbitrary 1-qubit unitary. Recall that any single qubit unitary map U has an Euler decomposition as such that $U = Z_\gamma X_\beta Z_\alpha$. Such a unitary can be implemented by the following 5-qubit measurement pattern:

$$M_3^\gamma M_2^\beta M_1^\alpha E_{12} E_{23} E_{34} E_{45} N_2 N_3 N_4 N_5 \,.$$

The graphical form of this pattern is shown below:



A sequence of simple rewrites shows that the one-way program intended to compute such a unitary does indeed produce the desired map.



Notice that in this example, and in Example 10, measurement is represented via post-selection—that is, by projection. To capture the non-determinism of quantum measurements, the syntax and semantics of the calculus must be extended; see [8,3].

References

1. Backens, M.: The ZX-calculus is complete for stabilizer quantum mechanics. In: Proceedings of Quantum Physic and Logic IX (2012)
2. Coecke, B., Duncan, R.: Interacting Quantum Observables. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 298–310. Springer, Heidelberg (2008)
3. Coecke, B., Duncan, R.: Interacting quantum observables: categorical algebra and diagrammatics. New Journal of Physics 13, 043016 (2011), arXiv:0906.4725
4. Coecke, B., Duncan, R., Kissinger, A., Wang, Q.: Strong complementarity and non-locality in categorical quantum mechanics. In: Chiribella, G., Spekkens, R.W. (eds.) Proceedings of 27th IEEE Conference on Logic in Computer Science (LiCS). Extended version to appear in: Quantum Theory: Informational Foundations and Foils. Springer (2012)
5. Coecke, B., Edwards, B., Spekkens, R.W.: Phase groups and the origin of non-locality for qubits. ENTCS 271(2), 15–36 (2011), arXiv:1003.5005
6. Danos, V., Kashefi, E., Panangaden, P.: The measurement calculus. Journal of the ACM 54(2) (2007), arXiv:quant-ph/0412135
7. Duncan, R., Perdrix, S.: Graph States and the Necessity of Euler Decomposition. In: Ambos-Spies, K., Löwe, B., Merkle, W. (eds.) CiE 2009. LNCS, vol. 5635, pp. 167–177. Springer, Heidelberg (2009)
8. Duncan, R., Perdrix, S.: Rewriting Measurement-Based Quantum Computations with Generalised Flow. In: Abramsky, S., Gavoille, C., Kirchner, C., Meyer auf der Heide, F., Spirakis, P.G. (eds.) ICALP 2010. LNCS, vol. 6199, pp. 285–296. Springer, Heidelberg (2010)
9. Gottesman, D.: Stabilizer codes and quantum error correction. Ph.D. Thesis, Caltech (2007), arXiv:quant-ph/9705052
10. Hillebrand, A.: Quantum protocols involving multiparticle entanglement and their representations in the ZX-calculus. MSc. thesis, University of Oxford (2011)
11. Horsman, C.: Quantum picturalism for topological cluster-state computing. New Journal of Physics 13, 095011 (2011), arXiv:1101.4722
12. Zamdzhiev, V.N.: An abstract approach towards quantum secret sharing. MSc. thesis, University of Oxford (2012)

One-Way Reversible Multi-head Finite Automata

Martin Kutrib and Andreas Malcher

Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
{kutrib,malcher}@informatik.uni-giessen.de

Abstract. One-way multi-head finite automata are considered towards their ability to perform reversible computations. It is shown that, for every number $k \geq 1$ of heads, there are problems which can be solved by one-way k-head finite automata, but not by any one-way reversible k-head finite automaton. Additionally, a proper head hierarchy is obtained for one-way reversible multi-head finite automata. Finally, decidability problems are considered. It turns out that one-way reversible finite automata with two heads are still a powerful model, since almost all commonly studied problems are not even semidecidable.

1 Introduction

Reversible computations and reversible versions of computational devices have gained much interest in the recent years. For a reversible computational device it is essential that for every state which the device may enter there is both a uniquely defined successor state and a uniquely defined predecessor state. Thus, reversible devices show a forward and backward deterministic behavior. One motivation for studying such devices is given by the physical observation that the loss of information in irreversible computations results in heat dissipation [3,10]. On the other hand, it is also of great theoretical interest how information is processed in computational devices and in which way, if possible, computations can be made information preserving.

The first investigations of reversible computations date back to the early seventies when reversible Turing machines have been introduced and studied [3]. The main result obtained in [3] is that every Turing machine can be simulated by a reversible one. Thus, any problem solved by a Turing machine can also be solved reversibly. Since Turing machines are a strong computational model, the question of whether also computations in weaker devices can always be made reversible suggests itself. The question has been answered negatively for finite automata [1,15] and pushdown automata [8]. Hence, there exist problems solvable by finite or pushdown automata which are inherently irreversible for these devices. Reversible versions of parallel models have been studied as well. For example, the recent paper [12] summarizes results on reversible cellular automata, logic gates, logic circuits, and logic elements with memory. Reversible cellular automata in connection with language theory have been studied in [6,7]. Other reversible computational devices and other perspectives on reversible computing can be found in [14,18,19].

Recently, reversible multi-head finite automata, which are basically finite automata equipped with multiple reading heads that move on the input with two-way motion, have been introduced in [13], where also several examples of languages are given which can be accepted by reversible multi-head finite automata. These examples include the languages $\{ w \in \{a,b\}^* \mid w = w^R \}$ and $\{ a^{2^n} \mid n \geq 0 \}$. Multi-head finite automata are also interesting from a computational complexity point of view, since they characterize the complexity class L of languages accepted by deterministic logarithmically space-bounded Turing machines. The open problem of whether reversible multi-head finite automata characterize reversible deterministic logarithmic space or, more general, characterize deterministic logarithmic space has recently be solved in the affirmative in [2].

In this paper, we continue the investigation of reversible multi-head finite automata, but we restrict our interest to multi-head finite automata with one-way head motion. In general, this restriction leads to strictly weaker devices, which is true for the reversible case as well. Moreover, we show in Section 3 that for every number $k \geq 2$ of heads there a languages which can be accepted by irreversible k-head finite automata, but not by any reversible k-head finite automaton. This shows that reversibility is in fact a restriction in the one-way case. This is also in analogy to the results for (one-way) finite automata [15] and pushdown automata [8].

For not necessarily reversible multi-head finite automata, the existence of a proper head hierarchy is known [17]. This means that $k+1$ heads are more powerful than k heads. In Section 4, we obtain that this is also true for the reversible case. In detail, it is shown that the languages used in [17] can be accepted by reversible multi-head finite automata. Since we know by the results of Section 3 that reversible k-head finite automata are less powerful than k-head finite automata, it is an obvious question whether undecidable problems for k-head finite automata become decidable for reversible k-head finite automata. In Section 5 we will give a negative answer. All commonly studied decidability problems turn out to be not even semidecidable for reversible two-head finite automata. Finally, we show that the problem to determine, whether a given multi-head finite automaton is reversible, is also non-semidecidable. This is in contrast to the results for finite automata [15] and pushdown automata [8], but in analogy to the results for cellular automata [6,7].

2 Preliminaries and Definitions

Let A^* denote the set of all words over the finite alphabet A. The empty word is denoted by λ, and $A^+ = A^* \setminus \{\lambda\}$. The reversal of a word w is denoted by w^R and for the length of w we write $|w|$. Set inclusion is denoted by $\subseteq$, and strict set inclusion by $\subset$.

Let $k \geq 1$ be a natural number. A one-way k-head finite automaton is a finite automaton having a single read-only input tape whose inscription is the input word in between two endmarkers (we provide two endmarkers in order to have a definition consistent with two-way finite automata). The k heads of the automaton can move to the right or stay on the current tape square but not beyond the endmarkers. A formal definition is:

Definition 1. *A* deterministic one-way k-head finite automaton (1DFA(k)) *is a system* $M = \langle S, A, k, \delta, \rhd, \lhd, s_0, F \rangle$, *where*

1. S *is the finite set of* internal states*,*
2. A *is the finite set of* input symbols*,*
3. $k \geq 1$ *is the* number of heads*,*
4. $\rhd \notin A$ *is the* left *and* $\lhd \notin A$ *is the* right endmarker*,*
5. $s_0 \in S$ *is the* initial state*,*
6. $F \subseteq S$ *is the set of* accepting states*, and*
7. $\delta : S \times (A \cup \{\rhd, \lhd\})^k \to S \times \{0,1\}^k$ *is the partial transition function, where* 1 *means to move the head one square to the right, and* 0 *means to keep the head on the current square. Whenever* $(s', (d_1, d_2, \ldots, d_k)) = \delta(s, (a_1, a_2, \ldots, a_k))$ *is defined, then* $d_i = 0$ *if* $a_i = \lhd$*, for* $1 \leq i \leq k$*.*

A 1DFA(k) starts with all of its heads on the left endmarker. It halts when the transition function is not defined for the current situation. A *configuration* of a 1DFA(k) $M = \langle S, A, k, \delta, \rhd, \lhd, s_0, F \rangle$ at some time $t \geq 0$ is a triple $c_t = (w, s, P)$, where $w \in A^*$ is the input, $s \in S$ is the current state, and $P = (p_1, p_2, \ldots, p_k) \in \{0, 1, \ldots, |w| + 1\}^k$ gives the current head positions. If a position p_i is 0, then head i is scanning the symbol $\rhd$, if it satisfies $1 \leq p_i \leq |w|$, then the head is scanning the p_ith letter of w, and if it is $|w| + 1$, then the head is scanning the symbol $\lhd$. The *initial configuration* for input w is set to $(w, s_0, (0, \ldots, 0))$. During its course of computation, M runs through a sequence of configurations. One step from a configuration to its successor configuration is denoted by $\vdash$. Let $w = a_1 a_2 \ldots a_n$ be the input, $a_0 = \rhd$, and $a_{n+1} = \lhd$, then we set

$$(w, s, (p_1, p_2, \ldots, p_k)) \vdash (w, s', (p_1 + d_1, p_2 + d_2, \ldots, p_k + d_k))$$

if and only if $(s', (d_1, d_2, \ldots, d_k)) = \delta(s, (a_{p_1}, a_{p_2}, \ldots, a_{p_k}))$. As usual we define the reflexive, transitive closure of $\vdash$ by $\vdash^*$. Note, that due to the restriction of the transition function, the heads cannot move beyond the endmarkers.

The language accepted by a 1DFA(k) is precisely the set of words w such that there is some computation beginning with $\rhd w \lhd$ on the input tape and ending with the 1DFA(k) halting in an accepting state:

$$L(M) = \{\, w \in A^* \mid (w, s_0, (0, \ldots, 0)) \vdash^* (w, s, (p_1, p_2, \ldots, p_k)), s \in F, \\ \text{and } M \text{ halts in } (w, s, (p_1, p_2, \ldots, p_k)) \,\}.$$

So, an input is accepted if and only if the computation halts in an accepting state. In all other cases the input is rejected. That is, it is rejected if the computation halts in an non-accepting state, or if the computation runs into an infinite loop. In the latter case eventually all heads are stationary since the machines are one-way. In general, the family of all languages that are accepted by some device X is denoted by $\mathscr{L}(X)$.

Now we turn to one-way multi-head finite automata that are reversible for any computation that starts from an initial configuration. Basically, reversibility is meant with respect to the possibility of stepping the computation back and forth. So, the automata have also to be backward deterministic. In particular, the automata reread the input symbols which they have been read in a preceding forward computation step. So, for reverse computation steps the heads of the input tape are either moved to the *left* or stay stationary. Figurative one can assume that in a forward step, first the current input symbols are read and then the heads are moved. In a backward step the heads are first moved and then the symbols are read.

Let M be a 1DFA(k) and C be the set of all reachable configurations that occur in *any* computation of M beginning with an initial configuration, and $(w, s, (p_1, p_2, \ldots, p_k)) \in C$ with $w = x_1 x_2 \cdots x_n \in A^n$, $x_0 = \triangleright$, and $x_{n+1} = \triangleleft$. Then M is said to be *reversible* (REV-1DFA(k)), if the following two conditions are fulfilled.

1. For any two transitions

$$\begin{aligned}\delta(s_1, (a_1, a_2, \ldots, a_k)) &= (s, (d_1, d_2, \ldots, d_k)) \text{ and}\\ \delta(s'_1, (a'_1, a'_2, \ldots, a'_k)) &= (s, (d'_1, d'_2, \ldots, d'_k))\end{aligned}$$

it holds $(d_1, d_2, \ldots, d_k) = (d'_1, d'_2, \ldots, d'_k)$.

2. There is at most one transition of the form

$$\delta(s', (x_{p_1 - d_1}, x_{p_2 - d_2}, \ldots, x_{p_k - d_k})) = (s, (d_1, d_2, \ldots, d_k)).$$

Condition (1) means that transitions yielding the same state have to move the heads in the same way. In addition, Condition (2) says that for any reachable configuration the predecessor state is uniquely determined by the state (which implies the head movements) and the input symbols read. If the prerequisite to consider *reachable* configurations only is relaxed, then this definition is essentially the same as used in [13] for two-way reversible multi-head finite automata. It is similar to that of a reversible Turing machine in quintuple form [12]. However, here we stick with reachable configurations.

In order to clarify our notion we continue with an example.

Example 2. The non-context-free language $\{\, a^n b^n c^n \mid n \geq 1 \,\}$ is accepted by the REV-1DFA(2) $M = \langle \{s_0, s_1, s_2, s_3, s_f\}, \{a, b, c\}, 2, \delta, \triangleright, \triangleleft, s_0, \{s_f\} \rangle$, where the transition function δ is as follows.

(1) $\delta(s_0, \triangleright, \triangleright) = (s_0, 0, 1)$	(6) $\delta(s_2, b, c) = (s_2, 1, 1)$
(2) $\delta(s_0, \triangleright, a) = (s_0, 0, 1)$	(7) $\delta(s_2, b, \triangleleft) = (s_3, 1, 0)$
(3) $\delta(s_0, \triangleright, b) = (s_1, 1, 1)$	(8) $\delta(s_3, c, \triangleleft) = (s_3, 1, 0)$
(4) $\delta(s_1, a, b) = (s_1, 1, 1)$	(9) $\delta(s_3, \triangleleft, \triangleleft) = (s_f, 0, 0)$
(5) $\delta(s_1, a, c) = (s_2, 1, 1)$	

Since no transition violates the definition of reversibility, a simple inspection of the definition of δ shows that M cannot be irreversible.

The transitions (1) through (3) are used by M to move its second head to the first input symbol b and to change to state s_1. Then both heads are moved simultaneously to the right by transition (4). If the first head reads the last a when the second head sees the first c, the number of a's is equal to the number of b's. In this case, by transition (5), M changes to state s_2. Similarly as before, now transitions (6) and (7) are used to move the heads simultaneously to the right until the second head reads the right endmarker. If the first head sees the last symbol b at the same time, the number of b's is equal to the number of c's and, thus, the input is to be accepted. In exactly this situation, the second head is moved to the right endmarker by transition (8). Finally, by transition (9), M changes to state s_f and accepts if both heads are on the right endmarker. □

Example 3. Only slight modifications of the construction given in Example 2 show that, for all $m \geq 1$, the language $\{ a_1^n a_2^n \cdots a_m^n \mid n \geq 1 \}$ over the alphabet $\{a_1, a_2, \ldots, a_m\}$ is accepted by a REV-1DFA(2) as well. □

3 Computational Capacity

In this section, the computational capacity of REV-1DFA(k)s is considered. In particular, reversible classes are compared with ordinary ones and classical devices. We start at the bottom of the hierarchy of language classes. It is well known that any regular language is accepted by some deterministic one-way one-head finite automaton. Reversible variants of these devices have been defined and investigated in [1,15]. It turned out that there are regular languages for which no reversible deterministic finite automaton exists.

Corollary 4. *$\mathscr{L}$(REV-1DFA(1)) $\subset$ $\mathscr{L}$(1DFA(1)).*

In fact, the properness of the previous corollary can already be shown for unary languages. For example, the witness language $L = \{ a^x \mid x \geq 42 \}$ can be used to show the following lemma.

Lemma 5. *There is a unary regular language which is not accepted by any REV-1DFA(1).*

Proof. We use $L = \{ a^x \mid x \geq 42 \}$ as witness language. Any 1DFA(1) M accepting L has to ensure that no word shorter than 42 is accepted. To this end, at least 43 states are necessary, where no loop may appear before M has read 42 input symbols. On the other hand, a loop has to appear in order to accept all of the infinitely many words. Since L is unary, any accepting computation of M on inputs long enough starts with at least 43 different states $p_0, p_1, \ldots, p_{42}$ possibly followed by some different states $p_{43}, \ldots, p_i$ until the successor state p_j of p_i is one of the states that appeared before, say state $42 \leq j \leq i$. So, we have $\delta(p_i, a) = (p_j, d)$ and $\delta(p_{j-1}, a) = (p_j, d')$. If M is reversible, we obtain $d = d'$ and $p_i = p_{j-1}$ and, thus, a contradiction. Therefore, M is not reversible. □

In order to accept all unary regular languages reversibly one more head is sufficient.

Theorem 6. *Any unary regular language is accepted by some REV-1DFA*(2)*.*

Proof. Given a unary regular language L, we consider the deterministic finite automaton M accepting it. By definition, M moves its head in every transition. So, its transition function is such that it runs through an infinite loop after processing an initial computation. In general, this means that it passes through a sequence of different states as $p_0, p_1, \ldots, p_i$, where the successor of p_i is a state that appeared before, say p_j with $0 \leq j \leq i$.

If $j = 0$, the initial computation is empty. So, M runs through one loop from the very beginning. Clearly, in this case it is reversible. If $j \geq 1$, automaton M cannot be reversible, since the predecessor state of p_j is not unique. Here, we can use the second head to determine the predecessor state as follows. The second head keeps stationary on the left endmarker until state p_{j-1} appears. Subsequently, it is moved together with the first head in every step. In this way, the predecessor state of p_j can be determined as p_{j-1} if the second head reads the left endmarker, and as p_i otherwise. $\square$

Now we turn to the comparison with language classes accepted by variants of pushdown automata.

Theorem 7. *For all $k \geq 2$, the language class $\mathscr{L}$(REV-1DFA(k)) is incomparable with the (deterministic) (reversible) (linear) context-free languages.*

Proof. It is well known that the deterministic context-free languages are a proper sub-class of the context-free languages. In [8] it is shown that the reversible context-free languages are, in turn, a proper sub-class of the deterministic context-free ones. The mirror language $\{\, wcw^R \mid w \in \{a,b\}^* \,\}$ is deterministic reversible linear context free [8], but not accepted by any 1DFA(k). Conversely, Example 3 shows that the non-context-free language $\{\, a^n b^n c^n \mid n \geq 1 \,\}$ is accepted by some REV-1DFA(2). $\square$

At the other end of the hierarchy we inherit upper bounds from the ordinary 1DFA(k)s, as deterministic one-way k-head finite automata are strictly weaker than deterministic *two-way* k-head finite automata that characterize the complexity class L [4].

Corollary 8. *For all $k \geq 1$, the language class $\mathscr{L}$(REV-1DFA(k)) is a proper sub-class of L, that is, the languages accepted by deterministic log-space bounded Turing machines.*

Next we turn to compare the computational power of REV-1DFA(k) and 1DFA(k). By Corollary 4 we know already that, for one head, reversible devices are strictly weaker than ordinary ones. In the following we generalize this result to any number of heads, and continue with two heads. To this end, we use $L^{(2)} = \{\, (a^+b^+)^n \$ b^n \mid n \geq 1 \,\}$ as witness language.

Lemma 9. *The language $L^{(2)}$ is not accepted by any REV-1DFA(2).*

Proof. In contrast to the assertion, assume $L^{(2)}$ is accepted by a REV-1DFA(2) $M = \langle S, A, k, \delta, \rhd, \lhd, s_0, F\rangle$ with m states. For some fixed ℓ large enough, let $L^{(2,\ell)} = \{ (a^\ell b^\ell)^n \$ b^n \mid n \geq 1 \}$ be a subset of $L^{(2)}$. We consider accepting computations of M on words w from $L^{(2,\ell)}$, in particular, the unique configurations c_w where one head is moved to the \$ and the other head is still to the left of the \$. We safely can assume that M does not accept before one of its heads arrived at the \$. Dependent on the state and the position of the left head in the configurations c_w, the words from $L^{(2,\ell)}$ are partitioned into classes. Two words w and w' are in the same class if and only if the states in c_w and $c_{w'}$ are identical, the left head reads the same input symbol $x \in \{a, b\}$, and the positions of the left head in its current input x-block coincide. The latter means that in both configurations the left head has the same number of adjacent symbols x to its left (or right). Since the number of adjacent symbols x is at most ℓ, and ℓ is fixed, we can choose one class, say C, containing infinitely many words with different n.

Now we elaborate on the position of the left head. Let $w, w' \in C$ such that w includes n subwords ab and w' includes $n' > n$ subwords ab. If the distances of the left head to the \$ symbol are the same in c_w and $c_{w'}$, then M would accept the inputs $(a^\ell b^\ell)^n \$ b^{n'}$ and $(a^\ell b^\ell)^{n'} \$ b^n$ not belonging to $L^{(2)}$. We conclude that for words in C the positions of the left head may be arbitrarily far from the \$.

Next we turn to the continuations of the computations. If the head on the \$ symbol is stationary until the other one has passed over more than m further subwords ab, automaton M gets into a loop that cannot be left until the left head has also arrived at the \$. Since in this case we obtain immediately a contradiction, we conclude that eventually the right head moves from the \$ while the position of the left head still may be arbitrarily far from the \$. Moreover, for ℓ large enough, the right head has eventually to be stationary on some b while the left head passes over an a- or a b-block. Clearly, the blocks are passed over in loops. On the other hand, similarly as above, we obtain immediately a contradiction when the right head is totally stationary while the left head gets closer and closer to the \$. Therefore, the right head has to move while the left head passes from an a-block to a b-block or vice versa. More precisely, let q_0 be the state appearing when M moves the left head on the first symbol of the new block. Moreover, let $q_1, q_2, \ldots, q_i$ be the shortest sequence of states following q_0 until M runs again into a loop to pass over the new block. Then all states $q_0, q_1, \ldots, q_i$ are different, and the computation continues after one cycle of the loop with some state q_j, $0 \leq j \leq i$. Since M may not move the right head in the loop, but has to move the right head while running through these states, at least one state does not belong to the loop. So we have $j \geq 1$. However, if $j = 1$, then $\delta(q_i, b, b) = (q_1, 1, 0)$ and $\delta(q_0, b, b) = (q_1, d_1, 1)$. Therefore, M cannot be reversible. If $j \geq 2$, then $\delta(q_i, b, b) = (q_j, 1, 0)$ and $\delta(q_{j-1}, b, b) = (q_j, d_1, d_2)$. But even if $d_1 = 1$ and $d_2 = 0$, the states q_i and q_{j-1} are different though the input symbols are identical in both transitions and, thus, M cannot be reversible in this case, either. □

In [17] the head hierarchy for one-way multi-head finite automata is shown. The proof provides witness languages $L_{n_k} \subseteq \{a, b, \$\}^*$ which are accepted by 1DFA(k)s but which cannot be accepted by any 1DFA($k-1$). Moreover, the proof that L_{n_k} is not accepted by any 1DFA($k-1$) relies on the fact that $k-1$ heads are too few in order to compare certain subwords. In turn, a closer look on the proof reveals that in accepting computations all k heads have to read some inner symbol of the input simultaneously. Now we utilize this observation for our purposes. Let L'_{n_k} be the language L_{n_k} over a disjoint copy of $\{a, b, \$\}$, and define $L^{(k)} = \{\, (a^+b^+)^n v b^n \mid n \geq 1, v \in L'_{n_{k-1}} \,\}$.

Lemma 10. *For all $k \geq 3$, language $L^{(k)}$ is not accepted by any REV-1DFA(k).*

Proof. The proof of Lemma 9 is generalized as follows. Basically, the \$ in $L^{(2)}$ is replaced by a word v from $L'_{n_{k-1}}$. Due to the necessity that $k-1$ heads have to read symbols of v simultaneously to check whether v belongs to $L'_{n_{k-1}}$, the first part of the reasoning is as before with the right head replaced by the $k-1$ heads. In particular, for the sub-language $L^{(k,\ell)}$ some arbitrary but fixed $v_0 \in L'_{n_{k-1}}$ can be chosen, and the configurations for the partitioning are those where the last of the $k-1$ heads is moved on the first symbol of v_0. However, in addition to the conditions of Lemma 9 now all positions of the remaining $k-2$ heads have to be the same for words in the same class. But since these positions are all within v_0, there are $(|v_0|)^{k-2}$ possibilities. So, for n and ℓ large enough, class C can still be chosen infinite.

For the continuations of the computations, an accepting REV-1DFA(k) M has to move its left head over the a- and b-blocks in loops as before. During these loops, again, the $k-1$ heads eventually become stationary. The remaining reasoning is exactly as in the proof of Lemma 9. □

It is not hard to construct ordinary 1DFA(k)s that accept the languages $L^{(k)}$. For example, a 1DFA(2) can accept $L^{(2)}$ by moving the first head to the \$, and subsequently, iterating to move the second head over a subword a^+b^+ and the first head one position to the right. The input is accepted when the first head reaches the endmarker and the second head the \$ in the same iteration. So, the next theorem is shown by Lemmas 9 and 10, and Corollary 4.

Theorem 11. *For all $k \geq 1$, $\mathscr{L}$(REV-1DFA(k)) $\subset$ $\mathscr{L}$(1DFA(k)).*

4 Head Hierarchy

The question of the existence of a proper head hierarchy for ordinary multi-head finite automata has been raised in [16] and, finally, been answered in the affirmative in [17]. Here we obtain that there is a proper head hierarchy for reversible one-way multi-head finite automata as well. The witness languages used in [17] have essentially the form

$$L_n = \{\, \$w_1\$w_2\$\ldots\$w_{2n} \mid w_i \in \{a, b\}^* \text{ and } w_i = w_{2n+1-i}, \text{ for } 1 \leq i \leq n \,\},$$

and it is shown that, for any $k \geq 2$, language $L_{\frac{k(k-1)}{2}}$ can be accepted by some 1DFA(k), but not by any 1DFA(ℓ) with $\ell < k$. In order to prove the hierarchy one can construct a REV-1DFA(k) that accepts $L_{\frac{k(k-1)}{2}}$ as well. The principal idea is in essence already mentioned in [16]. It has slightly to be adapted to obtain reversible computations. Basically, the algorithm from [16] is as follows (see Example 12). For $n = \frac{k(k-1)}{2}$, any word in L_n consists of $2n = k(k-1)$ blocks starting with $.

Step 1(a): Move head 1 to the beginning of block $2n - k + 2$ and heads $i \in \{2, 3, \dots, k\}$ to the beginning of block $i - 1$.

Step 1(b): Use head 1 and head k to check the equality of the blocks $2n - (k-1) + 1$ and $k - 1$. Use head 1 and head $k - 1$ to check the equality of blocks $2n - (k-2) + 1$ and $k - 2$. Iterate this behavior. Finally, use head 1 and head 2 to check the equality of blocks $2n$ and 1.

Step 2(a): Move head 2 to the beginning of block $2n - 2(k-1) + 2$ and heads $i \in \{3, 4, \dots, k\}$ to the beginning of block $k + i - 3$.

Step 2(b): Use head 2 and head k to check the equality of the blocks $2n - 2(k-1) + 2$ and $2k - 3$. Use head 2 and head $k - 1$ to check the equality of blocks $2n - 2(k-1) + 3$ and $2k - 4$. Iterate this behavior. Finally, use head 2 and head 3 to check the equality of blocks $2n - k + 1$ and k.

Step $k-2$(a): Move head $k - 2$ to the beginning of block $n + 2$, head $k - 1$ to the beginning of block $n - 2$, and head k to the beginning of block $n - 1$.

Step $k-2$(b): Use head $k - 2$ and head k to check the equality of blocks $n + 2$ and $n - 1$. Use head $k - 2$ and head $k - 1$ to check the equality of blocks $n + 3$ and $n - 2$.

Step $k-1$(a): Move head $k - 1$ to the beginning of block $n + 1$ and head k to the beginning of block n.

Step $k-1$(b): Use head $k - 1$ and head k to check the equality of blocks $n + 1$ and n.

Example 12. For $k = 4$ we obtain the language

$$L_6 = \{\, \$w_1\$w_2\$\cdots\$w_{12} \mid w_i \in \{a, b\}^* \text{ and } w_i = w_{13-i},\ \text{for } 1 \leq i \leq n \,\}.$$

Thus, we have to test with four heads whether $w_1 = w_{12}$, $w_2 = w_{11}$, $w_3 = w_{10}$, $w_4 = w_9$, $w_5 = w_8$, and $w_6 = w_7$. Applying the above construction, we can test in Step 1 with the help of heads $1, 2, 3$ and 4 whether $w_3 = w_{10}$, $w_2 = w_{11}$, and $w_1 = w_{12}$. In Step 2 it is tested with the help of heads $2, 3$ and 4 whether $w_5 = w_8$ and $w_4 = w_9$. Finally, in Step 3 it is tested with heads 3 and 4 whether $w_6 = w_7$. □

We see that for any $k \geq 2$ a fixed procedure is run, where phases of positioning the heads alternate with phases of equality checking. Let us now sketch how the algorithm can be implemented by reversible transitions.

To move the heads suitably, the number of symbols $ passed by the single heads is stored as part of the states. If a head has reached its correct position, it waits on the first symbol \$ of the block. If all heads have reached their correct position, the checking phase is entered. Thus, Step 1(a) leads to the following transitions, where $z \in \{a, b\}$:

$$\begin{aligned}
\delta((s_0, 0, \ldots, 0), \rhd, \ldots, \rhd) &= ((p_1, 0, \ldots, 0), 1, \ldots, 1),\\
\delta((p_1, 0, \ldots, 0), \$, \ldots, \$) &= ((p_2, 1, \ldots, 1), 1, 0, 1, \ldots, 1),\\
\delta((p_2, 1, \ldots, 1), z, \$, z, \ldots, z) &= ((p_2, 1, \ldots, 1), 1, 0, 1, \ldots, 1),\\
\delta((p_2, 1, \ldots, 1), \$, \$, \ldots, \$) &= ((p_3, 2, 1, 2, \ldots, 2), 1, 0, 0, 1, \ldots, 1),\\
\delta((p_3, 2, 1, 2, \ldots, 2), z, \$, \$, z, \ldots, z) &= ((p_3, 2, 1, 2, \ldots, 2), 1, 0, 0, 1, \ldots, 1),\\
&\vdots\\
\delta((p_k, k-1, 1, 2, \ldots, k-1), \$, \$, \ldots, \$) &= ((p_{k+1}, k, 1, 2, \ldots, k-1), 1, 0, \ldots, 0),\\
\delta((p_{k+1}, k, 1, 2, \ldots, k-1), z, \$, \ldots, \$) &= ((p_{k+1}, k, 1, 2, \ldots, k-1), 1, 0, \ldots, 0).
\end{aligned}$$

With similar transitions the first head is driven to the correct position. Finally, the following transition ends the positioning phase and starts the checking phase.

$$\begin{aligned}
&\delta((p, 2n-k+1, 1, 2, \ldots, k-1), \$, \ldots, \$) =\\
&\qquad\qquad ((c_1, 2n-k+2, 1, 2, \ldots, k-1), 1, 0, \ldots, 0, 1)
\end{aligned}$$

The checking phase starts after positioning the heads correctly. When the equality of all subwords involved has been checked, the next positioning phase is entered. Thus, Step 1(b) leads to the following transitions, where $z \in \{a, b\}$.

$$\begin{aligned}
&\delta((c_1, 2n-k+2, 1, 2, \ldots, k-1), z, \$, \ldots, \$, z) =\\
&\qquad\qquad ((c_1, 2n-k+2, 1, 2, \ldots, k-1), 1, 0, \ldots, 0, 1),\\
&\delta((c_1, 2n-k+2, 1, 2, \ldots, k-1), \$, \ldots, \$) =\\
&\qquad\qquad ((c_2, 2n-k+3, 1, 2, \ldots, k), 1, 0, \ldots, 0, 1, 0),\\
&\delta((c_2, 2n-k+3, 1, 2, \ldots, k-2, k), z, \$, \ldots, \$, z, \$) =\\
&\qquad\qquad ((c_2, 2n-k+3, 1, 2, \ldots, k-2, k), 1, 0, \ldots, 0, 1, 0),\\
&\delta((c_2, 2n-k+3, 1, 2, \ldots, k-2, k), \$, \ldots, \$) =\\
&\qquad\qquad ((c_3, 2n-k+4, 1, 2, \ldots, k-1, k), 1, 0, \ldots, 0, 1, 0, 0).
\end{aligned}$$

With similar transitions the first head checks all subwords involved. Finally, the following transition ends the checking phase and starts the next positioning phase.

$$\begin{aligned}
&\delta((c_{k-1}, 2n, 1, 3, \ldots, k-1, k), \lhd, \$, \ldots, \$,) =\\
&\qquad\qquad ((p'_1, 2n, 2, 3, \ldots, k-1, k), 0, 1, \ldots, 1)
\end{aligned}$$

The remaining steps can similarly be implemented into transitions. We can partition the state set into the initial state s_0, states from a subset P for positioning the heads suitably, and a subset C for checking equality. Furthermore, states from P and C alternate and no state is reentered after an alternation took place. Moreover, the number of symbols $ passed together with the input symbols seen by the heads uniquely define the next state and head moves. Since all states correspond unambiguously to head movements, it is also possible to reconstruct the predecessor state from the current state and the input symbols seen. Therefore, the 1DFA(k) constructed is reversible. Thus, for any $k \geq 2$, $L_{\frac{k(k-1)}{2}}$ belongs to $\mathscr{L}(\text{REV-1DFA}(k))$. Since $L_{\frac{k(k-1)}{2}} \notin \mathscr{L}(\text{1DFA}(k-1))$ [17], we obtain the following proper head hierarchy.

Theorem 13. *For all $k \geq 1$, $\mathscr{L}(\text{REV-1DFA}(k)) \subset \mathscr{L}(\text{REV-1DFA}(k+1))$.*

5 Decidability Problems

In this section, we study decidability questions for reversible one-way multi-head finite automata. The undecidability results are obtained by reductions of the emptiness problem for deterministic linearly space bounded one-tape, one-head Turing machines, so-called linear bounded automata (LBA). The following approach has recently be used in [9] to show the undecidability of emptiness for 1DFA(2) with four *states*. Since the 1DFA(2)s constructed there are not reversible, the construction has to be improved. For the sake of completeness, we summarize the necessary definitions and notations.

For the reduction, we consider strings which record all configurations of an accepting computation of a given LBA that gets its input in between two endmarkers. By standard techniques an arbitrary LBA can effectively be converted into an equivalent one that makes no stationary moves, accepts by halting in some unique state f on the leftmost input symbol, and is sweeping, that is, the read-write head changes its direction at endmarkers only. For example, the latter property can be achieved by marking the current head position in the input, remembering the current state, and to update both during two consecutive sweeps.

Let Q be the state set of some LBA M, where q_0 is the initial state, $T \cap Q = \emptyset$ is the tape alphabet containing the endmarkers $\triangleright$ and $\triangleleft$, and $\Sigma \subset T$ is the input alphabet. Since M is sweeping, the set of states can be partitioned into Q_R and Q_L of states appearing in right-to-left and in left-to-right moves. A configuration of M can be written as a string of the form $\triangleright T^*QT^*\triangleleft$ such that, $\triangleright t_1t_2\cdots t_ist_{i+1}\cdots t_n\triangleleft$ is used to express that $\triangleright t_1t_2\cdots t_n\triangleleft$ is the tape inscription, M is in state s, for $s \in Q_R$ scans tape symbol t_{i+1}, and for $s \in Q_L$ scans tape symbol t_i. Let Σ' be a primed copy of the input alphabet Σ. Now we consider words of the form $\$w_0\$w_1\$\cdots\$w_m$, where $\$ \notin T \cup Q$, $w_i \in T^*QT^*$ are configurations of M with endmarkers chopped off, w_0 is an initial configuration of the form $q_0\Sigma^+\Sigma'$ with the last input symbol suitably marked, $w_m \in \{f\}T^*$ is a halting, that is, accepting configuration, and w_{i+1} is the successor configuration of w_i. These configurations are now encoded so that every state symbol is merged together with its both adjacent symbols into a metasymbol. We assume that the length of the LBA input is at least two and rewrite every substring of

$\$w_0\$\cdots\$w_m$ having the form tqt' to $[t, q, t']$, where $q \in Q$, $t, t' \in T \cup \Sigma' \cup \{\$\}$. The set of these encodings is defined to be the set of *valid computations* of M. We denote it by VALC(M).

Example 14 ([9]). We consider the following computation of an LBA on input $x_1x_2x_3$ where each configuration consists of the current tape inscription, the current state, and the current position of the read-write head, $q_0, \dots, q_3, f \in Q_R$ and $p_0, \dots, p_3 \in Q_L$.

tape	state	position
$\triangleright x_1x_2x_3\triangleleft$	q_0	1
$\triangleright y_1x_2x_3\triangleleft$	q_1	2
$\triangleright y_1y_2x_3\triangleleft$	q_2	3
$\triangleright y_1y_2y_3\triangleleft$	q_3	4
$\triangleright y_1y_2y_3\triangleleft$	p_3	3
$\triangleright y_1y_2z_3\triangleleft$	p_2	2
$\triangleright y_1z_2z_3\triangleleft$	p_1	1
$\triangleright z_1z_2z_3\triangleleft$	p_0	0
$\triangleright z_1z_2z_3\triangleleft$	f	1

The corresponding valid computation is:

$$[\$, q_0, x_1]x_2x_3'\$[y_1, q_1, x_2]x_3\$y_1[y_2, q_2, x_3]\$y_1y_2[y_3, q_3, \$]y_1y_2[y_3, p_3, \$]$$
$$y_1[y_2, p_2, z_3]\$[y_1, p_1, z_2]z_3[\$, p_0, z_1]z_2z_3[\$, f, z_1]z_2z_3$$

□

Lemma 15. *Let M be an LBA. Then a REV-1DFA(2) accepting VALC(M) can effectively be constructed.*

Now, we have all prerequisites to show that emptiness is non-semidecidable for REV-1DFA(2)s, where a problem is said to be *semidecidable* (see [5]) if the set of all instances for which the answer is "yes" is recursively enumerable.

Theorem 16. *Emptiness is not semidecidable for REV-1DFA(2)s.*

Proof. Let M be an LBA accepting inputs over the alphabet Σ. According to Lemma 15 we can effectively construct a REV-1DFA(2) M' accepting VALC(M). Clearly, $L(M') = \text{VALC}(M)$ is empty if and only if $L(M)$ is either empty or contains some words from the finite set $\{\lambda\} \cup \Sigma$. Since the word problem is decidable and emptiness is not semidecidable for LBAs, the theorem follows. □

The following technical lemma is used to show further undecidability results.

Lemma 17. *Let $k \geq 2$ and M be a REV-1DFA(k) with input alphabet A not containing a, b, c, and \$. Then $L'(M) = \{\, a^nb^nc^n\$w \mid n \geq 1 \text{ and } w \in L(M) \,\}$ is accepted by some REV-1DFA(k).*

Proof. A reversible REV-1DFA(k) M' for $L'(M)$ first checks similar to Example 2 that the input starts with a prefix $a^n b^n c^n\$$ for some $n \geq 1$. To this end, we use the states of Example 2 which we choose to be different from the states of M. The separating symbol \$ acts as right endmarker for the first part of the computation and as left endmarker for the simulation of M. A close inspection of the construction in Example 2 shows that the prefix $a^n b^n c^n$ is accepted when all heads are on the right endmarker and the state s_f is entered. Thus, if a prefix of the form $a^n b^n c^n\$$ has been detected in M', we know that state s_f is entered and all heads are on the symbol \$. Then, we start the simulation of M by adding a transition which simulates the first step of any computation of M. Formally, we add the transition $\delta'(s_f, \$, \dots, \$) = \delta(s_0, \triangleright, \dots, \triangleright)$. Then, M' can simulate M by using the states and transitions of M and by interpreting any \$ as $\triangleright$. Altogether, the input is accepted whenever M accepts. □

Theorem 18. *For $k \geq 2$, emptiness, finiteness, infiniteness, universality, inclusion, equivalence, regularity, and context-freeness are not semidecidable for REV-1DFA(k).*

Proof. The non-semidecidability of emptiness has been shown in Theorem 16. Since a REV-1DFA(k) that accepts nothing can effectively be constructed, the non-semidecidability of equivalence and inclusion follows immediately.

To show that finiteness is not semidecidable, we consider the language $L'(M)$ of Lemma 17 which can be constructed for an arbitrary REV-1DFA(k) M given. Clearly, $L'(M)$ is finite if and only if $L(M)$ is empty. This implies the assertion for finiteness.

For the next result we need that the language family $\mathscr{L}$(REV-1DFA(k)) is closed under complementation. A REV-1DFA(k) can enter an infinite loop only at the very beginning of the computation without moving the heads, that is, the initial state maps to itself when all heads read the left endmarker. In this case the empty language is accepted. Its complement is A^* which is clearly a REV-1DFA(k) language. In any other case it can be assumed that a REV-1DFA(k) halts on every input, either in an accepting or non-accepting state. Thus, closure under complementation is obtained by interchanging accepting and non-accepting states. So, infiniteness and universality are not semidecidable, since finiteness and emptiness are not.

For regularity and context-freeness we consider again language $L'(M)$ of Lemma 17. Clearly, $L'(M)$ is regular or context free if and only if $L(M)$ is empty. Thus, also non-semidecidability of regularity and context-freeness follows from the non-semidecidability of emptiness. □

It is shown in [11] that there cannot exist pumping lemmas or minimization algorithms for the computational model of *cellular automata*. The proofs rely on the fact that infiniteness and emptiness are not semidecidable for such automata. Similarly, we obtain that, for all $k \geq 2$, the family $\mathscr{L}$(REV-1DFA(k)) and each language family including $\mathscr{L}$(REV-1DFA(k)) do not possess a pumping lemma. Moreover, the following theorem can be shown as in [11].

Theorem 19. *For all $k \geq 2$, there is no minimization algorithm converting an arbitrary REV-1DFA(k) to an equivalent REV-1DFA(k) which has a minimal number of states.*

Finally, we show that reversibility of a given 1DFA(k) cannot be determined by inspecting its transitions. To this end, it is moreover necessary to have knowledge about the reachable configurations. However, the next theorem shows that this is impossible. We obtain that it is not semidecidable whether a given 1DFA(k) is reversible. It is worth mentioning, that when the prerequisite for reversibility to consider reachable configurations only is relaxed, then reversibility of a given 1DFA(k) is trivially decidable by inspecting the transitions.

Theorem 20. *For $k \geq 2$, it is not semidecidable whether a 1DFA(k) is reversible.*

Proof. Let $M = \langle S, A, k, \delta, \rhd, \lhd, s_0, F \rangle$ be an arbitrary REV-1DFA(k). By definition, M accepts an input if and only if it halts in an accepting state from F. We define the 1DFA(k) $M' = \langle S, A, k, \delta', \rhd, \lhd, s_0, F \rangle$ where δ' simulates the transitions of M, but has the following additional transitions. If $\delta(s, y_1, y_2, \ldots, y_k)$ is undefined for $s \in F$ and $y_i \in A \cup \{\lhd\}$, $1 \leq i \leq k$, then $\delta'(s, y_1, y_2, \ldots, y_k) = (s, 0, \ldots, 0)$ is added.

Clearly, M' is still reversible if $L(M)$ is empty, since any halting accepting state $s \in F$ is never entered. On the other hand, if $w \in L(M)$, then we know that some halting accepting state $s \in F$ and, by construction of M', an infinite loop is entered on input w. This implies that M' is irreversible, except for the case where M does nothing, that is, the transition function is totally undefined and the initial state is accepting. In this case, M' runs into an infinite loop from the very beginning. However, then M accepts A^* and, moreover, this situation can be detected by a simple inspection of the transition function. Thus, M' is reversible if and only if $L(M)$ is empty or M does nothing.

So, if reversibility is semidecidable for a 1DFA(k), then it is semidecidable whether a REV-1DFA(k) accepts the empty language or does nothing. Since doing nothing is decidable for a REV-1DFA(k), this implies that emptiness is semidecidable for a REV-1DFA(k), a contradiction. □

Acknowledgments. We like to thank the anonymous referees for their valuable comments which significantly improved the presentation of the paper.

References

1. Angluin, D.: Inference of reversible languages. J. ACM 29, 741–765 (1982)
2. Axelsen, H.B.: Reversible Multi-head Finite Automata Characterize Reversible Logarithmic Space. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 95–105. Springer, Heidelberg (2012)
3. Bennet, C.H.: Logical reversibility of computation. IBM J. Res. Dev. 17, 525–532 (1973)

4. Hartmanis, J.: On non-determinancy in simple computing devices. Acta Inform. 1, 336–344 (1972)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
6. Kutrib, M., Malcher, A.: Fast reversible language recognition using cellular automata. Inform. Comput. 206, 1142–1151 (2008)
7. Kutrib, M., Malcher, A.: Real-time reversible iterative arrays. Theoret. Comput. Sci. 411, 812–822 (2010)
8. Kutrib, M., Malcher, A.: Reversible pushdown automata. J. Comput. Syst. Sci. 78, 1814–1827 (2012)
9. Kutrib, M., Malcher, A., Wendlandt, M.: States and Heads Do Count for Unary Multi-head Finite Automata. In: Yen, H.-C., Ibarra, O.H. (eds.) DLT 2012. LNCS, vol. 7410, pp. 214–225. Springer, Heidelberg (2012)
10. Landauer, R.: Irreversibility and heat generation in the computing process. IBM J. Res. Dev. 5, 183–191 (1961)
11. Malcher, A.: Descriptional complexity of cellular automata and decidability questions. J. Autom., Lang. Comb. 7, 549–560 (2002)
12. Morita, K.: Reversible computing and cellular automata – A survey. Theoret. Comput. Sci. 395, 101–131 (2008)
13. Morita, K.: Two-way reversible multi-head finite automata. Fund. Inform. 110, 241–254 (2011)
14. Phillips, I.C.C., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. 73, 70–96 (2007)
15. Pin, J.E.: On Reversible Automata. In: Simon, I. (ed.) LATIN 1992. LNCS, vol. 583, pp. 401–416. Springer, Heidelberg (1992)
16. Rosenberg, A.L.: On multi-head finite automata. IBM J. Res. Dev. 10, 388–394 (1966)
17. Yao, A.C., Rivest, R.L.: $k+1$ heads are better than k. J. ACM 25, 337–340 (1978)
18. Yokoyama, T.: Reversible computation and reversible programming languages. Electron. Notes Theor. Comput. Sci. 253, 71–81 (2010)
19. 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, Part II. LNCS, vol. 5126, pp. 258–270. Springer, Heidelberg (2008)

A Deterministic Two-Way Multi-head Finite Automaton Can Be Converted into a Reversible One with the Same Number of Heads

Kenichi Morita

Department of Information Engineering, Hiroshima University
Higashi-Hiroshima, 739-8527, Japan

Abstract. A two-way multi-head finite automaton (MFA) is a variant of a finite automaton consisting of a finite-state control, a finite number of heads that can move in two directions, and a read-only input tape. Here, we show that for any given deterministic MFA we can construct a reversible MFA with the same number of heads that accepts the same language as the former. We then apply this conversion method to a Turing machine. By this, we can obtain, in a simple way, an equivalent reversible Turing machine that is garbage-less, uses the same number of tape symbols, and uses the same amount of the storage tape.

Keywords: multi-head finite automaton, reversible computing, reversible Turing machine, garbage-less computation.

1 Introduction

A multi-head finite automaton is a classical model for language recognition, and has relatively high recognition capability (see [4] for the survey). In [7], a reversible two-way multi-head finite automaton is introduced, and its basic properties are investigated. It is well known that the class of two-way deterministic multi-head finite automata is characterized by the complexity class of deterministic logarithmic space. Lange, McKenzie, and Tapp [6] showed that the class of deterministic space $S(n)$ is equal to the class of reversible space $S(n)$. Hence, the class of deterministic multi-head finite automata is characterized by the class of reversible space $\log n$. Later, Axelsen [1] showed that the class of reversible multi-head finite automata is also characterized by this complexity class.

In [7] it is conjectured that a stronger result holds, i.e., a deterministic two-way multi-head finite automaton can be simulated by a reversible one with the same number of heads. In this paper, we prove it by giving a concrete conversion method. The technique employed here is based on the method of Lange et al. [6] where a computation tree of a deterministic space-bounded Turing machine is traversed by a reversible one using the same amount of space. But, our method is simpler, and does not assume a simulated automaton always halts, and hence the converted reversible automaton traverses a computation graph that may not be a tree. We then apply our method to Turing machines, which may or may not be a space-bounded one, and does not necessarily halt. By this, we can obtain a garbage-less reversible Turing machine that uses the same amount of space and the same number of tape symbols as the simulated one.

2 A Two-Way Multi-Head Finite Automaton

Definition 1. *A* two-way multi-head finite automaton *(MFA) consists of a finite-state control, a finite number of heads that can move in two directions, and a read-only input tape (Fig. 1). An MFA with k heads is denoted by MFA(k).*

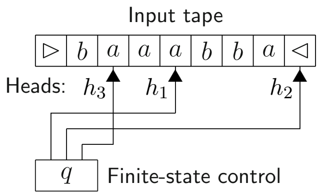


Fig. 1. A two-way multi-head finite automaton (MFA)

It is formally defined by

$$M = (Q, \Sigma, k, \delta, \triangleright, \triangleleft, q_0, A, R),$$

where Q is a nonempty finite set of states, Σ is a nonempty finite set of input symbols, $k\,(\in \{1, 2, \dots\})$ is a number of heads, $\triangleright$ and $\triangleleft$ are left and right endmarkers, respectively, which are not elements of Σ (i.e., $\{\triangleright, \triangleleft\} \cap \Sigma = \emptyset$), $q_0\,(\in Q)$ is the initial state, $A\,(\subset Q)$ is a set of accepting states, and $R\,(\subset Q)$ is a set of rejecting states such that $A \cap R = \emptyset$. δ is a subset of $(Q \times ((\Sigma \cup \{\triangleright, \triangleleft\})^k \cup \{-1, 0, +1\}^k) \times Q)$ that determines the transition relation on M's configurations (defined below). Note that $-1, 0,$ and $+1$ stand for left-shift, no-shift, and right-shift of each head, respectively. In what follows, we also use $-$ and $+$ instead of -1 and $+1$ for simplicity. Each element $r = [p, \mathbf{x}, q] \in \delta$ is called a rule *(in the triple form) of M, where $\mathbf{x} = [s_1, \dots, s_k] \in (\Sigma \cup \{\triangleright, \triangleleft\})^k$ or $\mathbf{x} = [d_1, \dots, d_k] \in \{-1, 0, +1\}^k$. A rule of the form $[p, [s_1, \dots, s_k], q]$ is called a* read-rule*, and means if M is in the state p and reads symbols $[s_1, \dots, s_k]$ by its k heads, then enter the state q. A rule of the form $[p, [d_1, \dots, d_k], q]$ is called a* shift-rule*, and means if M is in the state p then shift the heads to the directions $[d_1, \dots, d_k]$ and enter the state q.*

Suppose a word of the form $\triangleright w \triangleleft \in (\{\triangleright\}\Sigma^*\{\triangleleft\})$ is given to M. For any $q \in Q$ and for any $\mathbf{h} \in \{0, \dots, |w|+1\}^k$, a triple $[\triangleright w \triangleleft, q, \mathbf{h}]$ is called a *configuration* of M on w. We now define a function $s_w : \{0, \dots, |w|+1\}^k \to (\Sigma \cup \{\triangleright, \triangleleft\})^k$ as follows. If $\triangleright w \triangleleft = a_0 a_1 \cdots a_n a_{n+1}$ (hence $a_0 = \triangleright, a_{n+1} = \triangleleft$, and $w = a_1 \cdots a_n \in$

Σ^*), and $\mathbf{h} = [h_1, \ldots, h_k] \in \{0, \ldots, |w|+1\}^k$, then $s_w(\mathbf{h}) = [a_{h_1}, \ldots, a_{h_k}]$. The function s_w gives a k-tuple of symbols in $\rhd w \lhd$ read by the k heads of M at the position $\mathbf{h}$. The *transition relation* $\underset{M}{\vdash}$ between a pair of configurations $[\rhd w \lhd, q, \mathbf{h}]$ and $[\rhd w \lhd, q', \mathbf{h}']$ is defined as follows.

$$\begin{aligned}&[\rhd w \lhd, q, \mathbf{h}] \underset{M}{\vdash} [\rhd w \lhd, q', \mathbf{h}'] \\ &\quad \text{iff } ([q, s_w(\mathbf{h}), q'] \in \delta \ \wedge \ \mathbf{h}' = \mathbf{h}) \vee \\ &\qquad \exists\, \mathbf{d} \in \{-1, 0, +1\}^k \ ([q, \mathbf{d}, q'] \in \delta \ \wedge \ \mathbf{h}' = \mathbf{h} + \mathbf{d})\end{aligned}$$

The reflexive and transitive closure of the relation $\underset{M}{\vdash}$ is denoted by $\overset{*}{\underset{M}{\vdash}}$. Likewise, the m-step transition relation is denoted by $\overset{m}{\underset{M}{\vdash}}$ for each $m \in \{0, 1, \ldots\}$. When M is clear from the context, we use $\vdash$ instead of $\underset{M}{\vdash}$. A configuration c is called a *halting configuration* if there is no configuration c' such that $c \vdash c'$.

Definition 2. *Let $M = (Q, \Sigma, k, \delta, \rhd, \lhd, q_0, A, R)$ be an MFA. M is called* deterministic *iff the following condition holds.*

$$\begin{aligned}&\forall\, r_1 = [p, \mathbf{x}, q] \in \delta,\ \forall\, r_2 = [p', \mathbf{x}', q'] \in \delta \\ &\quad ((r_1 \neq r_2 \ \wedge \ p = p') \Rightarrow (\mathbf{x} \notin \{-1, 0, +1\}^k \wedge \mathbf{x}' \notin \{-1, 0, +1\}^k \wedge \mathbf{x} \neq \mathbf{x}'))\end{aligned}$$

It means that for any two distinct rules r_1 and r_2 in δ, if $p = p'$ then they are both read-rules and the k-tuples of symbols $\mathbf{x}$ and $\mathbf{x}'$ are different.

M is called reversible *iff the following condition holds.*

$$\begin{aligned}&\forall\, r_1 = [p, \mathbf{x}, q] \in \delta,\ \forall\, r_2 = [p', \mathbf{x}', q'] \in \delta \\ &\quad ((r_1 \neq r_2 \ \wedge \ q = q') \Rightarrow (\mathbf{x} \notin \{-1, 0, +1\}^k \wedge \mathbf{x}' \notin \{-1, 0, +1\}^k \wedge \mathbf{x} \neq \mathbf{x}'))\end{aligned}$$

It means that for any two distinct rules r_1 and r_2 in δ, if $q = q'$ then they are both read-rules and the k-tuples of symbols $\mathbf{x}$ and $\mathbf{x}'$ are different. The above is called the reversibility condition.

We denote a deterministic MFA (or MFA(k)) by DMFA (or DMFA(k)), and a reversible and deterministic MFA (or MFA(k)) by RDMFA (or RDMFA(k)). Note that, here, we do not consider nondeterministic MFAs.

In [7], an MFA(k) is defined in the quadruple form, because the total number of rules is generally smaller than the case of the triple form when designing an MFA. However, here we use the triple form to simplify the construction of an RDMFA that accepts the same language as that of a given DMFA, because the triple form is convenient to design an RDMFA that simulates a DMFA both in forward and backward directions, and switches from one direction to another very often. It is also used in [1]. Note that conversion between the quadruple form and the triple form is easy, and thus we omit to describe its method here.

Definition 3. *Let $M = (Q, \Sigma, k, \delta, \rhd, \lhd, q_0, A, R)$ be an MFA. We say an input word $w \in \Sigma^*$ is* accepted *by M, if $[\rhd w \lhd, q_0, \mathbf{0}] \overset{*}{\underset{M}{\vdash}} [\rhd w \lhd, q, \mathbf{h}]$ for some $q \in A$ and $\mathbf{h} \in \{0, \ldots, |w|+1\}^k$, where $\mathbf{0} = [0, \ldots, 0] \in \{0\}^k$. The configurations $[\rhd w \lhd, q_0, \mathbf{0}]$ and $[\rhd w \lhd, q, \mathbf{h}]$ such that $q \in A$ are called an* initial configuration *and an* accepting configuration*, respectively. The* language *accepted by M is the set of all words accepted by M, and is denoted by $L(M)$, i.e.,*

$$L(M) = \{w \mid \exists q \in A, \exists \mathbf{h} \in \{0, \ldots, |w|+1\}^k ([\rhd w \lhd, q_0, \mathbf{0}] \overset{*}{\underset{M}{\vdash}} [\rhd w \lhd, q, \mathbf{h}])\}.$$

In the following, we assume, without loss of generality, each MFA $M = (Q, \Sigma, k, \delta, \rhd, \lhd, q_0, A, R)$ satisfies the following conditions (M1) – (M5). They are for the later convenience of making RDMFAs.

(M1) The initial state q_0 does not appear as the third component of any rule in δ, i.e., in a backward computation q_0 is a halting state: $\forall [q, \mathbf{x}, q'] \in \delta \;\; (q' \neq q_0)$.

(M2) All the accepting and rejecting states do not appear as the first component of any rule, i.e., they are halting states: $\forall [q, \mathbf{x}, q'] \in \delta \;\; (q \notin A \cup R)$.

(M3) Every state other than the initial state appears as the third component of some rule in δ (otherwise such a state can be removed): $\forall q \in Q - \{q_0\},\ \exists q' \in Q,\ \exists \mathbf{x} \in (\Sigma \cup \{\rhd, \lhd\})^k \cup \{-1, 0, +1\}^k \;\; ([q', \mathbf{x}, q] \in \delta)$.

(M4) M performs read and shift operations alternately. Hence, Q is written as $Q = Q_{\text{read}} \cup Q_{\text{shift}}$ for some Q_{read} and Q_{shift} such that $Q_{\text{read}} \cap Q_{\text{shift}} = \emptyset$, and δ satisfies the following condition:

$$\begin{aligned} &\forall\ [p, \mathbf{x}, q] \in \delta\ (\mathbf{x} \in (\Sigma \cup \{\rhd, \lhd\})^k \Rightarrow p \in Q_{\text{read}} \wedge q \in Q_{\text{shift}}) \wedge \\ &\forall\ [p, \mathbf{x}, q] \in \delta\ (\mathbf{x} \in \{-, 0, +\}^k \Rightarrow p \in Q_{\text{shift}} \wedge q \in Q_{\text{read}}). \end{aligned}$$

It is easy to modify M so that it satisfies the above condition by adding new states to it. Each element of Q_{read} and Q_{shift} is called a *read-state* and a *shift-state*, respectively. We further assume $q_0 \in Q_{\text{read}}$, and $A \cup R \subset Q_{\text{shift}}$, though each state in $A \cup R$ makes no further move.

(M5) The heads of M must not go beyond the left and right endmarkers in a forward computation, and hence M does not go to an illegal configuration:

$$\begin{aligned} &\forall p, r \in Q_{\text{read}},\ \forall q \in Q_{\text{shift}}, \\ &\forall [s_1, \ldots, s_k] \in (\Sigma \cup \{\rhd, \lhd\})^k,\ \forall [d_1, \ldots, d_k] \in \{-, 0, +\}^k,\ \forall i \in \{1, \ldots, k\} \\ &\quad ([p, [s_1, \ldots, s_k], q], [q, [d_1, \ldots, d_k], r] \in \delta \\ &\qquad \Rightarrow\ (s_i = \rhd\ \Rightarrow\ d_i \in \{0, +\}) \wedge (s_i = \lhd\ \Rightarrow\ d_i \in \{-, 0\})). \end{aligned}$$

Likewise, M must satisfy a similar condition in a backward computation, and hence M does not come from an illegal configuration:

$$\begin{aligned} &\forall p, r \in Q_{\text{shift}},\ \forall q \in Q_{\text{read}}, \\ &\forall [s_1, \ldots, s_k] \in (\Sigma \cup \{\rhd, \lhd\})^k,\ \forall [d_1, \ldots, d_k] \in \{-, 0, +\}^k,\ \forall i \in \{1, \ldots, k\} \\ &\quad ([r, [d_1, \ldots, d_k], q], [q, [s_1, \ldots, s_k], p] \in \delta \\ &\qquad \Rightarrow\ (s_i = \rhd\ \Rightarrow\ d_i \in \{-, 0\}) \wedge (s_i = \lhd\ \Rightarrow\ d_i \in \{0, +\})). \end{aligned}$$

Lemma 1. [7] *Let $M = (Q, \Sigma, k, \delta, \rhd, \lhd, q_0, A, R)$ be an RDMFA. If M satisfies (M1), then it eventually halts for any input $w \in \Sigma^*$.*

3 Converting a DMFA(k) into an RDMFA(k)

In this section, we show that for any given DMFA(k) M we can construct an RDMFA(k) $M^\dagger$ that simulates M. The idea is based on that of Lange et al. [6], where a computation tree of a space-bounded Turing machine is traversed by a reversible one. Here, we make $M^\dagger$ so that it traverses a computation graph from a leaf node that corresponds to the initial configuration. Note that, if M halts on an input w, then the computation graph becomes a finite tree, since M is deterministic. But, if it loops and does not halt, then the graph is not a tree. We shall see that both cases are managed properly. If the graph is a tree, $M^\dagger$ visits all the nodes of the tree by the depth-first search, and finally halts when coming back to the initial configuration. If $M^\dagger$ finds an accepting configuration in the traversal, it memorizes the fact in the finite-state control, and gives an answer when it halts. On the other hand, if the graph is not a tree, $M^\dagger$ may not visit all the nodes, but we shall see it will always come back to the initial configuration, and thus the input is rejected.

Theorem 1. *For any DMFA(k) $M = (Q, \Sigma, k, \delta, \rhd, \lhd, q_0, A, R)$, we can construct an RDMFA(k) $M^\dagger = (Q^\dagger, \Sigma, k, \delta^\dagger, \rhd, \lhd, q_0, \{\hat{q}_0^1\}, \{q_0^1\})$ that satisfies the following.*

$$\begin{array}{l} \forall w \in \Sigma^* \ (w \in L(M) \ \Rightarrow \ [\rhd w \lhd, q_0, \mathbf{0}] \mid\!\frac{*}{M^\dagger} [\rhd w \lhd, \hat{q}_0^1, \mathbf{0}]) \\ \forall w \in \Sigma^* \ (w \notin L(M) \ \Rightarrow \ [\rhd w \lhd, q_0, \mathbf{0}] \mid\!\frac{*}{M^\dagger} [\rhd w \lhd, q_0^1, \mathbf{0}]) \end{array}$$

Proof. We first define the *computation graph* $G_{M,w} = (V, E)$ of M with an input $w \in \Sigma^*$ as follows. Let C be the set of all configurations of M with w, i.e., $C = \{[\rhd w \lhd, q, \mathbf{h}] \mid q \in Q \wedge \mathbf{h} \in \{0, \ldots, |w|+1\}^k\}$. The set $V (\subset C)$ of nodes is the smallest set that contains the initial configuration $[\rhd w \lhd, q_0, \mathbf{0}]$, and satisfies the following condition: $\forall c_1, c_2 \in C\,((c_1 \in V \ \wedge \ (c_1 \mid\!\frac{}{M} c_2 \ \vee \ c_2 \mid\!\frac{}{M} c_1)) \Rightarrow \ c_2 \in V)$. The set E of directed edges is: $E = \{(c_1, c_2) \mid c_1, c_2 \in V \ \wedge \ c_1 \mid\!\frac{}{M} c_2\}$. Apparently $G_{M,w}$ is a finite connected graph. Since M is deterministic, outdegree of each node in V is either 0 or 1, where a node of outdegree 0 corresponds to a halting configuration. It is easy to see there is at most one node of outdegree 0 in $G_{M,w}$, and if there is one, then $G_{M,w}$ is a tree (Fig. 2 (a)). In this case, we identify the node of outdegree 0 as its root. Therefore, all the nodes of indegree 0 are leaves. On the other hand, if there is no node of outdegree 0, then the graph represents the computation of M having a loop, and thus it is not a tree (Fig. 2 (b)).

To make $M^\dagger$ traverse a computation graph of M, we need some preparations for it. Let Q_{read} and Q_{shift} be the sets of states as described in (M4). First, we define the following five functions: prev-read : $Q_{\text{read}} \to 2^{Q_{\text{shift}} \times \{-,0,+\}^k}$, prev-shift : $Q_{\text{shift}} \times (\Sigma \cup \{\rhd, \lhd\})^k \to 2^{Q_{\text{read}}}$, $\deg_{\text{r}} : Q_{\text{read}} \to \mathbb{N}$, $\deg_{\text{s}} : Q_{\text{shift}} \times (\Sigma \cup \{\rhd, \lhd\})^k \to \mathbb{N}$, and $\deg_{\max} : Q \to \mathbb{N}$ as follows.

$$\begin{aligned} \text{prev-read}(q) &= \{[p, \mathbf{d}] \mid p \in Q_{\text{shift}} \wedge \mathbf{d} \in \{-, 0, +\}^k \wedge [p, \mathbf{d}, q] \in \delta\} \\ \text{prev-shift}(q, \mathbf{s}) &= \{p \mid p \in Q_{\text{read}} \wedge [p, \mathbf{s}, q] \in \delta\} \\ \deg_{\text{r}}(q) &= |\text{prev-read}(q)| \\ \deg_{\text{s}}(q, \mathbf{s}) &= |\text{prev-shift}(q, \mathbf{s})| \\ \deg_{\max}(q) &= \begin{cases} \deg_{\text{r}}(q) & \text{if } q \in Q_{\text{read}} \\ \max\{\deg_{\text{s}}(q, \mathbf{s}) \mid \mathbf{s} \in (\Sigma \cup \{\rhd, \lhd\})^k\} & \text{if } q \in Q_{\text{shift}} \end{cases} \end{aligned}$$

Assume M is in the configuration $[\rhd w \lhd, q, \mathbf{h}]$. If q is a read-state (shift-state, respectively), then $\deg_{\text{r}}(q)$ ($\deg_{\text{s}}(q, s_w(\mathbf{h}))$) denotes the total number of previous configurations of $[\rhd w \lhd, q, \mathbf{h}]$, and each element $[p, \mathbf{d}] \in$ prev-read(q)

($p \in$ prev-shift$(q, s_w(\mathbf{h}))$) gives a previous state and a shift direction (a previous state). We further assume that the set Q and, of course, the set $\{-1, 0, +1\}$ are totally ordered, and thus the elements of the sets prev-read(q) and prev-shift(q, s) are sorted based on these orders. So, in the following, we denote prev-read(q) and prev-shift(q, s) by the ordered lists as below.

$$\begin{aligned}\text{prev-read}(q) &= [[p_1, \mathbf{d}_1], \ldots, [p_{\deg_r(q)}, \mathbf{d}_{\deg_r(q)}]]\\ \text{prev-shift}(q, \mathbf{s}) &= [p_1, \ldots, p_{\deg_s(q,\mathbf{s})}]\end{aligned}$$

We now construct an RDMFA(k) $M^\dagger$ that simulates the DMFA(k) M by traversing $G_{M,w}$ for a given w. $Q^\dagger$ and $\delta^\dagger$ of $M^\dagger$ are as below, where $\mathbf{S} = (\Sigma \cup \{\rhd, \lhd\})^k$.

$$\begin{aligned}
&Q^\dagger = \{q, \hat{q} \mid q \in Q\} \cup \{q^j, \hat{q}^j \mid q \in Q \wedge j \in (\{1\} \cup \{1, \ldots, \deg_{\max}(q)\})\}\\
&\delta^\dagger = \ \delta_1 \cup \cdots \cup \delta_6 \cup \hat{\delta}_1 \cup \cdots \cup \hat{\delta}_5 \cup \delta_a \cup \delta_r\\
&\delta_1 = \{\, [p_1, \mathbf{d}_1, q^2], \ldots, [p_{\deg_r(q)-1}, \mathbf{d}_{\deg_r(q)-1}, q^{\deg_r(q)}], [p_{\deg_r(q)}, \mathbf{d}_{\deg_r(q)}, q] \mid\\
&\qquad q \in Q_{\text{read}} \wedge \deg_r(q) \geq 1 \wedge \text{prev-read}(q) = [[p_1, \mathbf{d}_1], \ldots, [p_{\deg_r(q)}, \mathbf{d}_{\deg_r(q)}]] \,\}\\
&\delta_2 = \{\, [p_1, \mathbf{s}, q^2], \ldots, [p_{\deg_s(q,\mathbf{s})-1}, \mathbf{s}, q^{\deg_s(q,\mathbf{s})}], [p_{\deg_s(q,\mathbf{s})}, \mathbf{s}, q] \mid\\
&\qquad q \in Q_{\text{shift}} \wedge \mathbf{s} \in \mathbf{S} \wedge \deg_s(q, \mathbf{s}) \geq 1 \wedge \text{prev-shift}(q, \mathbf{s}) = [p_1, \ldots, p_{\deg_s(q,\mathbf{s})}] \,\}\\
&\delta_3 = \{\, [q^1, -\mathbf{d}_1, p_1^1], \ldots, [q^{\deg_r(q)}, -\mathbf{d}_{\deg_r(q)}, p^1_{\deg_r(q)}] \mid\\
&\qquad q \in Q_{\text{read}} \wedge \deg_r(q) \geq 1 \wedge \text{prev-read}(q) = [[p_1, \mathbf{d}_1], \ldots, [p_{\deg_r(q)}, \mathbf{d}_{\deg_r(q)}]] \,\}\\
&\delta_4 = \{\, [q^1, \mathbf{s}, p_1^1], \ldots, [q^{\deg_s(q,\mathbf{s})}, \mathbf{s}, p^1_{\deg_s(q,\mathbf{s})}] \mid\\
&\qquad q \in Q_{\text{shift}} \wedge \mathbf{s} \in \mathbf{S} \wedge \deg_s(q, \mathbf{s}) \geq 1 \wedge \text{prev-shift}(q, \mathbf{s}) = [p_1, \ldots, p_{\deg_s(q,\mathbf{s})}] \,\}\\
&\delta_5 = \{\, [q^1, \mathbf{s}, q] \mid q \in Q_{\text{shift}} - (A \cup R) \wedge \mathbf{s} \in \mathbf{S} \wedge \deg_s(q, \mathbf{s}) = 0 \,\}\\
&\hat{\delta}_i = \{\, [\hat{p}, \mathbf{x}, \hat{q}] \mid [p, \mathbf{x}, q] \in \delta_i \,\} \ (i = 1, \ldots, 5)\\
&\delta_6 = \{\, [q, \mathbf{s}, q^1] \mid q \in Q_{\text{read}} - \{q_0\} \wedge \mathbf{s} \in \mathbf{S} \wedge \neg\exists p\, ([q, \mathbf{s}, p] \in \delta) \,\}\\
&\delta_a = \{\, [q, \mathbf{0}, \hat{q}^1] \mid q \in A \,\}\\
&\delta_r = \{\, [q, \mathbf{0}, q^1] \mid q \in R \,\}
\end{aligned}$$

The set of states $Q^\dagger$ has four types of states. They are of the forms $q, \hat{q}, q^j$ and $\hat{q}^j$. The states without a superscript (i.e., q and $\hat{q}$) are for forward computation, while those with a superscript (i.e., q^j and $\hat{q}^j$) are for backward computation. Note that $Q^\dagger$ contains q^1 and $\hat{q}^1$ even if $\deg_{\max}(q) = 0$. The states with "ˆ" (i.e., $\hat{q}$ and $\hat{q}^j$) are the ones indicating that an accepting configuration was found in the process of traverse, while those without "ˆ" (i.e., q and q^j) are for indicating no accepting configuration has been found so far.

The set of rules δ_1 (δ_2, respectively) is for simulating forward computation of M in $G_{M,w}$ for M's shift-states (read-states). δ_3 (δ_4, respectively) is for simulating backward computation of M in $G_{M,w}$ for M's read-states (shift-states). δ_5 is for turning the direction of computation from backward to forward in $G_{M,w}$ for shift-states. $\hat{\delta}_i$ ($i = 1, \ldots, 5$) is the set of rules for the states of the form $\hat{q}$, and is identical to δ_i except that each state has "ˆ". δ_6 is for turning the direction of computation from forward to backward in $G_{M,w}$ for halting configurations with a read-state. δ_a (δ_r, respectively) is for turning the direction of computation from forward to backward for accepting (rejecting) states. In addition, each rule in δ_a makes $M^\dagger$ change the state from a one without "ˆ" to the corresponding one with "ˆ". We can verify that $M^\dagger$ is deterministic and reversible by a careful inspection of $\delta^\dagger$. We can also see that $M^\dagger$ satisfies the conditions (M1) – (M5).

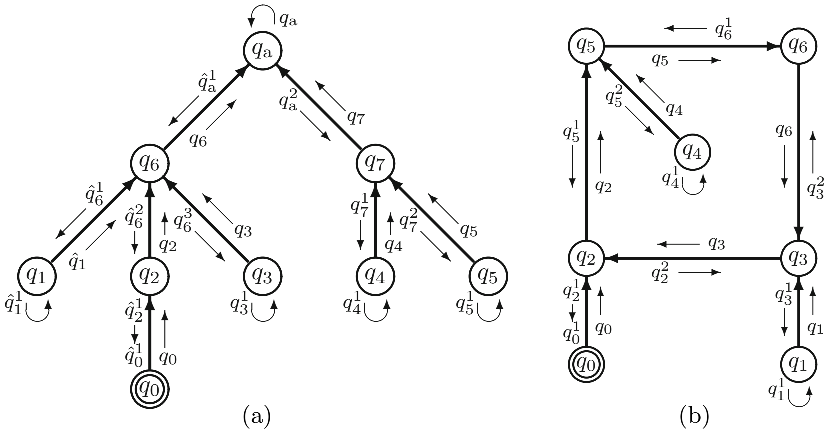


Fig. 2. Examples of computation graphs $G_{M,w}$ of a DMFA(k) M. Each node represents a configuration of M, though only a state of the finite-state control is written in a circle. Thick arrows are the edges of $G_{M,w}$. The node labeled by q_0 represents the initial configuration of M. An RDMFA(k) $M^\dagger$ traverses these graphs along thin arrows using its configurations. (a) This is a case M halts in an accepting state $q_{\rm a}$. Here, the state transition of $M^\dagger$ in the traversal of the tree is as follows: $q_0 \to q_2 \to q_6^3 \to q_3^1 \to q_3 \to q_6 \to q_{\rm a}^2 \to q_7^1 \to q_4^1 \to q_4 \to q_7^2 \to q_5^1 \to q_5 \to q_7 \to q_{\rm a} \to \hat{q}_{\rm a}^1 \to \hat{q}_6^1 \to \hat{q}_1^1 \to \hat{q}_1 \to \hat{q}_6^2 \to \hat{q}_2^1 \to \hat{q}_0^1$. (b) This is a case M loops forever. Here, $M^\dagger$ traverses the graph as follows: $q_0 \to q_2^2 \to q_3^1 \to q_1^1 \to q_1 \to q_3^2 \to q_6^1 \to q_5^1 \to q_2^1 \to q_0^1$.

$M^\dagger$ simulates M as follows. First, consider the case $G_{M,w}$ is a tree. If an input w is given, $M^\dagger$ traverses $G_{M,w}$ by the depth-first search (Fig. 2 (a)). Note that the search starts not from the root of the tree but from the leaf node $[\rhd w \lhd, q_0, \mathbf{0}]$. Since each node of $G_{M,w}$ is identified by the configuration of M of the form $[\rhd w \lhd, q, \mathbf{h}]$, it is easy for $M^\dagger$ to keep it by the configuration of $M^\dagger$. But, if $[\rhd w \lhd, q, \mathbf{h}]$ is a non-leaf node, it may be visited $\deg_{\max}(q)+1$ times (i.e., the number of its child nodes plus 1) in the process of depth-first search, and thus $M^\dagger$ should keep this information in the finite state control. To do so, $M^\dagger$ uses $\deg_{\max}(q)+1$ states $q^1, \ldots, q^{\deg_{\max}(q)}$, and q for each state q of M. Here, the states $q^1, \ldots, q^{\deg_{\max}(q)}$ are used for visiting its child nodes, and q is used for visiting its parent node. In other words, the states with a superscript are for going downward in the tree (i.e., a backward simulation of M), and the state without a superscript is for going upward in the tree (i.e., a forward simulation). At a leaf node $[\rhd w \lhd, q, \mathbf{h}]$, which satisfies $\deg_{\rm s}(q, s_w(\mathbf{h})) = 0$, $M^\dagger$ turns the direction of computing by the rule $[q^1, s_w(\mathbf{h}), q] \in \delta_5$.

If $M^\dagger$ enters an accepting state of M, say $q_{\rm a}$, which is the root of the tree while traversing the tree, then $M^\dagger$ goes to the state $\hat{q}_{\rm a}$, and continues the depth-first search. After that, $M^\dagger$ uses the states of the form $\hat{q}$ and $\hat{q}^j$ indicating that the input w should be accepted. $M^\dagger$ will eventually reach the initial configuration of M by its configuration $[\rhd w \lhd, \hat{q}_0^1, \mathbf{0}]$. Thus, $M^\dagger$ halts and accepts the input. Note that we can assume there is no rule of the form $[q_0, \mathbf{s}, q]$ such that $\mathbf{s} \notin \{\rhd\}^k$ in

δ, because the initial configuration of M is $[\rhd w \lhd, q_0, \mathbf{0}]$, and (M1) is assumed. Therefore, $M^\dagger$ never reaches a configuration $[\rhd w \lhd, q_0, \mathbf{h}]$ of M such that $\mathbf{h} \neq \mathbf{0}$.

If $M^\dagger$ enters a halting state of M other than the accepting states, then it continues the depth-first search without entering a state of the form $\hat{q}$. Also in this case, $M^\dagger$ will finally reach the initial configuration of M by its configuration $[\rhd w \lhd, q_0^1, \mathbf{0}]$. Thus, $M^\dagger$ halts and rejects the input.

Second, consider the case $G_{M,w}$ is not a tree (Fig. 2 (b)). In this case, since there is no accepting configuration in $G_{M,w}$, $M^\dagger$ never enters an accepting state of M no matter how $M^\dagger$ visits the nodes of $G_{M,w}$ (it may not visit all the nodes of $G_{M,w}$). Thus, $M^\dagger$ uses only the states without "$\hat{\ }$". From $\delta^\dagger$ we can see q_0^1 is the only halting state without "$\hat{\ }$". Since M satisfies the condition (M1), $M^\dagger$ halts with the configuration $[\rhd w \lhd, q_0^1, \mathbf{0}]$ by Lemma 1, and rejects the input.

By above, we have the following relations.

$$\forall w \in \Sigma^*(w \in L(M) \Rightarrow ([\rhd w \lhd, q_0, \mathbf{0}] \mathrel{\mid\!\!\!\frac{\;*\;}{M^\dagger}} [\rhd w \lhd, \hat{q}_0^1, \mathbf{0}]))$$
$$\forall w \in \Sigma^*(w \notin L(M) \Rightarrow ([\rhd w \lhd, q_0, \mathbf{0}] \mathrel{\mid\!\!\!\frac{\;*\;}{M^\dagger}} [\rhd w \lhd, q_0^1, \mathbf{0}]))$$

Thus, $L(M^\dagger) = L(M)$. We can also see $M^\dagger$ is "garbage-less" in the sense it always halts with all the heads at the left endmarker. □

Example 1. Let $L_\mathrm{p} = \{w \mid \text{the length of } w \in \{1\}^* \text{ is a prime number}\}$. The following irreversible DMFA(3) M_p accepts L_p.

$$\begin{aligned}
M_\mathrm{p} &= (Q, \{1\}, 3, \delta, \rhd, \lhd, q_0, \{q_\mathrm{a}\}, \{\ \}) \\
Q &= \{q_0, q_1, \ldots, q_{16}, q_\mathrm{a}\} \\
\delta &= \{\, [q_0, [\rhd, \rhd, \rhd], q_1],\ [q_1, [0, +, 0], q_2],\ [q_2, [\rhd, 1, \rhd], q_3],\ [q_3, [0, +, 0], q_4], \\
&\quad [q_4, [\rhd, 1, \rhd], q_5],\ [q_5, [+, -, +], q_6],\ [q_6, [1, 1, 1], q_5],\ [q_6, [1, \rhd, 1], q_7], \\
&\quad [q_6, [\lhd, 1, 1], q_9],\ [q_6, [\lhd, \rhd, 1], q_9],\ [q_7, [0, +, -], q_8],\ [q_8, [1, 1, 1], q_7], \\
&\quad [q_8, [1, 1, \rhd], q_5],\ [q_9, [0, +, -], q_{10}],\ [q_{10}, [\lhd, 1, \rhd], q_{14}],\ [q_{10}, [\lhd, 1, 1], q_{11}], \\
&\quad [q_{11}, [-, +, -], q_{13}],\ [q_{12}, [-, 0, 0], q_{13}],\ [q_{13}, [1, 1, 1], q_{11}],\ [q_{13}, [1, 1, \rhd], q_{12}], \\
&\quad [q_{13}, [\rhd, 1, \rhd], q_3],\ [q_{14}, [0, +, 0], q_{15}],\ [q_{15}, [\lhd, \lhd, \rhd], q_\mathrm{a}],\ [q_{15}, [\lhd, 1, \rhd], q_{16}], \\
&\quad [q_{16}, [0, -, 0], q_{10}]\ \}
\end{aligned}$$

If a word $1^n \in \{1\}^*$ $(n = 2, 3, \ldots)$ is given, M_p divides n by $m = 2, 3, \ldots, n$ successively. If $m = n$ is the least integer among them that divides n, then M_p accepts 1^n. If n is not a prime, M_p halts in a non-accepting state (if $n = 0$ or $n = 1$), or loops forever (if $n > 1$). The first head is used as a counter for carrying out a division of n by m. The second head is a counter for keeping the divisor m. The third head is an auxiliary counter for restoring m. The states q_0, q_1, and q_2 are used for initializing the algorithm. $q_3, \ldots, q_8$ are for dividing n by m. $q_9, \ldots, q_{13}$ are for checking if n is divisible by m. $q_{14}, \ldots, q_{17}$ are for checking if $m = n$. Note that M_p is irreversible, since e.g., the pair $[q_{11}, [-, +, -], q_{13}]$ and $[q_{12}, [-, 0, 0], q_{13}]$ violates the reversibility condition. Examples of computing processes of M_p are as below.

$$[\ \rhd 1111111 \lhd,\ q_0,\ [0,0,0]\ ] \mathrel{\mid\!\!\!\frac{\;273\;}{M_\mathrm{p}}} [\ \rhd 1111111 \lhd,\ q_\mathrm{a},\ [8,8,0]\ ]$$
$$\begin{aligned}
[\ \rhd 111111 \lhd,\ q_0,\ [0,0,0]\ ] &\mathrel{\mid\!\!\!\frac{\;32\;}{M_\mathrm{p}}} [\ \rhd 111111 \lhd,\ q_{10},\ [7,2,0]\ ] \\
&\mathrel{\mid\!\!\!\frac{\;4\;}{M_\mathrm{p}}} [\ \rhd 111111 \lhd,\ q_{10},\ [7,2,0]\ ] \mathrel{\mid\!\!\!\frac{\;\;}{M_\mathrm{p}}} \cdots
\end{aligned}$$

An RDMFA(3) $M_{\mathrm{p}}^{\dagger}$ that simulates M_{p} obtained by the method in Theorem 1 is:

$$\begin{aligned}
M_{\mathrm{p}}^{\dagger} &= (Q^{\dagger}, \{1\}, 3, \delta^{\dagger}, \rhd, \lhd, q_0, \{\hat{q}_0^1\}, \{q_0^1\}) \\
Q^{\dagger} &= \{q, \hat{q}, q^1, \hat{q}^1 \mid q \in Q\} \cup \{\, q_3^2, q_{10}^2, q_{13}^2, \hat{q}_3^2, \hat{q}_{10}^2, \hat{q}_{13}^2 \,\} \\
\delta^{\dagger} &= \delta_1 \cup \cdots \cup \delta_6 \cup \hat{\delta}_1 \cup \cdots \cup \hat{\delta}_5 \cup \delta_{\mathrm{a}} \cup \delta_{\mathrm{r}} \\
\delta_1 &= \{\, [q_1, [0,+,0], q_2],\ [q_3, [0,+,0], q_4],\ [q_5, [+,-,+], q_6],\ [q_7, [0,+,-], q_8], \\
&\quad [q_9, [0,+,-], q_{10}^2],\ [q_{11}, [-,+,-], q_{13}^2],\ [q_{12}, [-,0,0], q_{13}],\ [q_{14}, [0,+,0], q_{15}], \\
&\quad [q_{16}, [0,-,0], q_{10}] \,\} \\
\delta_2 &= \{\, [q_0, [\rhd,\rhd,\rhd], q_1],\ [q_2, [\rhd,1,\rhd], q_3^2],\ [q_4, [\rhd,1,\rhd], q_5],\ [q_6, [1,1,1], q_5], \\
&\quad [q_6, [1,\rhd,1], q_7],\ [q_6, [\lhd,1,1], q_9],\ [q_6, [\lhd,\rhd,1], q_9],\ [q_8, [1,1,1], q_7], \\
&\quad [q_8, [1,1,\rhd], q_5],\ [q_{10}, [\lhd,1,\rhd], q_{14}],\ [q_{10}, [\lhd,1,1], q_{11}],\ [q_{13}, [1,1,1], q_{11}], \\
&\quad [q_{13}, [1,1,\rhd], q_{12}],\ [q_{13}, [\rhd,1,\rhd], q_3],\ [q_{15}, [\lhd,\lhd,\rhd], q_{\mathrm{a}}],\ [q_{15}, [\lhd,1,\rhd], q_{16}] \,\} \\
\delta_3 &= \{\, [q_2^1, [0,-,0], q_1^1],\ [q_4^1, [0,-,0], q_3^1],\ [q_6^1, [-,+,-], q_5^1],\ [q_8^1, [0,-,+], q_7^1], \\
&\quad [q_{10}^1, [0,-,+], q_9^1],\ [q_{10}^2, [0,+,0], q_{16}^1],\ [q_{13}^1, [+,-,+], q_{11}^1],\ [q_{13}^2, [+,0,0], q_{12}^1], \\
&\quad [q_{15}^1, [0,-,0], q_{14}^1] \,\} \\
\delta_4 &= \{\, [q_1^1, [\rhd,\rhd,\rhd], q_0^1],\ [q_3^1, [\rhd,1,\rhd], q_2^1],\ [q_3^2, [\rhd,1,\rhd], q_{13}^1],\ [q_5^1, [\rhd,1,\rhd], q_4^1], \\
&\quad [q_5^1, [1,1,1], q_6^1],\ [q_5^1, [1,1,\rhd], q_8^1],\ [q_7^1, [1,\rhd,1], q_6^1],\ [q_7^1, [1,1,1], q_8^1], \\
&\quad [q_9^1, [\lhd,1,1], q_6^1],\ [q_9^1, [\lhd,\rhd,1], q_6^1],\ [q_{11}^1, [\lhd,1,1], q_{10}^1],\ [q_{11}^1, [1,1,1], q_{13}^1], \\
&\quad [q_{12}^1, [1,1,\rhd], q_{13}^1],\ [q_{14}^1, [\lhd,1,\rhd], q_{10}^1],\ [q_{16}^1, [\lhd,1,\rhd], q_{15}^1],\ [q_{\mathrm{a}}^1, [\lhd,\lhd,\rhd], q_{15}^1] \,\} \\
\delta_5 &= \{\, [q_1^1, [\rhd,\rhd,1], q_1],\ [q_1^1, [\rhd,\rhd,\lhd], q_1],\ \ldots\ ,\ [q_{16}^1, [\lhd,\lhd,\lhd], q_{16}] \,\} \\
\hat{\delta}_i &= \{\, [\hat{p}, \mathbf{x}, \hat{q}] \mid [p, \mathbf{x}, q] \in \delta_i \,\}\ (i = 1, \ldots, 5) \\
\delta_6 &= \{\, [q_2, [\rhd,\rhd,\rhd], q_2^1],\ [q_2, [\rhd,\rhd,1], q_2^1],\ \ldots\ ,\ [q_{15}, [\lhd,\lhd,\lhd], q_{15}^1] \,\} \\
\delta_{\mathrm{a}} &= \{\, [q_{\mathrm{a}}, [0,0,0], \hat{q}_{\mathrm{a}}^1] \,\} \\
\delta_{\mathrm{r}} &= \{\,\}
\end{aligned}$$

The details of δ_5 and δ_6 are omitted here since they have 228 and 174 rules, respectively. Examples of computing processes of $M_{\mathrm{p}}^{\dagger}$ are as follows.

$$[\ \rhd 1111111 \lhd,\ q_0,\ [0,0,0]\ ] \mid\!\!\frac{2016}{M_{\mathrm{p}}^{\dagger}}\ [\ \rhd 1111111 \lhd,\ \hat{q}_0^1,\ [0,0,0]\ ]$$
$$[\ \rhd 111111 \lhd,\ q_0,\ [0,0,0]\ ] \mid\!\!\frac{118}{M_{\mathrm{p}}^{\dagger}}\ [\ \rhd 111111 \lhd,\ q_0^1,\ [0,0,0]\ ]$$

□

Note that Theorem 1 generalizes the result by Kondacs and Watrous [5] that a deterministic finite automaton can be simulated by a reversible two-way finite automaton.

Sipser [8] showed that a deterministic space-bounded Turing machine and a DMFA can be converted to equivalent deterministic machines that always halt. In his method, the constructed machine traverses a computation graph of the simulated machine from the node corresponding to its *accepting configuration*. Therefore, the graph always becomes a tree, and by this the halting property is guaranteed. The method of Lange et al. [6] is based on this idea. On the other hand, in our method, the computation graph of a simulated DMFA is traversed from the node corresponding to its *initial configuration*, and thus the graph may not be a tree. However, since Lemma 1 holds, it is automatically guaranteed that the constructed RDMFA halts even if it is not a tree. Thus, it is convenient to use this method, because we only need the initial configuration of the simulated machine, and there is no need to know how the accepting configuration is. This is particularly useful for simplifying the method of converting a deterministic Turing machine to an equivalent reversible one. It is shown in the next section.

4 Applying the Conversion Method to Turing Machines

Here we apply the method of the previous section for converting a deterministic Turing machine to a reversible one. The resulting machine is garbage-less, uses the same number of tape symbols, and also uses the same amount of a storage tape under the assumption that the original machine does not rewrite a non-blank symbol into a blank one. This shows another simple method of constructing a garbage-less reversible Turing machine other than the method of Bennett [2].

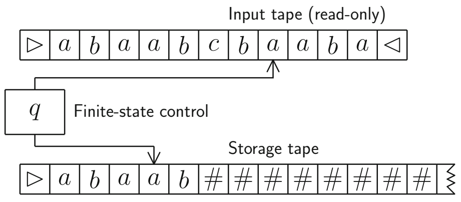


Fig. 3. A two-tape Turing machine

Definition 4. *A* two-tape Turing machine *(TM) as an acceptor of a language consists of a finite-state control with two heads, a read-only input tape, and a storage tape (Fig. 3) . It is defined by*

$$T = (Q, \Sigma, \Gamma, \delta, \rhd, \lhd, q_0, \#, A, R),$$

where Q is a nonempty finite set of states, Σ and Γ are nonempty finite sets of input symbols and storage tape symbols. $\rhd$ and $\lhd$ are left and right endmarkers such that $\{\rhd, \lhd\} \cap (\Sigma \cup \Gamma) = \emptyset$, where only $\rhd$ is used for the storage tape. q_0 $(\in Q)$ is the initial state, $\#$ $(\notin \Gamma)$ is a blank symbol of the storage tape, A $(\subset Q)$ and R $(\subset Q)$ are sets of accepting and rejecting states such that $A \cap R = \emptyset$. δ is a subset of $(Q \times (((\Sigma \cup \{\rhd, \lhd\}) \times (\Gamma \cup \{\rhd, \#\})^2) \cup \{-1, 0, +1\}^2) \times Q)$ that determines the transition relation on T's configurations. Each element $r = [p, x, y, q] \in \delta$ is called a rule *(in the quadruple form) of T, where $(x, y) = (s_1, [s_2, s_3]) \in ((\Sigma \cup \{\rhd, \lhd\}) \times (\Gamma \cup \{\rhd, \#\})^2)$ or $(x, y) = (d_1, d_2) \in \{-1, 0, +1\}^2$. A rule of the form $[p, s_1, [s_2, s_3], q]$ is called a* read-write-rule*, and means if T is in the state p and reads an input symbol s_1 and a storage tape symbol s_2, then rewrites s_2 to s_3 and enters the state q. Note that the left endmarker $\rhd$ of the storage tape should not be rewritten to any other symbol. A rule of the form $[p, d_1, d_2, q]$ is called a* shift-rule*, and means if T is in the state p then shift the two heads to the directions d_1 and d_2, and enter the state q.*

Suppose an input of the form $\rhd w \lhd \in (\{\rhd\}\Sigma^*\{\lhd\})$ is given to T. Let $q \in Q$, $v \in \Gamma^*$, $h_{\mathrm{i}} \in \{0, \ldots, |w|+1\}$, and $h_{\mathrm{s}} \in \{0, 1, \ldots\}$. A quintuple $[\rhd w \lhd, \rhd v, q, h_{\mathrm{i}}, h_{\mathrm{s}}]$ is called a *configuration* of T over w, where $\rhd v$ is the non-blank part of the storage tape, and h_{i} and h_{s} are the positions of the input and the storage tape heads. The *transition relation* $\mathrel{\vdash_{\overline{T}}}$ between a pair of configurations is defined similarly to the case of MFA, and hence we omit its details here. We say an input w is *accepted* by T if $[\rhd w \lhd, \rhd, q_0, 0, 0] \mathrel{\vdash_{\overline{T}}^{*}} [\rhd w \lhd, \rhd v, q, h_{\mathrm{i}}, h_{\mathrm{s}}]$ for some $v \in \Gamma^*$, $q \in A$, $h_{\mathrm{i}} \in \{0, \ldots, |w|+1\}$, and $h_{\mathrm{s}} \in \{0, 1, \ldots\}$.

Determinism and reversibility of T are defined as follows. We denote a deterministic TM by DTM, and a reversible and deterministic TM by RDTM. A TM T is called *deterministic* iff the following condition holds.

$$\begin{aligned}&\forall\, r_1 = [p, x, y, q] \in \delta,\ \forall\, r_2 = [p', x', y', q'] \in \delta\\ &\quad ((r_1 \neq r_2\ \wedge\ p = p') \Rightarrow\ ((x, y) \notin \{-, 0, +\}^2 \wedge (x', y') \notin \{-, 0, +\}^2 \wedge\\ &\quad \forall s_2, s_3, s_2', s_3' \in \Gamma \cup \{\rhd, \#\}(y = (s_2, s_3) \wedge y' = (s_2', s_3')\ \Rightarrow\ (x, s_2) \neq (x', s_2'))))\end{aligned}$$

T is called *reversible* iff the following condition holds.

$$\begin{aligned}&\forall\, r_1 = [p, x, y, q] \in \delta,\ \forall\, r_2 = [p', x', y', q'] \in \delta\\ &\quad ((r_1 \neq r_2\ \wedge\ q = q') \Rightarrow\ ((x, y) \notin \{-, 0, +\}^2 \wedge (x', y') \notin \{-, 0, +\}^2 \wedge\\ &\quad \forall s_2, s_3, s_2', s_3' \in \Gamma \cup \{\rhd, \#\}(y = (s_2, s_3) \wedge y' = (s_2', s_3')\ \Rightarrow\ (x, s_3) \neq (x', s_3'))))\end{aligned}$$

Let (T1) – (T5) be the conditions for TMs corresponding to (M1) – (M5), and we assume a given TM satisfies them. Here, we omit their details, but note that Q_{rw} (the set of states for read-write) and Q_{s} (the set of states for shifting heads) are defined similarly in (M4). Also note that (M5) should be modified so that both input and storage tape heads do not go beyond the endmarkers. We also assume, without loss of generality, a given DTM, which is to be converted to an RDTM, satisfies the condition (T6) (but, the resulting RDTM will not).

(T6) Let T be a DTM. It does not erase a non-blank symbol on its storage tape, and it does not read a blank symbol other than the leftmost one in each configuration (if otherwise, it can be easily achieved by adding a new non-blank symbol, say $\#'$, to T, which plays a role of the blank symbol $\#$):

$$\begin{aligned}&\forall p, q \in Q,\ \forall s_1 \in \Sigma \cup \{\rhd, \lhd\},\ \forall s_2, s_3 \in \Gamma \cup \{\rhd, \#\}\\ &\quad ([p, s_1, [s_2, s_3], q] \in \delta\ \Rightarrow\ (s_2 \neq \#\ \Rightarrow\ s_3 \neq \#)),\\ &\forall p, q, r \in Q,\ \forall s_1 \in \Sigma \cup \{\rhd, \lhd\},\ \forall s_2, s_3 \in \Gamma \cup \{\rhd, \#\},\ \forall d_1, d_2 \in \{-, 0, +\}^k\\ &\quad ([p, s_1, [s_2, s_3], q], [q, d_1, d_2, r] \in \delta\ \Rightarrow\ (s_2 = s_3 = \#\ \Rightarrow\ d_2 \in \{-, 0\})).\end{aligned}$$

We say a DTM T with $w \in \Sigma^*$ uses *bounded amount of the storage tape* if the following holds.

$$\begin{aligned}&\exists m \in \{0, 1, \ldots\}, \forall v \in \Gamma^*, \forall h_{\mathrm{i}} \in \{0, \ldots, |w|+1\}, \forall h_{\mathrm{s}} \in \{0, \ldots, |v|+1\}\\ &\quad ([\rhd w \lhd, \rhd, 0, 0] \mathrel{\vdash_{\overline{T}}^{*}} [\rhd w \lhd, \rhd v, h_{\mathrm{i}}, h_{\mathrm{s}}]\ \Rightarrow\ |v| \leq m)\end{aligned}$$

Otherwise, we say T with w uses *unbounded amount of the storage tape.* If T with w eventually halts, then it uses bounded amount of the storage tape.

Theorem 2. *For any DTM $T = (Q, \Sigma, \Gamma, \delta, \rhd, \lhd, q_0, \#, A, R)$, we can construct an RDTM $T^\dagger = (Q^\dagger, \Sigma, \Gamma, \delta^\dagger, \rhd, \lhd, q_0, \#, \{\hat{q}_0^1\}, \{q_0^1\})$ such that the following holds.*

$$\begin{aligned}
&\forall w \in \Sigma^* \ (w \in L(T) \ \Rightarrow \ [\rhd w \lhd, \rhd, q_0, 0, 0] \ \vdash_{T^\dagger}^{*} \ [\rhd w \lhd, \rhd, \hat{q}_0^1, 0, 0]\,) \\
&\forall w \in \Sigma^* \ (w \notin L(T) \wedge T \textit{ with } w \textit{ uses bounded amount of the storage tape} \\
&\qquad\qquad \Rightarrow \ [\rhd w \lhd, \rhd, q_0, 0, 0] \ \vdash_{T^\dagger}^{*} \ [\rhd w \lhd, \rhd, q_0^1, 0, 0]\,) \\
&\forall w \in \Sigma^* \ (w \notin L(T) \wedge T \textit{ with } w \textit{ uses unbounded amount of the storage tape} \\
&\qquad\qquad \Rightarrow \ T^\dagger\textit{'s computation starting from } [\rhd w \lhd, \rhd, q_0, 0, 0] \textit{ does not halt})
\end{aligned}$$

Hence $L(T) = L(T^\dagger)$. Furthermore, if T uses at most m squares of the storage tape on an input w, then $T^\dagger$ with w also uses at most m squares in any of its configuration in its computing process.

Proof outline. The construction method is similar to the case of RDMFA in Theorem 1. We first give the description of $T^\dagger$ below.

The five functions prev-rw : $Q_{\mathrm{rw}} \to 2^{Q_{\mathrm{s}} \times \{-,0,+\}^2}$, prev-s : $Q_{\mathrm{s}} \times (\Sigma \cup \{\rhd, \lhd\}) \times (\Gamma \cup \{\rhd, \#\}) \to 2^{Q_{\mathrm{rw}} \times (\Gamma \cup \{\rhd, \#\})}$, $\mathrm{deg}_{\mathrm{rw}} : Q_{\mathrm{rw}} \to \mathbb{N}$, $\mathrm{deg}_{\mathrm{s}} : Q_{\mathrm{s}} \times (\Sigma \cup \{\rhd, \lhd\}) \times (\Gamma \cup \{\rhd, \#\}) \to \mathbb{N}$, and $\mathrm{deg}_{\max} : Q \to \mathbb{N}$ are defined as follows.

$$\begin{aligned}
\text{prev-rw}(q) &= \{[p, d, d'] \mid p \in Q_{\mathrm{s}} \wedge d, d' \in \{-, 0, +\} \wedge [p, d, d', q] \in \delta\} \\
\text{prev-s}(q, s, u) &= \{[p, t] \mid p \in Q_{\mathrm{rw}} \wedge t \in (\Gamma \cup \{\rhd, \#\}) \wedge [p, s, [t, u], q] \in \delta\} \\
\mathrm{deg}_{\mathrm{rw}}(q) &= |\text{prev-rw}(q)| \\
\mathrm{deg}_{\mathrm{s}}(q, s, u) &= |\text{prev-s}(q, s, u)| \\
\mathrm{deg}_{\max}(q) &= \begin{cases} \mathrm{deg}_{\mathrm{rw}}(q) & \text{if } q \in Q_{\mathrm{rw}} \\ \max\{\mathrm{deg}_{\mathrm{s}}(q, s, u) \mid s \in (\Sigma \cup \{\rhd, \lhd\}) & \\ \qquad\qquad \wedge\ u \in (\Gamma \cup \{\rhd, \#\})\} & \text{if } q \in Q_{\mathrm{s}} \end{cases}
\end{aligned}$$

Also in this case, we assume that the sets Q and $\Gamma \cup \{\rhd, \#\}$ are totally ordered, and thus prev-rw(q) and prev-s(q, s, u) are expressed by the ordered lists below.

$$\begin{aligned}
\text{prev-rw}(q) &= [[p_1, d_1, d_1'], \ldots, [p_{\mathrm{deg}_{\mathrm{rw}}(q)}, d_{\mathrm{deg}_{\mathrm{rw}}(q)}, d'_{\mathrm{deg}_{\mathrm{rw}}(q)}]] \\
\text{prev-s}(q, s, u) &= [[p_1, t_1], \ldots, [p_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}, t_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}]]
\end{aligned}$$

$Q^\dagger$ and $\delta^\dagger$ of $T^\dagger$ are defined as below.

$$\begin{aligned}
&Q^\dagger = \{q, \hat{q} \mid q \in Q\} \cup \{q^j, \hat{q}^j \mid q \in Q \wedge j \in (\{1\} \cup \{1, \ldots, \mathrm{deg}_{\max}(q)\})\} \\
&\delta^\dagger = \ \delta_1 \cup \cdots \cup \delta_6 \cup \hat{\delta}_1 \cup \cdots \cup \hat{\delta}_5 \cup \delta_{\mathrm{a}} \cup \delta_{\mathrm{r}} \\
&\delta_1 = \{\, [p_1, d_1, d_1', q^2], \ldots, [p_{\mathrm{deg}_{\mathrm{rw}}(q)-1}, d_{\mathrm{deg}_{\mathrm{rw}}(q)-1}, d'_{\mathrm{deg}_{\mathrm{rw}}(q)-1}, q^{\mathrm{deg}_{\mathrm{rw}}(q)}], \\
&\qquad [p_{\mathrm{deg}_{\mathrm{rw}}(q)}, d_{\mathrm{deg}_{\mathrm{rw}}(q)}, d'_{\mathrm{deg}_{\mathrm{rw}}(q)}, q] \mid q \in Q_{\mathrm{rw}} \wedge \mathrm{deg}_{\mathrm{rw}}(q) \geq 1 \\
&\qquad \wedge \text{prev-rw}(q) = [[p_1, d_1, d_1'], \ldots, [p_{\mathrm{deg}_{\mathrm{rw}}(q)}, d_{\mathrm{deg}_{\mathrm{rw}}(q)}, d'_{\mathrm{deg}_{\mathrm{rw}}(q)}]] \,\} \\
&\delta_2 = \{\, [p_1, s, [t_1, u], q^2], \ldots, [p_{\mathrm{deg}_{\mathrm{s}}(q,s,u)-1}, s, [t_{\mathrm{deg}_{\mathrm{s}}(q,s,u)-1}, u], q^{\mathrm{deg}_{\mathrm{s}}(q,s,u)}], \\
&\qquad [p_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}, s, [t_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}, u], q] \mid q \in Q_{\mathrm{s}} \wedge s \in (\Sigma \cup \{\rhd, \lhd\}) \wedge u \in (\Gamma \cup \{\rhd, \#\}) \\
&\qquad \wedge \mathrm{deg}_{\mathrm{s}}(q, s, u) \geq 1 \wedge \text{prev-s}(q, s, u) = [[p_1, t_1], \ldots, [p_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}, t_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}]] \,\} \\
&\delta_3 = \{\, [q^1, -d_1, -d_1', p_1^1], \ldots, [q^{\mathrm{deg}_{\mathrm{rw}}(q)}, -d_{\mathrm{deg}_{\mathrm{rw}}(q)}, -d'_{\mathrm{deg}_{\mathrm{rw}}(q)}, p^1_{\mathrm{deg}_{\mathrm{rw}}(q)}] \mid q \in Q_{\mathrm{rw}} \\
&\qquad \wedge \mathrm{deg}_{\mathrm{rw}}(q) \geq 1 \wedge \text{prev-rw}(q) = [[p_1, d_1, d_1'], \ldots, [p_{\mathrm{deg}_{\mathrm{rw}}(q)}, d_{\mathrm{deg}_{\mathrm{rw}}(q)}, d'_{\mathrm{deg}_{\mathrm{rw}}(q)}]] \,\} \\
&\delta_4 = \{\, [q^1, s, [u, t_1], p_1^1], \ldots, [q^{\mathrm{deg}_{\mathrm{s}}(q,s,u)}, s, [u, t_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}], p^1_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}] \mid \\
&\qquad q \in Q_{\mathrm{s}} \wedge s \in (\Sigma \cup \{\rhd, \lhd\}) \wedge u \in (\Gamma \cup \{\rhd, \#\}) \wedge \mathrm{deg}_{\mathrm{s}}(q, s, u) \geq 1 \\
&\qquad \wedge \text{prev-s}(q, s, u) = [[p_1, t_1], \ldots, [p_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}, t_{\mathrm{deg}_{\mathrm{s}}(q,s,u)}]] \,\}
\end{aligned}$$

$$\begin{aligned}
\delta_5 =&\{\, [q^1, s, [u,u], q] \mid q \in Q_{\mathrm{s}} - (A \cup R) \wedge s \in (\Sigma \cup \{\triangleright, \triangleleft\}) \wedge u \in (\Gamma \cup \{\triangleright, \#\}) \\
&\wedge \mathrm{deg_s}(q,s,u) = 0 \,\} \\
\hat{\delta}_i =&\{\, [\hat{p}, x, y, \hat{q}] \mid [p,x,y,q] \in \delta_i \,\} \ (i = 1,\ldots,5) \\
\delta_6 =&\{\, [q, s, [t,t], q^1] \mid q \in Q_{\mathrm{rw}} - \{q_0\} \wedge s \in (\Sigma \cup \{\triangleright, \triangleleft\}) \wedge t \in (\Gamma \cup \{\triangleright, \#\}) \\
&\wedge \neg \exists u \exists p\, ([q,s,[t,u],p] \in \delta) \,\} \\
\delta_{\mathrm{a}} =&\{\, [q, 0, 0, \hat{q}^1] \mid q \in A \,\} \\
\delta_{\mathrm{r}} =&\{\, [q, 0, 0, q^1] \mid q \in R \,\}
\end{aligned}$$

Assume an input w is given to T. The computation graph $G_{T,w}$ can be defined similarly as in MFA. Let $c = [\triangleright w \triangleleft, \triangleright v, q, h_{\mathrm{i}}, h_{\mathrm{s}}]$ and $c' = [\triangleright w \triangleleft, \triangleright v', q', h'_{\mathrm{i}}, h'_{\mathrm{s}}]$ be two configurations of T such that $c \mathrel{|\!\!\frac{\;*\;}{T}} c'$, where $v, v' \in \Gamma^*$, $q, q' \in Q$, $h_{\mathrm{i}}, h'_{\mathrm{i}} \in \{0, \ldots, |w|+1\}$, $h_{\mathrm{s}} \in \{0, \ldots, |v|+1\}$, and $h'_{\mathrm{s}} \in \{0, \ldots, |v'|+1\}$. Since T does not erase a non-blank symbol by the assumption (T6), the relation $|v| \le |v'|$ holds. Therefore, for each c', the set $\{c \mid c \mathrel{|\!\!\frac{\;*\;}{T}} c'\}$ is finite.

First, consider the case where T with w uses bounded amount of the storage tape. Assume T finally halts in a configuration c_{h}. Then $G_{T,w}$ becomes a finite tree with the root c_{h}, since $\{c \,|\, c \mathrel{|\!\!\frac{\;*\;}{T}} c_{\mathrm{h}}\}$ is finite. On the other hand, if T eventually enters a loop of configurations $c_1 \mathrel{|\!\!\frac{\;\;}{T}} \cdots \mathrel{|\!\!\frac{\;\;}{T}} c_k \mathrel{|\!\!\frac{\;\;}{T}} c_1 \mathrel{|\!\!\frac{\;\;}{T}} \cdots$, then $G_{T,w}$ is not a tree. But, it is a finite graph, since $\{c \mid \exists i \in \{1, \ldots, k\}(c \mathrel{|\!\!\frac{\;*\;}{T}} c_i)\}$ is again finite. In these cases, $T^\dagger$ traverses $G_{T,w}$ in the same way as in Theorem 1, and halts in the state q_0^1 or $\hat{q}_0^1$. This is because a lemma for RDTM analogous to Lemma 1 holds, since the total number of its configurations is finite. Here, there is a small problem: $T^\dagger$ may halt in the configuration $[\triangleright w \triangleleft, \triangleright v, \hat{q}_0^1, 0, 0]$ or $[\triangleright w \triangleleft, \triangleright v, q_0^1, 0, 0]$ such that $v \in \Gamma^+$. However, such a case is excluded by modifying T so that it confirms the storage tape square of the position 1 (i.e., the square immediately right of $\triangleright$) contains the symbol $\#$ just after it starts from q_0. It is done by adding a few states to Q. By this, $T^\dagger$ gives a correct result.

Second, consider the case where T with w uses unbounded amount of the storage tape. In this case, $G_{T,w}$ becomes an infinite graph. Thus, $T^\dagger$ traverses $G_{T,w}$ indefinitely without halting (its proof is omitted here).

It is easy to see that if T uses m squares of the storage tape, then $T^\dagger$ also uses m squares. This is because the head positions and the contents of the storage tape of T is directly simulated by those of $T^\dagger$,

By above, we can conclude that the theorem holds. □

Example 2. Consider the the following language.

$$L_{\mathrm{eq}} = \{w \mid w \in \{a,b\}^* \wedge \text{ the number of } a\text{'s in } w \text{ is the same as that of } b\text{'s}\}$$

The DTM T_{eq} defined below accepts the language L_{eq}.

$$\begin{aligned}
T_{\mathrm{eq}} &= (Q, \{a,b\}, \{a,b\}, \delta, \triangleright, \triangleleft, q_0, \#, \{q_{\mathrm{a}}\}, \{q_{\mathrm{r}}\}) \\
Q &= \{q_0, q_1, \ldots, q_6, q_{\mathrm{a}}, q_{\mathrm{r}}\} \\
\delta &= \{\, [q_0, \triangleright, [\triangleright, \triangleright], q_1],\ [q_1, +, +, q_2], \\
&\qquad [q_2, a, [\#, a], q_1],\ [q_2, b, [\#, \#], q_3],\ [q_2, \triangleleft, [\#, \#], q_4],\ [q_3, +, 0, q_2], \\
&\qquad [q_4, -, -, q_5],\ [q_5, a, [a, a], q_6],\ [q_5, a, [\triangleright, \triangleright], q_6],\ [q_5, b, [a, b], q_4], \\
&\qquad [q_5, b, [\triangleright, \triangleright], q_{\mathrm{r}}],\ [q_5, \triangleright, [a, a], q_{\mathrm{r}}],\ [q_5, \triangleright, [\triangleright, \triangleright], q_{\mathrm{a}}],\ [q_6, -, 0, q_5] \,\}
\end{aligned}$$

If an input word $w \in \{a,b\}^*$ is given, $T_{\rm eq}$ scans it from left to right. Each time $T_{\rm eq}$ finds the symbol a in w, it writes a in the storage tape, and shift the head to the right. It is performed by the states q_1, q_2 and q_3. If the input head reaches $\lhd$, then $T_{\rm eq}$ changes the direction of scanning. After that, each time $T_{\rm eq}$ finds the symbol b in w, it rewrites the storage tape symbol a into b, and shift the head to the left using q_4, q_5 and q_6. If both of the heads reads $\rhd$, it accepts the input. Otherwise it rejects. $T_{\rm eq}$ is irreversible, since the pairs $([q_1,+,+,q_2],\ [q_3,+,0,q_2])$ and $([q_4,-,-,q_5],\ [q_6,-,0,q_5])$ violate the reversibility condition. Examples of computing processes of $T_{\rm eq}$ are as below.

$$[\ \rhd aabbabba\lhd,\ \rhd,\ q_0,\ 0,\ 0\ ] \mathrel{|\!\!\frac{37}{T_{\rm eq}}} [\ \rhd aabbabba\lhd,\ \rhd bbbb,\ q_{\rm a},\ 0,\ 0\ ]$$
$$[\ \rhd aabbabaa\lhd,\ \rhd,\ q_0,\ 0,\ 0\ ] \mathrel{|\!\!\frac{37}{T_{\rm eq}}} [\ \rhd aabbabaa\lhd,\ \rhd aabbb,\ q_{\rm r},\ 0,\ 2\ ]$$

An RDTM $T_{\rm eq}^\dagger$ that simulates $T_{\rm eq}$ obtained by the method in Theorem 2 is:

$$\begin{array}{lll}
T_{\rm eq}^\dagger &=& (Q^\dagger, \{a,b\}, \{a,b\}, \delta^\dagger, \rhd, \lhd, q_0, \#, \{\hat{q}_0^1\}, \{q_0^1\}) \\
Q^\dagger &=& \{q, \hat{q}, q^1, \hat{q}^1 \mid q \in Q\} \cup \{\ q_2^2, q_5^2, \hat{q}_2^2, \hat{q}_5^2\ \} \\
\delta^\dagger &=& \delta_1 \cup \cdots \cup \delta_6 \cup \hat{\delta}_1 \cup \cdots \cup \hat{\delta}_5 \cup \delta_{\rm a} \cup \delta_{\rm r} \\
\delta_1 &=& \{\ [q_1,+,+,q_2^2],\ [q_3,+,0,q_2],\ [q_4,-,-,q_5^2],\ [q_6,-,0,q_5]\ \} \\
\delta_2 &=& \{\ [q_0,\rhd,[\rhd,\rhd],q_1],\ [q_2,a,[\#,a],q_1],\ [q_2,b,[\#,\#],q_3],\ [q_2,\lhd,[\#,\#],q_4], \\
&& [q_5,a,[a,a],q_6],\ \ [q_5,a,[\rhd,\rhd],q_6],\ [q_5,b,[a,b],q_4],\ \ [q_5,b,[\rhd,\rhd],q_{\rm r}], \\
&& [q_5,\rhd,[a,a],q_{\rm r}],\ \ [q_5,\rhd,[\rhd,\rhd],q_{\rm a}]\ \} \\
\delta_3 &=& \{\ [q_2^1,-,-,q_1^1],\ [q_2^2,-,0,q_3^1],\ [q_5^1,+,+,q_4^1],\ [q_5^2,+,0,q_6^1]\ \} \\
\delta_4 &=& \{\ [q_1^1,\rhd,[\rhd,\rhd],q_0^1],\ [q_1^1,a,[a,\#],q_2^1],\ [q_3^1,b,[\#,\#],q_2^1],\ [q_4^1,\lhd,[\#,\#],q_2^1], \\
&& [q_6^1,a,[a,a],q_5^1],\ \ [q_6^1,a,[\rhd,\rhd],q_5^1],\ [q_4^1,b,[b,a],q_5^1],\ \ [q_{\rm r}^1,b,[\rhd,\rhd],q_5^1], \\
&& [q_{\rm r}^1,\rhd,[a,a],q_5^1],\ \ [q_{\rm a}^1,\rhd,[\rhd,\rhd],q_5^1]\ \} \\
\delta_5 &=& \{\ [q_1^1,\rhd,[\#,\#],q_1],\ [q_1^1,\rhd,[a,a],q_1],\ \ \ldots\ ,[q_6^1,\lhd,[b,b],q_6]\ \} \\
\hat{\delta}_i &=& \{\ [\hat{p},x,y,\hat{q}] \mid [p,x,y,q] \in \delta_i\ \}\ (i=1,\ldots,5) \\
\delta_6 &=& \{\ [q_2,\rhd,[\rhd,\rhd],q_2^1],\ [q_2,\rhd,[\#,\#],q_2^1],\ \ \ldots\ ,[q_5,\lhd,[b,b],q_5^1]\ \} \\
\delta_{\rm a} &=& \{\ [q_{\rm a},0,0,\hat{q}_{\rm a}^1]\ \} \\
\delta_{\rm r} &=& \{\ [q_{\rm r},0,0,q_{\rm r}^1]\ \}
\end{array}$$

The details of δ_5 and δ_6 are omitted here since they have 57 and 23 rules, respectively. Examples of computing processes of $T_{\rm eq}^\dagger$ are as follows.

$$[\ \rhd aabbabba\lhd,\ \rhd,\ q_0,\ 0,\ 0\ ] \mathrel{|\!\!\frac{129}{T_{\rm eq}^\dagger}} [\ \rhd aabbabba\lhd,\ \rhd,\ \hat{q}_0^1,\ 0,\ 0\ ]$$
$$[\ \rhd aabbabaa\lhd,\ \rhd,\ q_0,\ 0,\ 0\ ] \mathrel{|\!\!\frac{129}{T_{\rm eq}^\dagger}} [\ \rhd aabbabaa\lhd,\ \rhd,\ q_0^1,\ 0,\ 0\ ]$$

$\square$

5 Concluding Remarks

We showed that any DMFA can be converted to an equivalent RDMFA without increasing the number of heads. The RDMFA is garbage-less in the sense it always halts for any input putting its heads at the left endmarker. The method is then applied for converting a DTM to an equivalent RDTM. The constructed RDTM not only uses the same number of storage tape squares, but also it is garbage-less, and uses the same number of storage symbols as the DTM. Thus it gives a bit stronger result than that of Lange et al. [6]. In addition, the method is applicable to a TM whose space-bound is not known.

In this paper, we formulated a TM as an acceptor. But, our method can also be applied to a TM that computes a function, i.e., a TM as a transducer. An easy way of doing it is to add an output tape to a simulating RDTM $T^\dagger$ besides its input and storage tapes. There, the traversing method of a computation graph of a given DTM T is the same as in Theorem 2. The difference is as follows. If the simulating RDTM $T^\dagger$ enters an accepting configuration of T, it copies the contents of the storage tape to the output tape. After that $T^\dagger$ continues to traverse the graph, and halts when it reaches the initial configuration of T. Other methods that do not use an output tape, e.g., using another track of the storage tape, are also possible. We can also apply our method to some of other acceptors, e.g., a marker automaton [3], a model closely related to an MFA.

Acknowledgement. This work was supported by JSPS KAKENHI Grant Number 21500015, 24500017.

References

1. Axelsen, H.: Reversible Multi-head Finite Automata Characterize Reversible Logarithmic Space. In: Dediu, A.-H., Martín-Vide, C. (eds.) LATA 2012. LNCS, vol. 7183, pp. 95–105. Springer, Heidelberg (2012)
2. Bennett, C.H.: Logical reversibility of computation. IBM J. Res. Dev. 17, 525–532 (1973)
3. Blum, M., Hewitt, C.: Automata on a two-dimensional tape. In: Proc. IEEE Symp. on Switching and Automata Theory, pp. 155–160. IEEE (1967)
4. Holzer, M., Kutrib, M., Malcher, A.: Complexity of multi-head finite automata: Origins and directions. Theoret. Comput. Sci. 412, 83–96 (2011)
5. Kondacs, A., Watrous, J.: On the power of quantum finite state automata. In: Proc. 36th FOCS, pp. 66–75. IEEE (1997)
6. Lange, K.J., McKenzie, P., Tapp, A.: Reversible space equals deterministic space. J. Comput. Syst. Sci. 60, 354–367 (2000)
7. Morita, K.: Two-way reversible multi-head finite automata. Fundamenta Informaticae 110, 241–254 (2011)
8. Sipser, M.: Halting space-bounded computations. Theoret. Comput. Sci. 10, 335–338 (1980)

Undecidability of the Surjectivity of the Subshift Associated to a Turing Machine

Rodrigo Torres[1], Nicolas Ollinger[2], and Anahí Gajardo[1,⋆]

[1] Departamento de Ingeniería Matemática, Centro de Investigación en Ingeniería Matemática, Centro de Modelamiento Matemático, Universidad de Concepción, Casilla 160-C, Concepción, Chile
`rtorres, anahi@ing-mat.udec.cl`
[2] LIFO, Université d'Orléans
BP 6759, F-45067 Orléans Cedex 2, France
`Nicolas.Ollinger@univ-orleans.fr`

Abstract. We consider Turing machines (TM) from a dynamical system point of view, and in this context, we associate a subshift by taking the sequence of symbols and states that the head has at each instant. Taking a subshift that select only a part of the state of a system is a classical technic in dynamical systems that plays a central role in their analysis. Surjectivity of Turing machines is equivalent to their reversibility and it can be simply identified from the machine rule. Nevertheless, the associated subshift can be surjective even if the machine is not, and the property results to be undecidable in the symbolic system.

Keywords: Turing machines, discrete-time dynamical systems, subshifts, formal languages.

Relations between dynamics and computation has been looked for in several works [1,2,3,4]. In a first approach, these two concepts are very different things, roughly speaking, one can say that computation consists in obtaining an output starting from an input by means of a dynamics. The dynamics itself is not relevant, several dynamics can produce the same result. On the other hand, a complex computation cannot be obtained through a too simple dynamics. Some –weak– relations exists.

A direct way to tackle this topic consists in looking at Turing machines with the tools of dynamical systems theory. A first paper by Kürka has taken this viewpoint [4] and several others have followed [5,6,1,3,2]. There, notions such as equicontinuity, entropy and periodicity have been studied, and putted in relation with more natural properties of the machines. Some of these properties were proved to be undecidable, as is the case of *periodicity* of Turing machines in [3].

Here we continue in the line of [1,2] that focus on a particular symbolic system (a subshift) associated with the Turing machine, that is called *t-shift*. It consists in taking the linear sequence of states and symbols that the machine reads

⋆ This work has been supported by CONICYT FONDECYT #1090568 and BASAL project CMM, Universidad de Chile, and CI^{2}MA, Universidad de Concepción.

during its evolution over a given initial configuration, and to consider afterwards the set of infinite sequences produced by all the possible initial configurations. Subshifts are key tools in the study of general dynamical systems, they give crucial information about the system (see for example [7]). In this approach, the complexity of the subshift has been related with the complexity of the machine.

In this paper, we study the *surjectivity* of the t-shift. A function T is surjective if for every y, there exists an x such that $T(x) = y$. If T is the function that defines the evolution of a Turing machine, it results to be equivalent to the reversibility of the machine and it can be characterized in a very simple way from the machine's transition rule. If the machine is surjective, so it is its associated t-shift, but the converse is not true. Thus we look for a characterization of the surjectivity of the t-shift in terms of some property of the machine.

When a subshift is surjective, every sequence can be extended by the left, in such a way that the subshift itself can be considered as a set of bi-infinite sequences, *i.e.*, sequences running over $\mathbb{Z}$. In this case, the shift action is reversible, and other properties can be considered.

Another reason to study this property is that surjectivity is a necessary condition for transitivity, which is a relevant property in the area of dynamical systems.

The following section provides definitions and concepts about symbolic dynamics and Turing machines. Section 2 gives a characterization of the t-shift surjectivity. In section 3, we establish the undecidability of some preliminary problems, to conclude with the undecidability of the property in the last section.

1 Definitions

1.1 Turing Machine

Turing Machine Written in Quadruples. Following Morita [8], a Turing machine (TM) M is a tuple (Q, Σ, δ), where Q is a finite set of states, Σ is a finite set of symbols and $\delta \subseteq Q \times \Sigma \times \Sigma \times Q \cup Q \times \{/\} \times \{-1, 0, +1\} \times Q$ is the writing/moving relation of the machine. The machine works on a tape, usually bi-infinite, full of symbols from Σ. A *configuration* is an element (w, i, q) of $\Sigma^{\mathbb{Z}} \times \mathbb{Z} \times Q$. A writing instruction is a quadruple (q, s, s', q'); it can be applied to a configuration (w, i, q'') if $w_i = s$ and $q = q''$, leading to the configuration (w', i, q'), where $w'_i = s'$ and $w'_k = w_k$ for all $k \neq i$. A moving instruction is a quadruple $(q, /, d, q')$; it can be applied to a configuration (w, i, q'') if $q = q''$, leading to the configuration $(w, i + d, q')$.

Turing Machine Written in Quintuples. A Turing machine M can also be written in quintuples by having the writing/moving relation δ considered as $\delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{-1, 0, +1\}$. A quintuple instruction (q, s, q', s', d) can be applied to a configuration (w, i, q'') if $w_i = s$ and $q = q''$, leading to the configuration $(w', i + d, q')$, where $w'_i = s'$ and $w'_k = w_k$ for all $k \neq i$.

Turing machines, when viewed as computing model, have a particular starting state q_0, and a particular symbol called blank symbol; the computation is intended to start over a configuration $(w, 0, q_0)$, where w represents the input, a word with a finite number of non-blank symbols. The computation process stops when the machine reaches another particular state: the halting state q_F. In this paper, we are omitting these three parameters, since we do not want the machine to halt and we will study its dynamics for arbitrary initial configurations. In any case, the halting problem can be translated to the present context as the problem of deciding whether the machine reaches a particular state when starting in another particular state with an homogeneous configuration except for a finite number of cells.

We also remark that the quintuples model is the traditional one, while the quadruples model is used for reversible Turing machines. One can translate any machine written in quadruples into a machine written in quintuples in a simple way because *writing instructions* are just quintuples instructions that do not cause any movement, and *moving instructions* are those that do not modify the tape. The converse transformation is also possible but a quintuple instruction will need to be replaced by a writing instruction followed by a moving instruction, thus the set of states needs to be duplicated and the time is also multiplied by two. Therefore, both models are equivalent as computing system, but not as dynamical system.

Deterministic Turing Machine. A Turing machine M is *deterministic* if, for any configuration $(w, i, q) \in X$, at most one instruction can be applied (regardless the machine is written in quadruples or quintuples). In terms of quintuples, this is equivalent to give δ as a (possibly partial) function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, +1\}$. This function δ can be projected into three components $\delta_Q : Q \times \Sigma \to Q$, $\delta_S : Q \times \Sigma \to \Sigma$ and $\delta_D : Q \times \Sigma \to \{-1, 0, +1\}$.

Complete Turing Machine. In any of the two models, if no instruction can be applied, the machine halts. A Turing machine M is *complete* if for each configuration (w, i, q), at least one instruction can be applied, *i.e.*, it never halts.

Analogous notions can be defined when going backward in time.

Backward Deterministic Turing Machine. A Turing machine M is *backward deterministic* if each configuration comes from at most one previous configuration.

A Turing machine written in quadruples is backward deterministic (as seen in [8]) if and only if for any two different quadruples (q, s, s', q') and (q'', s'', s''', q') in δ, it holds:

$$s \neq / \wedge s'' \neq / \wedge s' \neq s''' . \tag{1}$$

Backward Complete Turing Machine. A Turing machine M is *backward complete* if each configuration comes from at least one preimage.

Reversible Turing Machine. A Turing machine is reversible if it is deterministic forward and backward. For a machine written in quadruples, reversing the quadruples gives the reverse machine. The reverse instruction of a writing instruction (q, s, s', q') is (q', s', s, q). The reverse instruction of a movement instruction $(q, /, d, q')$ is $(q', /, -d, q)$. It is not difficult to see that a reversible Turing machine is complete if and only if its reverse is complete.

All of these last properties are local, and they can be checked in a finite number of steps.

1.2 Dynamical System

A *dynamical system* is a pair (X, T), where X is called phase space and $T : X \to X$ is called *global transition function*. In this paper, we consider $X = \Sigma^{\mathbb{Z}} \times \mathbb{Z} \times Q$. $\Sigma^{\mathbb{Z}}$ is called the *two-sided full shift* and its elements are called *bi-infinite words*, the symbol $'$ will be used to mark the position 0; for example, $\ldots 2333'233124 \ldots$ indicates that 2 is set in the position 0. $\Sigma^{\mathbb{N}}$ is the *one-sided full shif*t and its elements are called *infinite words*.

Subshifts. The *shift* function σ, is defined both in $\Sigma^{\mathbb{Z}}$ and $\Sigma^{\mathbb{N}}$ either by $\sigma(\ldots w_{-2}w'_{-1}w_0w_1w_2\ldots) = \ldots w_{-1}w'_0w_1w_2w_3\ldots$ or $\sigma(w_1w_2w_3\ldots) = w_2w_3\ldots$; it is a bijective function in the first case. Σ^* denotes the set of finite sequences of elements of Σ, called *finite words*. Two words $z = z_0...z_n$ and $y = y_0...y_m$ can be *concatenated* by just putting them one after the other: $zy = z_0...z_ny_0...y_m$. A word x can also be concatenated with a semi-infinite word $w = w_0w_1w_2...$: $xw = x_0...x_nw_0w_1\ldots$ A finite word z is said to be a *subword* of another (finite or infinite) word v, if there exists two indices i and j, such that $z = v_iv_{i+1}...v_j$. In this case we write: $z \sqsubseteq v$. Subsets of Σ^* are called *formal languages*. Given a subset of the full shift S, a formal language is defined:

$$\mathcal{L}(S) = \{z \in \Sigma^* \mid (\exists w \in S)\ z \sqsubseteq w\} . \tag{2}$$

Reciprocally, given a formal language L, a set of infinite sequences can be defined:

$$\mathcal{S}_L = \{w \in \Sigma^{\mathbb{M}} \mid (\forall z \sqsubseteq w)\ z \in L\} . \tag{3}$$

When S satisfies $\mathcal{S}_{\mathcal{L}(S)} = S$, it is called a *subshift*.

The *t*-shift. A complete and deterministic Turing machine $M = (Q, \Sigma, \delta)$ can be associated with a dynamical system (X, T), where X is the set of configurations $\Sigma^{\mathbb{Z}} \times \mathbb{Z} \times Q$, and the global transition function $T : X \to X$ consists into apply one transition of the Turing machine. We define $\pi : X \to Q \times \Sigma$ by $\pi(w, i, q) = (q, w_i)$. The *t-shift* associated to T, denoted by $S_T \subseteq (Q \times \Sigma)^{\mathbb{N}}$, is the set of orbits $\tau(x) = (\pi(T^n(x)))_{n\in\mathbb{N}}$, for $x \in X$. It is not difficult to see that S_T is in fact a subshift [1].

2 Surjectivity

As we have said, when M is deterministic and complete, T is a function. In this context, backward determinism is equivalent to injectivity of T and backward completeness corresponds to surjectivity. Through a cardinality argument, it is possible to show that, when the machine is deterministic and complete, surjectivity is equivalent to injectivity and both are easy to check from the machine's transition rule. From now on, we will work only with deterministic and complete Turing machines.

Remark 1. A Turing machine $M = (Q, \Sigma, \delta)$ written in quintuples is surjective if and only if, for every $q' \in Q$ and s', r', $t' \in \Sigma$, there is at least one $q \in Q$ and $s \in \Sigma$ such that $\delta(q, s) = (q', s', +1)$ or $\delta(q, s) = (q', r', -1)$ or $\delta(q, s) = (q', t', 0)$.

If for some q', the condition of TM surjectivity is not satisfied, we say that q' is *defective.* Thus a machine is surjective if and only if it has no defective state.

Definition 1. *A state $q' \in Q$ of a Turing machine $M = (Q, \Sigma, \delta)$ is said to be defective if:*

1. *(Quintuple model) There exist symbols s', r', $t' \in \Sigma$ such that no instruction gives: $(q', s', +1)$, $(q', r', -1)$ or $(q', t', 0)$.*
2. *(Quadruple model) There exist $s' \in \Sigma$ such that there exist no instruction $(q, /, d, q')$ nor (q, s, s', q'), for no $q \in Q$, $s \in \Sigma$, and $d \in \{-1, 0, +1\}$.*

Notice that an unreachable state is indeed defective, but we will assume that every state is reachable. If not, the subshift S_T will not be surjective in any case.

Surjectivity of T is inherited by the subshift S_T, however, if T is not surjective, the subshift can still be surjective. For example, let M be the Turing machine that simply moves to the right by always writing a 0. This machine is not surjective, but the associated subshift does.

Remark 2. A t-shift is surjective if and only if: $(\forall u \in S_T)(\exists a \in Q \times \Sigma)\ au \in S_T$

If $u = \left(\begin{smallmatrix} q_1 & q_2 & \cdots \\ s_1 & s_2 & \cdots \end{smallmatrix}\right) \in S_T$ and $a = (q, s)$, condition $au \in S_T$ says that $\delta_Q(q, s) = q_1$ and that the configuration that produces u has the symbol s at position $-\delta_D(q, s)$. If the machine does not visit position $-\delta_D(q, s)$, s can be any symbol, otherwise it is restricted to the constraint $\delta_S(q, s) = s_{i+1}$, where $i = \min\{j \mid \sum_{k=1}^{j} \delta_D(u_k) = -\delta_D(q, s)\}$.

In the example, the unique state of the machine is defective, it does not admit the symbol '1' at the left of the head, but position -1 is never revisited, that is why any symbol can be appended at the beginnig of u. If the state q_1 is defective and it does not admits the symbol s_{i+1} at position $-\delta(q, s)$, then $au \notin S_T$. Surjectivity will be possible when defective states avoid the head from going in to the "conflictive" positions, we develop this in the next section.

It is important to note that in the quadruples model, the surjectivity of T is held by the subshift S_T and vice versa. If we have a defective state q_1, then there exist no moving instruction leading to q_1. From the previous assertion, if q_1 is defective, then S_T is not surjective. We are interested in surjectivity only within the quintuples model.

2.1 Blocking States

We say that a state q is a *blocking state to the left (right)* if:

$$(\forall u \in S_T)(\forall s \in \Sigma)\ u_1 = (s,q) \Rightarrow \left[(\forall j \in \mathbb{N})\ \sum_{k=1}^{j} \delta_D(u_k) \neq -1(+1)\right] . \tag{4}$$

We also say that q is an *s-blocking state to the left (right)* for a given $s \in \Sigma$, if:

$$(\forall u \in S_T)\ u_1 = (s,q) \Rightarrow \left[(\forall j \in \mathbb{N})\ \sum_{k=1}^{j} \delta_D(u_k) \neq -1(+1)\right] . \tag{5}$$

Finally we say that q is just a *blocking state*, if for every $s \in \Sigma$, q is an s-blocking state either to the left or right.

A state q is said to be *reachable from the left (right)* if there exists a state q' and symbols s, s' such that $\delta(s', q') = (s, q, +1)$ (resp. -1).

The surjectivity on S_T can be characterized through these notions, in fact, S_T is surjective if and only if for each $q' \in Q$ at least one of the following holds:

1. q' is not defective: If q' is not defective, then, independently on the context, it can be reached from some configuration.
2. q' is blocking to the left (right) and it is reachable from the left (right): If q' happens to be a blocking state to the left (right), no configuration producing $u \in S_T$, with $u_1 = (q', s_1)$, is able to revisit the position -1 $(+1)$ (with respect to the initial head position), so any $(q, s) \in Q \times \Sigma$, such that $\delta_Q(q, s) = q'$ and $\delta_D(q, s) = +1(-1)$, can be appended at the beginning of u.
3. q' is blocking and it is reachable from the left and from the right: If q' happens to be a blocking state, $u \in S_T$ starts with $u_1 = (q', s_1)$ and q' is s_1-blocking to the left (right), then no configuration producing u is able to revisit the position -1 $(+1)$ (with respect to the initial head position), so any $(q, s) \in Q \times \Sigma$, satisfying $\delta_Q(q, s) = q'$ and $\delta_D(q, s) = +1(-1)$ can be appended at the beginning of u.

If we could decide when a state is blocking, we could decide surjectivity. In the next section, however, we see that checking the blocking property is not possible.

3 Undecidability of Preliminary Problems

In this section, we show the undecidability of several problems related with the blocking property of a state, that will serve as intermediate to finally prove the undecidability of the surjectivity on S_T in section 4.

Let us remark that the following proofs are equivalent in quadruples and quintuples model, one only has to use the usual transformation described in the section 1. The last is possible because the following problems are related to movement abilities of the head.

3.1 Undecidability of the Blocking State Problem

Let us consider the next three problems.

(BSl) Given a Turing machine M and a state q, decide whether q is a blocking state to the left.
(BSr) Given a Turing machine M and a state q, decide whether q is a blocking state to the right.
(BS) Given a Turing machine M and a state q, decide whether q is a blocking state.

Let us remark that (BSl) and (BSr) are Turing equivalent, *i.e.*, (BSl) reduces to (BSr) and vice versa. To see this it is enough to see that switching the movement direction on every instruction of a given machine M produces a machine M' whose states are blocking to the left if and only if the respective states of M are blocking to the right.

We prove the undecidability of these three problems by reduction from the emptiness problem, which is known to be undecidable for machines written either in quadruples or quintulples and also for machines restricted to work on a semi infinite tape. The definition of the emptiness problem is adapted to the present context as follows.

(E) Given a Turing machine M, and two states q_0 and q_F, decide whether there is an input configuration $(w, 0, q_0)$ that makes the machine to reach the state q_F in finite time.

Lemma 1. *(BSl) is undecidable.*

Proof. We prove undecidability by reduction from the emptiness problem. Let $M = (Q, \Sigma, \delta)$ be a Turing machine, and let $q_0, q_F \in Q$ be two states. We will assume, without loss of generality, that M is written in quintuples, and that starting with q_0 the head never goes to the left of position 0 (this is equivalent to say that the machine works only on the right side of the tape). Let us define M' just like M but with an additional state q_{aux}, and some small differences in its transition function δ:

$$\delta(q_F, s) = (q_{aux}, s, -1), \text{ and } \delta(q_{aux}, s) = (q_{aux}, s, -1), \text{ for every } s \in \Sigma \ . \quad (6)$$

Thus, M reaches q_F for some input $(w, 0, q_0)$ if and only if the state q_0 is not a blocking state to the left for M'. ■

Theorem 1. *(BS) is undecidable.*

Proof. Let $M = (Q, \Sigma, \delta)$ be a Turing machine, and let $q_0, q_F \in Q$ be two states. Let M' be a machine defined as in the last proof. Since M works only on the right side, we have that $\delta_D(q_0, s) = +1$ for every s, thus q_0 is not s-blocking to the right for any symbol s. Therefore, q_0 is blocking to the left for M' if and only if q_0 is a blocking state for M'.

It results that the emptiness problem is satisfied for (M, q_0, q_F) if and only if the blocking problem is satisfied for (M', q_0). ■

3.2 Undecidability of the Blocking State Problem in Complete RTMs

With the results of the last section, we discard the possibility of solving the problem of surjectivity via blocking states. However, knowing about the surjectivity of S_T for a given machine M does not help to solve the blocking problem for a particular state q of M. Thus surjectivity can still be decidable. We want to reduce the blocking state problem to the surjectivity problem, but to do so we need to produce a machine whose surjectivity depends only on the blocking property of one of its states. This can be achieved by working with reversible machines, which are modified to make one of its states defective. That is why we introduce a new problem.

(BSLrtm) Given a complete and reversible Turing machine M and a state q that is reachable from the left, decide whether q is a blocking state to the left.

We prove the undecidability of this problem by reduction from the halting problem of reversible two counter machines, which is proved undecidable in [9].

Definition 2. *A k-counter machine (k-CM) is a triple (S, k, R), where S is a finite set, $k \in \mathbb{N}$ is the number of counters, and $R \subseteq S \times \{0,+\}^k \times \{1,..,k\} \times \{-1,0,+1\} \times S$ is the transition relation. A configuration of the machine is a pair (s,ν), where s is the current state and $\nu \in \mathbb{N}^k$ is the content of the k counters. By considering the function* sign: $\mathbb{N}^k \to \{0,+\}^k$ *defined by* $\text{sign}(\nu)_j = 0$ *if* $\nu_j = 0$ *and $+$ otherwise, an instruction $(s,u,i,d,t) \in R$ can be applied to a configuration (s,ν) if* $\text{sign}(\nu) = u$*, and the new configuration is (t,ν') where $\nu'_j = \nu_j$ for every $j \neq i$ and $\nu'_i = \nu_i + d$. R cannot contain the instruction $(s,u,i,-1,t)$ if $u_i = 0$.*

Just like Turing machines, a k-counter machine is said to be *deterministic (k-DCM)* if at most one instruction can be applied to each configuration. In addition, a k-CM C is said to be *reversible (k-RCM)* if it is forward and backward deterministic.

The halting problem consists in determining, given an initial configuration (s,ν), whether the machine reaches a given halting state t. It is undecidable for $k = 2$, even if the initial configuration is fixed to $(s_0,(0,0))$.

Theorem 2. *(BSLrtm) is undecidable.*

Proof. We prove undecidability by reduction from the halting problem of 2-RCM.

Let $C = (S, 2, R)$ be a 2-RCM, with initial configuration $(s_0,(0,0))$ and final state $t \in S$. For this proof we need a Turing machine M and a state $q = q_0$ meeting the following:

1. it simulates C on the right side of the tape,
2. it is reversible,
3. it reaches the position -1 starting at 0 from q_0 if and only if C halts (reaches t) starting from $(s_0,(0,0))$,

4. it is complete, and
5. q_0 is reachable from the left.

If the machine meets the above objectives, q_0 will not be blocking to the left if and only if, starting from $(s_0, (0, 0))$, C halts.

The first 3 objectives can be viewed in detail in the appendix; however, here we sketch them briefly. For simplicity, we define M in the quadruple form. For the first objective, we make a traditional simulation. Let $M = (Q, \Sigma, \delta)$ be a Turing machine. Starting with q_0, the machine writes "$<\ |\ >$" into the tape, and goes to state s_0 (as seen in appendix, Part 1). This corresponds to the initial configuration $(s_0, (0, 0))$ of the counter machine. The simulation will correctly work if the tape initially contains only 1s. In general, each configuration $(s, (n, m))$ of the counter machine will be represented in the Turing machine by the configuration $(...' < 1^n|1^m > ..., 0, s)$. This will be achieved by simulating each instructions of C, as seen in the appendix, Part 2. In this way, M simulates C. It is noteworthy that, throughout the simulation, the machine does not reach the left side of the tape.

Providing simple safeguards, each counter machine instruction can be simulated by M in a reversible way. However, when reaching a state $s \in S$, while the counter machine knows whether each of its register is empty or not, the Turing machine does not, thus it may not be reversible in such situations. In order to solve this, the symbols of the form (d, d') are added to Σ, for each $d, d' \in \{0, +\}$; and we better represent C configurations by $(...'(d, d')1^n|1^m > ..., 0, s)$.

Now, for the third goal, when M reaches the halting state t of C, we add an extra transition to move to the left: $(t, /, -1, t_{aux})$. Thus M is able to reach the -1 starting from q_0 if and only if C halts when it starts from $(s, (0, 0))$.

Next, for the fourth objective, we use an idea from [3]. Create a new machine $M' = (Q', \Sigma, \delta')$, with $Q' = Q \cup \{+, -\}$. States of the form $(q, +)$ act in the same way that in the M machine, and states of the form $(q, -)$ makes the reverse transitions. Each transition not defined for M, switch $+$ by $-$; analogously, transitions not defined in the reverse makes $-$ to become $+$. This makes M' complete.

The fifth goal is attained by modifying M' in only one instruction:
$((q_0, -), /, 0, (q_0, +))$ is switched by $((q_0, -), /, +1, (q_0, +))$.

Thus, q_0 is a blocking state to the left, reachable from the left, for the complete reversible TM M', if and only if C does not halt (reaches the t state) from $(s_0, (0, 0))$. ■

Remark 3. The same machine can be used to proof that the emptiness problem for reversible and complete TM is undecidable, We simply make $q = q_0$ and $q' = t$.:

(E_{RCTM}) Given a reversible and complete TM $M = (Q, \Sigma, \delta)$ and two states $q, q' \in Q$, decide if there exists a configuration $(w, 0, q)$ such that the state q' is attained in finite time.

4 Undecidability of the Surjectivity of the Subshift Associated to a Turing Machine

(Surj) Given a deterministic and complete Turing machine written in quintuples, decide whether its t-shift is surjective.

Theorem 3. *(Surj) is undecidable.*

Proof. We prove the undecidability by reduction from (BSLrtm). Let $M = (Q, \Sigma, \delta)$ be a complete and reversible Turing machine. Let q' be a state that is reachable from the left, and let q, s and s' be such that $\delta(q, s) = (q', s', +1)$. We know that this machine is surjective. We assume that δ is written as a function.

Now let us define M' as M, but with an additional state q_{aux} and the following new instructions:

$$(\forall t \in Q)\ \delta(q_{aux}, t) = (q', t, 0)\ . \tag{7}$$

And changing

$$\delta(q, s) = (q', s', +1) \text{ by } \delta(q, s) = (q_{aux}, s', +1). \tag{8}$$

M' is not surjective, because the configuration (w, i, q_{aux}) has not preimage if $w_{i-1} \neq s'$. So q_{aux} is the unique defective state of M'. In this way, if q' is a blocking state to the left for M, then so is q_{aux} and since it is also reachable from the left, the t-shift of M' is surjective. On the other hand, if this t-shift is surjective, q_{aux} must be blocking. As q_{aux} is only reachable from the left, it must be blocking to the left. This is possible only if q' is blocking to the left. ■

5 Conclusions

Surjectivity of the t-shift of a Turing machine resulted to be not equivalent to the surjectivity of the machine it self. The last property was characterized in terms of another property of the Turing machine, the blocking property of its states: in the absence of the surjectivity of the Turing machine, some of its states must be blocking.

But the blocking property resulted to be undecidable as well as the surjectivity of the t-shift. A rather simple problem in Turing machines can not be decided in t-shift, so we think that others more complicated problems (for example, transitivity) can not be decided too, but this will require a more deep investigation.

References

1. Gajardo, A., Mazoyer, J.: One head machines from a symbolic approach. Theor. Comput. Sci. 370, 34–47 (2007)
2. Gajardo, A., Guillon, P.: Zigzags in Turing Machines. In: Ablayev, F., Mayr, E.W. (eds.) CSR 2010. LNCS, vol. 6072, pp. 109–119. Springer, Heidelberg (2010)

3. 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)
4. Kůrka, P.: On topological dynamics of Turing machines. Theoret. Comput. Sci. 174(1-2), 203–216 (1997)
5. Blondel, V.D., Cassaigne, J., Nichitiu, C.: On the presence of periodic configurations in Turing machines and in counter machines. Theoret. Comput. Sci. 289, 573–590 (2002)
6. Oprocha, P.: On entropy and turing machine with moving tape dynamical model. Nonlinearity 19, 2475–2487 (2006)
7. Kůrka, P.: Topological and Symbolic Dynamics. Société Mathématique de France, Paris, France (2003)
8. Morita, K., Shirasaki, A., Gono, Y.: A 1-tape 2-symbol reversible Turing machine. IEICE Transactions E72-E(3), 223–228 (1989)
9. Morita, K.: Universality of a reversible two-counter machine. Theor. Comput. Sci. 168(2), 303–320 (1996)

Appendix

Construction of the Turing machine shown in the proof or Theorem 2. We will construct a reversible Turing machine $M = (Q, \Sigma, \delta)$ to simulate a 2-reversible counter machine $C = (S, 2, R)$. $Q = S \cup Q_0 \cup S_1 \cup ... \cup S_{|R|}$, where Q_0 is the set of states required for writing "$< \mid >$" on the tape, including q_0, and S_i the set of states needed to simulate the instruction i of R.

$\Sigma = \{<, |, >, 1\} \cup \{0, +\}^2$. The first set recreates the counters on the machine and the second contains auxiliary symbols indicating the status of the counters when reaching a new state.

The transition function is given by a graph, the notation is described in figure 1. It is noteworthy that this machine simulates arbitrary counter machine, so we will describe only generic instructions.

In general, each configuration $(s, (n, m))$ of the counting machine will be represented in the Turing machine by the configuration $(< 1^n|1^m >, 0, s)$. The machine simulates C from configuration $(s_0, (0, 0))$, by writing "$< \mid >$" on the tape. Subsequently, the machine adds and removes 1's from the tape, accordingly to the instructions of C. The machine will work as long as the background is full of 1's (the new visited cells), if it encounter any other symbol, it stops. It is important to note that, before reaching any state of C, the machine replaces the symbol "$<$" by the pair $(d, d') \in \{0, +\}^2$, in order to indicate the *sign* of each counter at the end of each instruction. In this way, the machine knows which of the instructions of the counter machine is the next to be applied. And this makes the reversibility of the counter machine to be inherited by the Turing machine.

Part 1: Initial configuration. The first action is to write "$< \mid >$" in the tape, see figure 2.

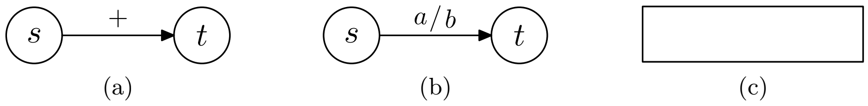


Fig. 1. (a): Instruction $(s, /, +, t)$. (b): Instruction (s, a, b, t). (c): Subroutine.

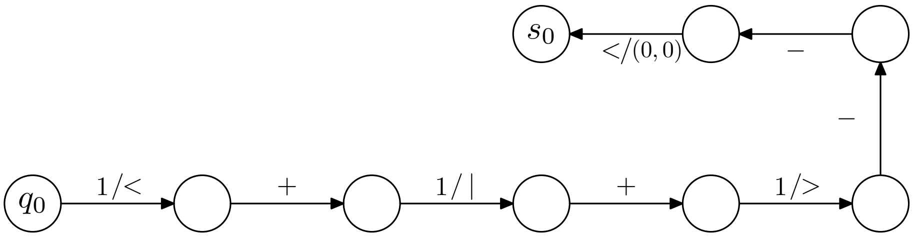


Fig. 2. The routine that writes the sequence "$< \mid >$" in the tape

Part 2: Executing instructions. Let us suppose that the following instructions are in R: $(s,(0,0),i,d,t)$, $(s,(0,+),i',d',t')$, $(s,(+,0),i'',d'',t'')$ y $(s,(+,+),i''',d''',t''')$. They are simulated with the routine depicted in figure 3.

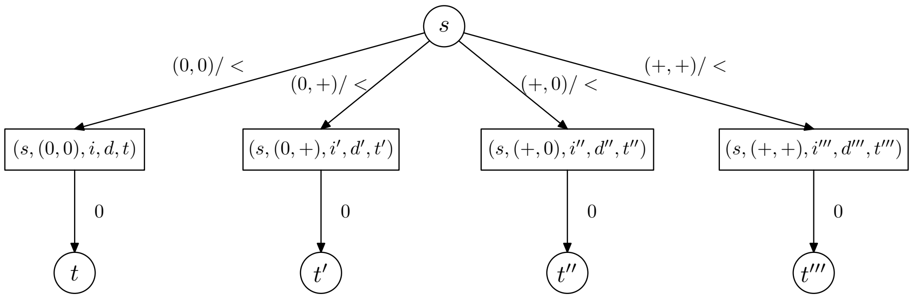


Fig. 3. Depending on the *sign* of the counters, the machine performs the instruction $(s,(0,0),i,d,t)$, $(s,(0,+),i',d',t')$, $(s,(+,0),i'',d'',t'')$ or $(s,(+,+),i''',d''',t''')$

Part 3: Addition/Subtraction. Each sub-routine uses an exclusive set of states. There are several cases, depending on the *sign* of each counter, but they are all similar, thus we present only two examples in figures 4 and 5.

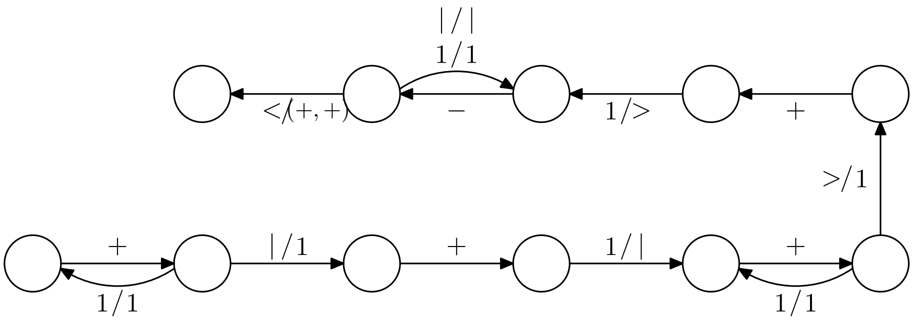


Fig. 4. Sub-routine corresponding to instruction $(s, (+,+), 1, +, t''')$, it adds one unit to counter 1, assuming counter 2 non empty

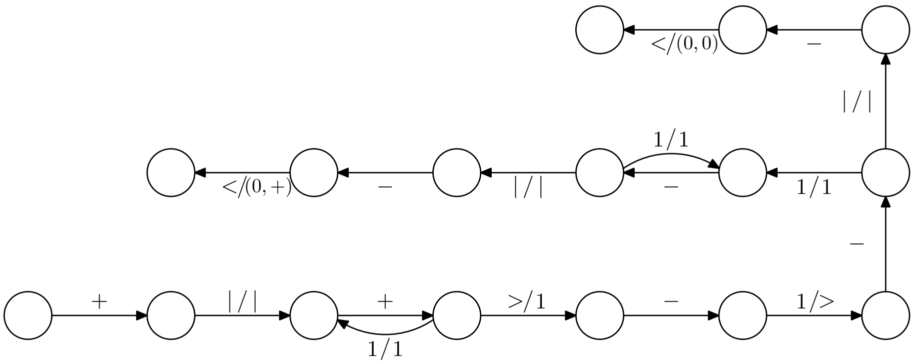


Fig. 5. Sub-routine corresponding to instruction $(s, (0,+), 2, -, t')$, it subtracts from counter 2, assuming counter 1 empty

Isomorphic Interpreters from Logically Reversible Abstract Machines

Roshan P. James and Amr Sabry

School of Informatics and Computing, Indiana University
{rpjames,sabry}@indiana.edu

Abstract. In our previous work, we developed a reversible programming language and established that every computation in it is a (partial) isomorphism that is reversible and that preserves information. The language is founded on type isomorphisms that have a clear categorical semantics but that are awkward as a notation for writing actual programs, especially recursive ones. This paper remedies this aspect by presenting a systematic technique for developing a large and expressive class of reversible recursive programs, that of logically reversible small-step abstract machines. In other words, this paper shows that once we have a logically reversible machine in a notation of our choice, expressing the machine as an isomorphic interpreter can be done systematically and does not present any significant conceptual difficulties. Concretely, we develop several simple interpreters over numbers and addition, move on to tree traversals, and finish with a meta-circular interpreter for our reversible language. This gives us a means of developing large reversible programs with the ease of reasoning at the level of a conventional small-step semantics.

1 Introduction

In recent papers [3,7], we developed a pure reversible model of computation that is obtained from the type isomorphisms and categorical structures that underlie models of linear logic and quantum computing and that treats information as a linear resource that can neither be erased nor duplicated. From a programming perspective, our model gives rise to a pure (with no embedded computational effects such as assignments) reversible programming language Π^o based on *partial isomorphisms*. In more detail, in the recursion-free fragment of Π^o, every program witnesses an isomorphism between two finite types; in the full language with recursion, some of these isomorphisms may be partial, i.e., may diverge on some inputs.

In this paper, we investigate the practicality of writing large recursive programs in Π^o. Specifically, we assume that we are given some reversible recursive algorithm expressed as a *reversible abstract machine*, and we show via a number of systematic steps, how to develop a Π^o program from that abstract machine. Our choice of starting from reversible abstract machines is supported by the following observations:

– it is an interesting enough class of reversible programs: researchers have invested the effort in manually designing such machines, e.g., the SE(M)CD machine [8], and the SECD-H [6];
– every reversible interpreter can be realized as such a machine: this means that our class of programs includes self-interpreters which are arguably a measure of the strength of any reversible language [2,12,13];
– general recursive programs can be systematically transformed to abstract machines: the technique is independent of reversible programs and consists of transforming general recursion to tail recursion and then applying *fission* to split the program into a driver and a small-step machine [5,9].

To summarize, we assume we are given some reversible abstract machine and we show how to derive a Π^o program that implements the semantics of the machine. Our derivation is systematic and expressive. In particular, Π^o can handle machines with stuck states because it is based on partial isomorphisms. We illustrate our techniques with simple machines that do bounded iteration on numbers, tree traversals, and a meta-circular interpreter for Π^o.

2 Review of the Reversible Language: Π^o

We briefly review the reversible language Π^o introduced in our previous work [3,7]. The set of types includes the empty type 0, the unit type 1, sum types $b_1 + b_2$, product types $b_1 \times b_2$, and recursive types $\mu x.b$. The set of values v includes () which is the only value of type 1, *left* v and *right* v which inject v into a sum type, (v_1, v_2) which builds a value of product type, and $\langle v \rangle$ which is used to construct recursive values. There are no values of type 0:

$$\begin{aligned} \mathit{value\ types}, b &::= 0 \mid 1 \mid b + b \mid b \times b \mid x \mid \mu x.b \\ \mathit{values}, v &::= () \mid \mathit{left}\ v \mid \mathit{right}\ v \mid (v, v) \mid \langle v \rangle \end{aligned}$$

The expressions of Π^o are witnesses to the following type isomorphisms:

$$\begin{array}{rcl} \mathit{zeroe} : & 0 + b \leftrightarrow b & : \mathit{zeroi} \\ \mathit{swap}^+ : & b_1 + b_2 \leftrightarrow b_2 + b_1 & : \mathit{swap}^+ \\ \mathit{assocl}^+ : & b_1 + (b_2 + b_3) \leftrightarrow (b_1 + b_2) + b_3 & : \mathit{assocr}^+ \\ \mathit{unite} : & 1 \times b \leftrightarrow b & : \mathit{uniti} \\ \mathit{swap}^\times : & b_1 \times b_2 \leftrightarrow b_2 \times b_1 & : \mathit{swap}^\times \\ \mathit{assocl}^\times : & b_1 \times (b_2 \times b_3) \leftrightarrow (b_1 \times b_2) \times b_3 & : \mathit{assocr}^\times \\ \mathit{distrib}_0 : & 0 \times b \leftrightarrow 0 & : \mathit{factor}_0 \\ \mathit{distrib} : & (b_1 + b_2) \times b_3 \leftrightarrow (b_1 \times b_3) + (b_2 \times b_3) & : \mathit{factor} \\ \mathit{fold} : & b[\mu x.b/x] \leftrightarrow \mu x.b & : \mathit{unfold} \end{array}$$

Each line of the above table introduces one or two combinators that witness the isomorphism in the middle. Collectively the isomorphisms state that the structure $(b, +, 0, \times, 1)$ is a *commutative semiring*, i.e., that each of $(b, +, 0)$ and $(b, \times, 1)$ is a commutative monoid and that multiplication distributes over addition. The last isomorphism witnesses the equivalence of a value of recursive type with all its "unrollings." The isomorphisms are extended to form a congruence relation by adding the following constructors that witness equivalence and compatible closure. In addition, the language includes a *trace* operator to express looping:

$$\frac{}{id : b \leftrightarrow b} \quad \frac{c : b_1 \leftrightarrow b_2}{sym\ c : b_2 \leftrightarrow b_1} \quad \frac{c_1 : b_1 \leftrightarrow b_2 \quad c_2 : b_2 \leftrightarrow b_3}{c_1 \fatsemi c_2 : b_1 \leftrightarrow b_3}$$

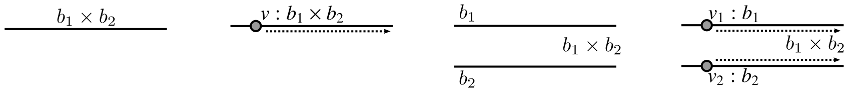
$$\frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 + c_2 : b_1 + b_2 \leftrightarrow b_3 + b_4} \quad \frac{c_1 : b_1 \leftrightarrow b_3 \quad c_2 : b_2 \leftrightarrow b_4}{c_1 \times c_2 : b_1 \times b_2 \leftrightarrow b_3 \times b_4} \quad \frac{c : b_1 + b_2 \leftrightarrow b_1 + b_3}{trace\ c : b_2 \leftrightarrow b_3}$$

Adjoint. An important property of the language is that every combinator c has an adjoint $c^\dagger$ that reverses the action of c. This is evident by construction for the primitive isomorphisms. For the closure combinators, the adjoint is homomorphic except for the case of sequencing in which the order is reversed, i.e., $(c_1 \fatsemi c_2)^\dagger = (c_2{}^\dagger) \fatsemi (c_1{}^\dagger)$.

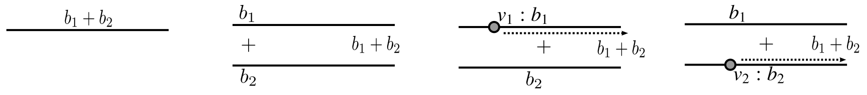
Graphical Language. Following the tradition for computations in monoidal categories [10], we present a graphical notation that conveys the semantics of Π^o. The general idea of the graphical notation is that combinators are modeled by "wiring diagrams" or "circuits" and that values are modeled as "particles" or "waves" that may appear on the wires. Evaluation therefore is modeled by the flow of waves and particles along the wires.

– The simplest sort of diagram is the $id : b \leftrightarrow b$ combinator which is simply represented as a wire labeled by its type b, as shown below on the left. In more complex diagrams, if the type of a wire is obvious from the context, it may be omitted. When tracing a computation, one might imagine a value v of type b on the wire, as shown below on the right:

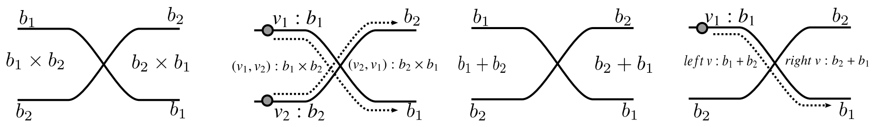
– The product type $b_1 \times b_2$ may be represented using either one wire labeled $b_1 \times b_2$ or two parallel wires labeled b_1 and b_2. In the case of products represented by a pair of wires, when tracing execution using particles, one should think of one particle on each wire or alternatively as in folklore in the literature on monoidal categories as a "wave:"



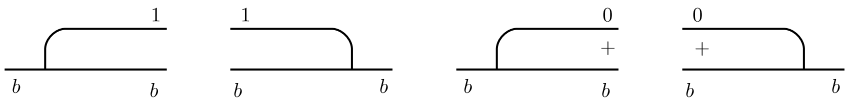
– Sum types may similarly be represented by one wire or using parallel wires with a + operator between them. When tracing the execution of two additive wires, a value can reside on only one of the two wires:



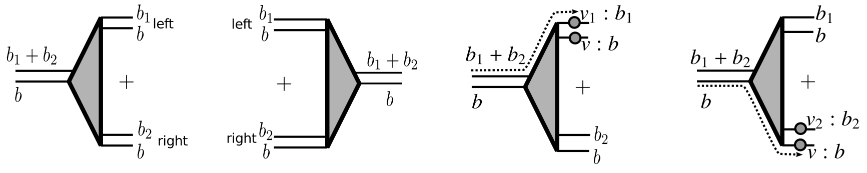
- Associativity is implicit in the graphical language. Thus, three parallel wires may represent $b_1 \times (b_2 \times b_3)$ or $(b_1 \times b_2) \times b_3$.
- Commutativity is represented by crisscrossing wires:



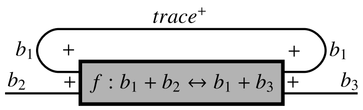
- The morphisms that witness that 0 and 1 are the additive and multiplicative units are represented as shown below. Note that since there is no value of type 0, there can be no particle on a wire of type 0. Also since the monoidal units can be freely introduced and eliminated, in many diagrams they are omitted and dealt with explicitly only when they are of special interest:



- Distributivity and factoring are interesting because they represent interactions between sum and pair types. Distributivity should essentially be thought of as a multiplexer that redirects the flow of $v : b$ depending on what value inhabits the type $b_1 + b_2$, as shown below. Factoring is the corresponding adjoint operation:



- Operations *fold* and *unfold* are specific to each recursive data type and are drawn as triangular boxes. For instance, Sec. 2.2 shows *fold*/*unfold* isomorphisms for numbers, $nat = \mu x.(1 + x)$.
- The *trace* operation is drawn as a looped circuit, where the traced type b_1 is shown as flowing backwards:

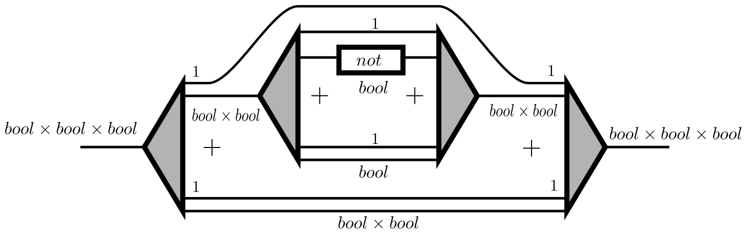


2.1 Universality

As an example, consider the implementation of the Toffoli gate below, which establishes that Π^o is complete for combinational circuits. The Toffoli gate takes three boolean inputs: if the first two inputs are *true* then the third is negated. This encoding uses the type $1 + 1$ as *bool* and values *left* () and *right*() as *true* and *false*. Boolean negation, $\mathit{not} : \mathit{bool} \leftrightarrow \mathit{bool}$, is simply swap^+.

Even though Π^o lacks conditional expressions, they are expressible using the distributivity laws. Given any combinator $c : b \leftrightarrow b$, we can construct a combinator called $\mathit{if}_c : \mathit{bool} \times b \leftrightarrow \mathit{bool} \times b$ in terms of c, which behaves like a one-armed if-expression. If the supplied boolean is *true* then the combinator c is used to transform the value of type b. If the boolean is *false* then the value of type b remains unchanged. We can write down the combinator for if_c in terms of c as $\mathit{distrib} \fatsemi ((\mathit{id} \times c) + \mathit{id}) \fatsemi \mathit{factor}$.

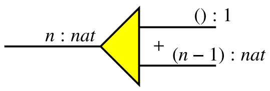
Given if_c, constructing the Toffoli gate is straightforward. If we choose c to be *not* we get $\mathit{if}_{\mathit{not}}$ which is controlled-not, $\mathit{cnot} : \mathit{bool} \times \mathit{bool} \leftrightarrow \mathit{bool} \times \mathit{bool}$. Further, $\mathit{if}_{\mathit{cnot}}$ is the required Toffoli gate, $\mathit{toffoli} : \mathit{bool} \times (\mathit{bool} \times \mathit{bool}) \leftrightarrow \mathit{bool} \times (\mathit{bool} \times \mathit{bool})$, and is drawn below.

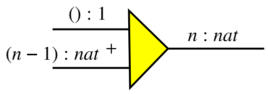


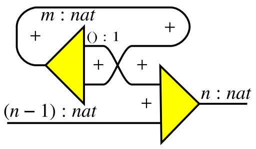
Previous work [7, Sec. 5] establishes that Π^o is Turing complete by showing the compilation of a Turing complete language — a first-order language with numbers and iteration — to Π^o.

2.2 Numeric Operations

We encode numbers, $\mathit{nat} = 0 \mid n + 1$, using the recursive Π^o type $\mu x.1 + x$. The $\mathit{fold}/\mathit{unfold}$ isomorphisms for *nat* are $\mathit{unfold} : \mathit{nat} \leftrightarrow 1 + \mathit{nat} : \mathit{fold}$.







The combinator on the left denotes the *unfold* isomorphism, which works as follows: If the number n is zero, the output is the top branch which has type 1. If the number is non-zero, the output is on the bottom branch and has value $n - 1$. The combinator in the middle is its dual *fold* isomorphism.

As explained in previous work [7, Sec. 4.2], numeric addition and subtraction can be expressed in Π^o as partial isomorphisms: The combinator on the right, $add_1 : nat \leftrightarrow nat$, acts like the successor function returning $n+1$ for all inputs n. Its adjoint, sub_1, obtained by flipping the diagram right to left, is a partial map that diverges on input 0 and returns $n-1$ for all other inputs n.

3 Simple Bounded Number Iteration

We illustrate the main concepts and constructions using two simple examples that essentially count n steps. The first machine does nothing else; the second uses this counting ability to add two numbers.

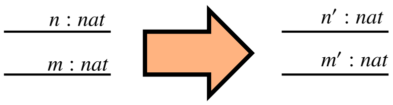
3.1 Counting

The first machine is defined as follows:

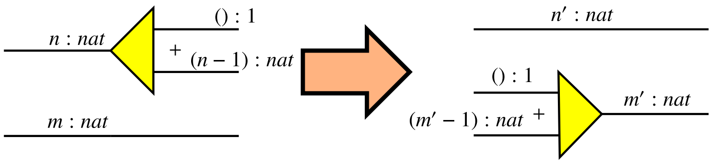
$$\begin{array}{ll} \mathit{Numbers}, n, m = 0 \mid n+1 & \mathit{Start\ state} = \langle n, 0\rangle \\ \mathit{Machine\ states} = \langle n, n\rangle & \mathit{Stop\ State} = \langle 0, n\rangle \end{array}$$
$$\langle n+1, m\rangle \mapsto \langle n, m+1\rangle$$

The machine is started with a number n in the first position and 0 in the second. Each reduction step decrements the first number and increments the second. The machine stops when the first number reaches 0, thereby taking exactly n steps. For example, if the machine is started in the configuration $\langle 3,0\rangle$, it would take exactly 3 steps to reach the final configuration: $\langle 3,0\rangle \mapsto \langle 2,1\rangle \mapsto \langle 1,2\rangle \mapsto \langle 0,3\rangle$. Dually, since the machine transition is clearly reversible, the machine can execute backwards from the final configuration to reach the initial configuration in also 3 steps.

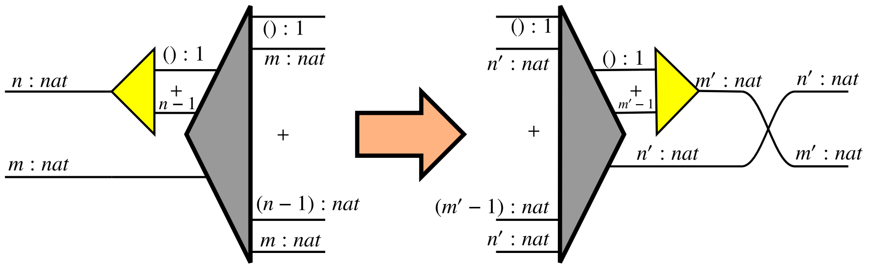
Our goal is to implement this abstract machine in Π^o. We start by writing a combinator $c : nat \times nat \leftrightarrow nat \times nat$ that executes one step of the machine transition. The combinator, when iterated, should map $(n,0)$ to $(0,n)$ by precisely mimicking the steps of the abstract machine. Let us analyze this desired combinator c in detail starting from its interface:



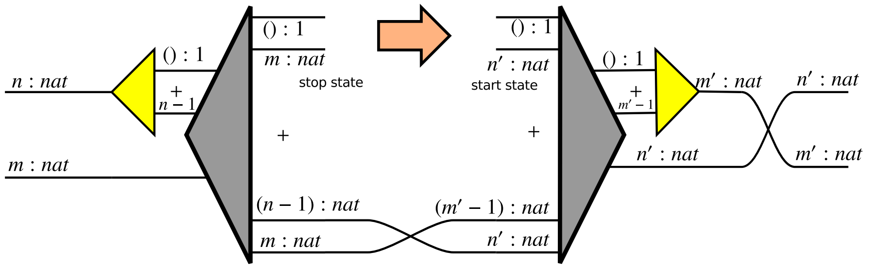
The reduction step of the machine examines the first nat and reconstructs the second nat. In other words, the first nat is unfolded to examine its structure and the second nat is folded to reconstruct it:



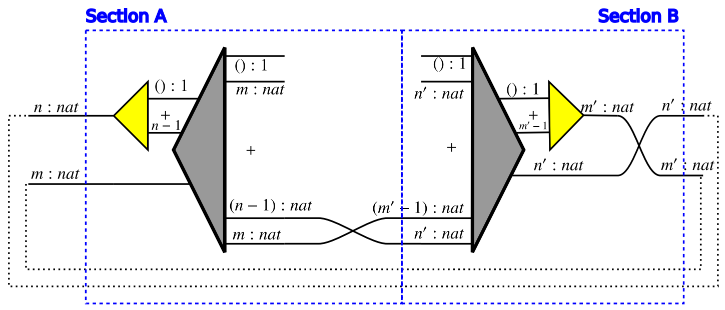
We now use distribution to isolate each possible machine state:



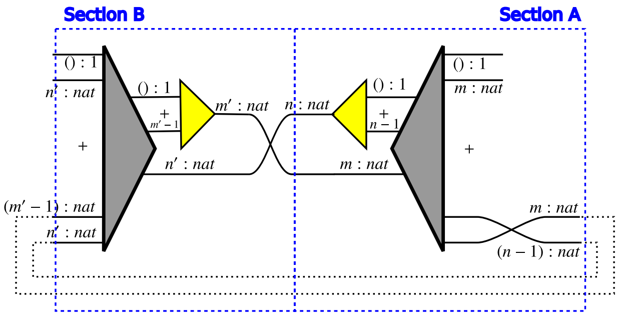
At this point, it is clear that the machine's transition consists connecting the $n-1$ input wire to n' on the output side and the m input wire to $m'-1$ on the output side:



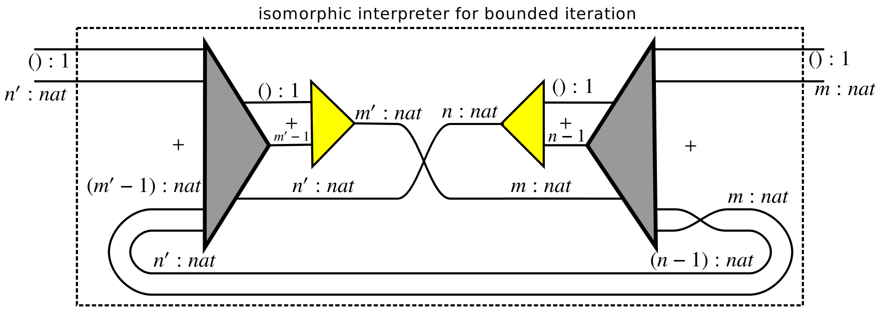
We are now close to the completion of the interpreter. The branch labeled $((), m)$ corresponds to the machine state $\langle 0, m\rangle$ which is the stop state of the machine. Similarly the branch labeled $((), n')$ corresponds to the start state of the machine. The last step in the construction involves introducing a *trace* operation for iterating the small step realized so far:



Sliding sections A and B past each other while maintaining the connectivity of the wires, exposes that the middle connections were really recursive calls to the machine transition:



The completed interpreter is given below:



3.2 Steps of the Construction

Although trivial, the previous example captures the fundamental aspects of our construction. In general, our starting point is an abstract machine in which every rewrite step is logically reversible. The formal criterion of logical reversibility in this setting is captured by Abramsky's *bi-orthogonal automata* [1, Sec. 2] which is summarized as:

1. Machine states are described as tuples whose components are algebraic data types. In the above example we used only the *nat* data type and tuples of the form $\langle n, m\rangle$.
2. Machines must have distinguishable start and stop states. There may be more than one valid start state and more than one valid stop state, however.
3. Each valid machine state must match a unique pattern on the left-hand side and right-hand side of the '$\mapsto$' relation.
4. Every reduction step must be computable (in Π^o), must not drop or duplicate pattern variables and must be logically reversible — i.e. it must be possible to reduce from right to left.

Given such a machine the process of encoding the machine in Π^o consists of the following steps:

1. Expand the input to expose enough structure to distinguish the left-hand side of each machine state;

2. Expand the output to expose enough structure to distinguish the right-hand side of each machine state;
3. Shuffle matching input terms to output terms, inserting any appropriate mediating computations.
4. Slide the two sections to expose the start and stop state and introduce a *trace* to iterate the construction.

3.3 Adder

Let us apply these steps to the slightly more interesting example of an adder:

$$\begin{array}{rlrl} \mathit{Numbers}, n, m, p &= 0 \mid n+1 & \qquad\qquad \mathit{Start\ state} &= \langle n, n, 0\rangle \\ \mathit{Machine\ states} &= \langle n, n, n\rangle & \mathit{Stop\ State} &= \langle 0, n, n\rangle \end{array}$$

$$\langle n+1, p, m\rangle \mapsto \langle n, p+1, m+1\rangle$$

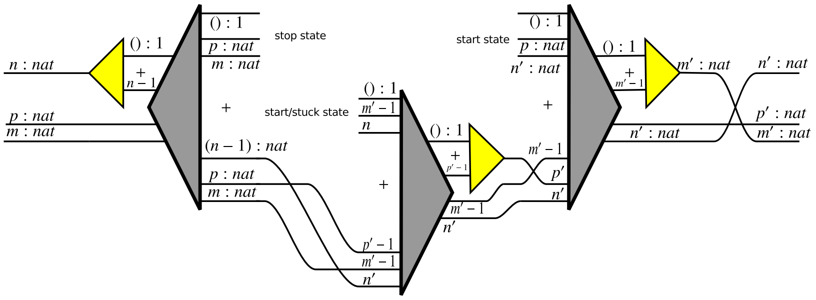
The idea of the machine is to start with 3 numbers: the two numbers to add and an accumulator initialized to 0. Each step of the machine, decrements one of the numbers and increments the second number and the accumulator. For example, $\langle 3,4,0\rangle \mapsto \langle 2,5,1\rangle \mapsto \langle 1,6,2\rangle \mapsto \langle 0,7,3\rangle$. In general, the sum of the two numbers will be in the second component, and the last component is supposed to record enough information to make the machine reversible.

A closer look however reveals that the machine defines a *partial* isomorphism: not all valid final states can be mapped to valid start states. Indeed consider the configuration $\langle 0,2,3\rangle$ which is a valid final state. Going backwards, the transitions start as follows $\langle 0,2,3\rangle \mapsto \langle 1,1,2\rangle \mapsto \langle 2,0,1\rangle$ at which point, the machine gets stuck at a state that is not a valid start state. This is a general problem that we discuss below in detail.

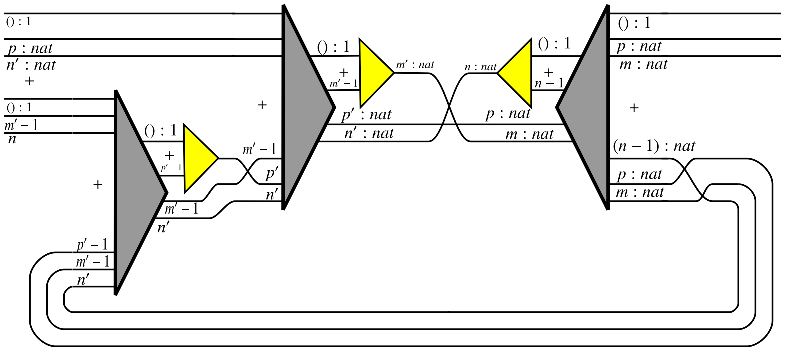
Stuck States. The type systems of most languages are not expressive enough to encode the precise domain and range of a function. For example, in most typed languages, division by zero is considered type-correct and the runtime system is required to deal with such an error. A stuck state of an abstract machine is just a manifestation of this general problem. The common solutions are:

– *Use a more expressive type system.* One could augment Π^o with a richer type system that distinguishes non-zero numbers from those that can be zero, thereby eliminating the $sub_1\ 0$ situation entirely. Similarly, in the metacircular interpreter in Sec. 4.3, one could use generalized abstract data types (GADTs [4,11]) to eliminate the stuck states.
– *Diverge.* Another standard approach in dealing with stuck states is to make the machine diverge or leave the output undefined or unobservable in some way. If the specific case of the machine above, we can use the primitive add_1 whose dual sub_1 is undefined when applied to 0 (see Sec. 2.2).

– *Stop the machine.* Alternatively, we can consider the stuck state as a valid final state. In the case of the example above, we would treat states of the form $\langle n, 0, n\rangle$ as valid stop states for reverse execution. This gives us two valid start states in the case of forward execution and makes the isomorphism total. If we chose this approach, the machine would look as shown below by the end of *step* 3:



Note that there are indeed two "start states" in the interpreter. As before, we can slide the two sides of the diagram and tie the knot using *trace* to get the desired interpreter:



4 Advanced Examples

We show the generality of our construction by applying it to three non-trivial examples: a tree traversal, parity translation of numbers and a meta-circular interpreter for Π^o.

4.1 Tree Traversal

The type of binary trees we use is $\mu x.(nat + x \times x)$, i.e., binary trees with no information at the nodes and with natural numbers at the leaves. To define the abstract machine, we need a notion of *tree contexts* to track which subtree is currently being explored. The definitions are shown on the left:

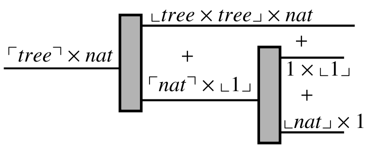
$$\begin{array}{rl} \mathit{Tree}, t = & L\ n \mid N\ t\ t \\ \mathit{Tree\ Contexts}, c = & \Box \mid \mathit{Lft}\ c\ t \mid \mathit{Rgt}\ t\ c \end{array}$$

$$\begin{array}{rl} \mathit{Machine\ states} = & \langle t, c\rangle \mid \{c, t\} \\ \mathit{Start\ state} = & \langle t, \Box\rangle \\ \mathit{Stop\ State} = & \{\Box, t\} \end{array} \qquad \begin{array}{rcl} \langle L\ n, c\rangle & \mapsto & \{c, L\ (n+1)\} \\ \langle N\ t_1\ t_2, c\rangle & \mapsto & \langle t_1, \mathit{Lft}\ c\ t_2\rangle \\ \{\mathit{Lft}\ c\ t_2, t_1\} & \mapsto & \langle t_2, \mathit{Rgt}\ t_1\ c\rangle \\ \{\mathit{Rgt}\ t_1\ c, t_2\} & \mapsto & \{N\ t_1\ t_2, c\} \end{array}$$

The reduction rules on the right traverse a given tree and increment every leaf value. The machine here is a little richer than the ones dealt with previously. In particular, we have two types of machine states $\langle t, c\rangle$ and $\{t, c\}$. The first of these corresponds to walking down a tree, building up the context in the process. The second corresponds to reconstructing the tree from the context and also switching focus to any unexplored subtrees in the process. There are also two syntactic categories to deal with (trees and tree contexts) where previously we only had numbers. The *fold* and *unfold* isomorphisms that we need for trees and tree contexts are:

$$\begin{array}{lcl} \mathit{unfold}_t : & t \leftrightarrow n + t \times t & : \mathit{fold}_t \\ \mathit{unfold}_c : & c \leftrightarrow 1 + c \times t + t \times c & : \mathit{fold}_c \end{array}$$

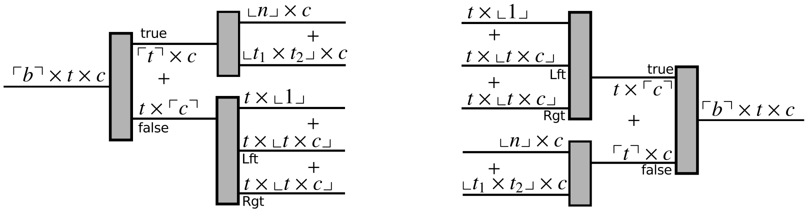
New Notation. To make the diagrams easier to understand, we introduce a syntactic convenience which combines the first steps of the construction that consist of *fold* / *unfold* and *distribute* / *factor*. We collectively represent these steps using thin vertical rectangle. Also we will introduce the convention that the component that is being expanded (or constructed) will be marked by using $\ulcorner\urcorner$ and the components that are being generated (or consumed) will be marked by $\llcorner\lrcorner$. For example, given a value of type $\mathit{tree} \times \mathit{nat}$, the diagram below shows how to first expand the *tree* component and then in one of the generated branches, expand the *nat* component:



We can now apply our construction. The first step to developing the isomorphic interpreter is to recognize that the two possible kinds of machine states simply hide an implicit *bool*. We make this explicit:

$$\begin{array}{rl} \mathit{Machine\ states} = & \langle \mathit{bool}, t, c\rangle \\ \\ \mathit{Start\ state} = & \langle \mathit{true}, t, \Box\rangle \\ \mathit{Stop\ state} = & \langle \mathit{false}, t, \Box\rangle \end{array} \qquad \begin{array}{rcl} \langle \mathit{true}, L\ n, c\rangle & \mapsto & \langle \mathit{false}, L\ (n+1), c\rangle \\ \langle \mathit{true}, N\ t_1\ t_2, c\rangle & \mapsto & \langle \mathit{true}, t_1, \mathit{Lft}\ c\ t_2\rangle \\ \langle \mathit{false}, t_1, \mathit{Lft}\ c\ t_2\rangle & \mapsto & \langle \mathit{true}, t_2, \mathit{Rgt}\ t_1\ c\rangle \\ \langle \mathit{false}, t_2, \mathit{Rgt}\ t_1\ c\rangle & \mapsto & \langle \mathit{false}, c, N\ t_1\ t_2\rangle \end{array}$$

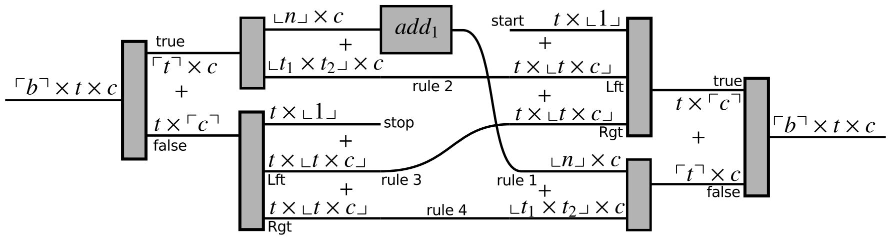
We start examining the machine components as before:



On the input side, for *true* states, we have expanded the tree component and for *false* states we have expanded the tree contexts. We have done the opposite on the output side exactly matching up what the abstract machine does. One thing to note is that we dropped the 1 introduced by expanding the *bool* and instead just labeled the *true* and *false* branches.

We are ready to start connecting the machine states corresponding to the reductions that we would like:

1. When we encounter a leaf we would like to increment its value and move to the corresponding *false* machine state. For the sake of simplicity, in this interpreter we won't be concerned with exposing stuck states: we simply use add_1 whose adjoint sub_1 diverges when applied to 0.
2. For all the other reduction rules, it is a straightforward mapping of related states following the reduction rules. For readability, we have annotated the diagram below with the names of the reduction rules and we have included subscripts on the various ts to indicate any implicit swaps that should be inserted.



This essentially completes the construction of the (partially) isomorphic tree-traversal interpreter, except for the final step which slides the input and output sides.

4.2 Parity Translation

Deriving Π^o combinators from abstract machines can sometimes be used as an efficient indirect way to program in Π^o. It is well known that every number n can be represented as $2a + 0$ or $2a + 1$, depending on whether it is odd or even. The later can be represented by the algebraic data type $par = odd \mid even \mid A\ par$

where the nesting of A constructor indicates the value of a. For example $0 = even, 1 = odd, 2 = A\ even, 3 = A\ odd, 4 = A\ (A\ even)$ etc.

Say we wish to build a Π^o combinator to map *nat* to its parity encoded version *par*. While the type *par* is expressible as $\mu x.(1+1)+x$ in Π^o, it is not a fixed unfolding of $nat = \mu x.1 + x$ and hence it is not immediately apparent how such a combinator can be constructed.

Instead of directly programming in Π^o however, one can derive the required $nat \leftrightarrow par$ combinator by first constructing an abstract machine that maps *nat* to *par* and then translating it to Π^o. We first express *nat* and *par* algebraically and then develop a logically reversible abstract machine:

$$\begin{aligned} Numbers, n, m &= 0 \mid n+1 \\ Parity, par &= even \mid odd \mid A\ par \end{aligned}$$

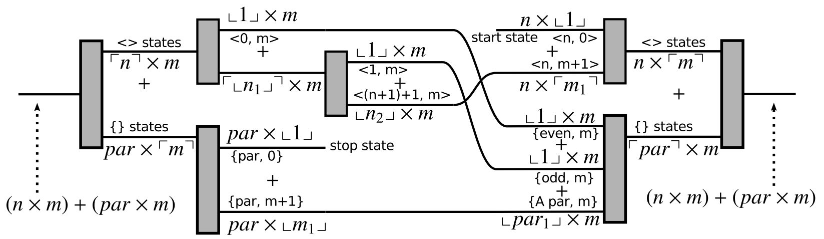
Machine States:

$$\begin{aligned} Machine\ states &= \langle nat, nat \rangle \mid \{parity, nat\} \\ Start\ state &= \langle n, 0 \rangle \\ Stop\ state &= \{par, 0\} \end{aligned}$$

Machine Reductions:

$$\begin{aligned} \langle 0, m \rangle &\mapsto \{even, m\} \\ \langle 1, m \rangle &\mapsto \{odd, m\} \\ \langle (n+1)+1, m \rangle &\mapsto \langle n, m+1 \rangle \\ \{par, m+1\} &\mapsto \{A\ par, m\} \end{aligned}$$

The derivation of the Π^o combinator is straightforward and proceeds as before. The partial combinator representing the wiring of machine reductions is shown below.



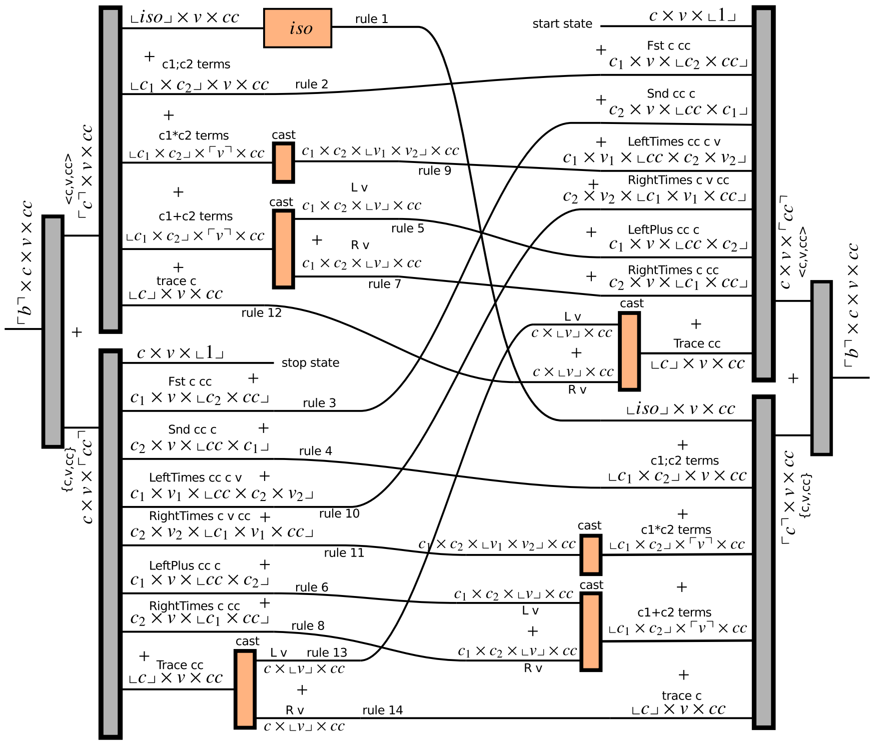
We thank Fritz Henglein for the motivation underlying this example.

4.3 A Π^o Interpreter

In the final construction we present a meta-circular interpreter for Π^o written in Π^o. This is a a non-trivial abstract machine with several cases, but the derivation follows in exactly the same way as before. Here is a logically reversible small-step abstract machine for Π^o and the derived isomorphic interpreter with the reductions labeled.

$$
\begin{array}{rcl}
\mathit{Combinators}, c & = & \mathit{iso} \mid c \fatsemi c \mid c \times c \mid c + c \mid \mathit{trace}\ c \\
\mathit{Combinator\ Contexts}, cc & = & \Box \mid \mathit{Fst}\ cc\ c \mid \mathit{Snd}\ c\ cc \\
 & \mid & \mathit{LeftTimes}\ cc\ c\ v \mid \mathit{RightTimes}\ c\ v\ cc \\
 & \mid & \mathit{LeftPlus}\ cc\ c \mid \mathit{RightPlus}\ c\ cc \mid \mathit{Trace}\ cc \\
\mathit{Values}, v & = & () \mid (v, v) \mid L\ v \mid R\ v
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{Machine\ states} & = & \langle c, v, cc \rangle \mid \{c, v, cc\} & \\
\mathit{Start\ state} & = & \langle c, v, \Box \rangle & \\
\mathit{Stop\ State} & = & \{c, v, \Box\} & \\
\langle \mathit{iso}, v, cc \rangle & \mapsto & \{\mathit{iso}, \mathit{iso}(v), cc\} & \mathit{rule}\ 1 \\
\langle c_1 \fatsemi c_2, v, cc \rangle & \mapsto & \langle c_1, v, \mathit{Fst}\ cc\ c_2 \rangle & \mathit{rule}\ 2 \\
\{c_1, v, \mathit{Fst}\ cc\ c_2\} & \mapsto & \langle c_2, v, \mathit{Snd}\ c_1\ cc \rangle & \mathit{rule}\ 3 \\
\{c_2, v, \mathit{Snd}\ c_1\ cc\} & \mapsto & \{c_1 \fatsemi c_2, v, cc\} & \mathit{rule}\ 4 \\
\langle c_1 + c_2, L\ v, cc \rangle & \mapsto & \langle c_1, v, \mathit{LeftPlus}\ cc\ c_2 \rangle & \mathit{rule}\ 5 \\
\{c_1, v, \mathit{LeftPlus}\ cc\ c_2\} & \mapsto & \{c_1 + c_2, L\ v, cc\} & \mathit{rule}\ 6 \\
\langle c_1 + c_2, R\ v, cc \rangle & \mapsto & \langle c_2, v, \mathit{RightPlus}\ c_1\ cc \rangle & \mathit{rule}\ 7 \\
\{c_2, v, \mathit{RightPlus}\ c_1\ cc\} & \mapsto & \{c_1 + c_2, R\ v, cc\} & \mathit{rule}\ 8 \\
\langle c_1 \times c_2, (v_1, v_2), cc \rangle & \mapsto & \langle c_1, v_1, \mathit{LeftTimes}\ cc\ c_2\ v_2 \rangle & \mathit{rule}\ 9 \\
\{c_1, v_1, \mathit{LeftTimes}\ cc\ c_2\ v_2\} & \mapsto & \langle c_2, v_2, \mathit{RightTimes}\ c_1\ v_1\ cc \rangle & \mathit{rule}\ 10 \\
\{c_2, v_2, \mathit{RightTimes}\ c_1\ v_1\ cc\} & \mapsto & \{c_1 \times c_2, (v_1, v_2), cc\} & \mathit{rule}\ 11 \\
\langle \mathit{trace}\ c, v, cc \rangle & \mapsto & \langle c, R\ v, \mathit{Trace}\ cc \rangle & \mathit{rule}\ 12 \\
\{c, L\ v, \mathit{Trace}\ cc\} & \mapsto & \langle c, L\ v, \mathit{Trace}\ cc \rangle & \mathit{rule}\ 13 \\
\{c, R\ v, \mathit{Trace}\ cc\} & \mapsto & \{\mathit{trace}\ c, R\ v, cc\} & \mathit{rule}\ 14
\end{array}
$$



The derivation of the Π^o interpreter, while tedious, is entirely straightforward. A couple of points however need clarification:

1. The definition here shows only the handling of the composition combinators, which are the interesting cases. All the primitive isomorphisms are hidden

in the $iso(v)$ application in *rule* 1. This should be read as “transform v according to the primitive isomorphism *iso*.”

2. Stuck states in this machine correspond to runtime values not matching their expected types. They can be handled as described in Sec. 3.3 using GADTs (which eliminates them entirely), divergence, or adding extra halting states. We have abstracted from this choice and marked the relevant cases with combinators labeled *cast*.

5 Conclusion

We have developed a technique for the systematic derivation of Π^o programs from logically reversible small-step abstract machines. Since techniques for devising small-step interpreters are well known, this allows for the direct development of a large class of Π^o programs. We have demonstrated the effectiveness of this approach by deriving a meta-circular interpreter for Π^o.

Acknowledgments. This material is based upon work supported by the National Science Foundation under Grant No. 1116725.

References

1. Abramsky, S.: A structural approach to reversible computation. Theor. Comput. Sci. 347, 441–464 (2005)
2. Axelsen, H.B., Glück, R.: What Do Reversible Programs Compute? In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011)
3. Bowman, W.J., James, R.P., Sabry, A.: Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In: Reversible Computation (2011)
4. Cheney, J., Hinze, R.: First-class phantom types. Tech. rep., Cornell Univ. (2003)
5. Danvy, O.: From Reduction-Based to Reduction-Free Normalization. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) AFP 2008. LNCS, vol. 5832, pp. 66–164. Springer, Heidelberg (2009)
6. Huelsbergen, L.: A logically reversible evaluator for the call-by-name lambda calculus. InterJournal Complex Systems 46 (1996)
7. James, R.P., Sabry, A.: Information effects. In: POPL, pp. 73–84. ACM (2012)
8. Kluge, W.E.: A Reversible SE(M)CD Machine. In: Koopman, P., Clack, C. (eds.) IFL 1999. LNCS, vol. 1868, pp. 95–113. Springer, Heidelberg (2000)
9. Rendel, T., Ostermann, K.: Invertible syntax descriptions: unifying parsing and pretty printing. In: Symposium on Haskell, pp. 1–12. ACM (2010)
10. Selinger, P.: A survey of graphical languages for monoidal categories. In: New Structures for Physics. Lecture Notes in Physics, pp. 289–355. Springer (2011)
11. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. In: POPL, pp. 224–235. ACM (2003)
12. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Conference on Computing Frontiers, pp. 43–54. ACM (2008)
13. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: PEPM, pp. 144–153. ACM (2007)

Synthesizing Loops for Program Inversion

Cong Hou[1], Daniel Quinlan[2], David Jefferson[2],
Richard Fujimoto[1], and Richard Vuduc[1]

[1] Georgia Institute of Technology
[2] Lawrence Livermore National Laboratory
hou_cong@gatech.edu, {dquinlan,jefferson6}@llnl.gov,
{fujimoto,richie}@cc.gatech.edu

Abstract. We propose a new automatic program inversion method for imperative programs that contain loops. In particular, given a loop that produces an output state given a particular input state, our method can synthesize an inverse loop that reconstructs the input state given the original loop's output state. The synthesis process consists of two major components: (a) building the inverse loop's body, and (b) building the inverse loop's predicate. Our method works for all natural loops, including those that take early exits (e.g., via breaks, gotos, returns). This work extends a program analysis and synthesis framework, called Backstroke[1], that we developed in prior work.

Keywords: Program Inversion, Program Synthesis, Compilers.

1 Introduction

We consider the problem of synthesizing *program inverses*. That is, given a program P with input state I and output state O, its *inverse* or *reverse program*, P^-, produces I given O. Our primary motivation comes from optimistic parallel discrete event simulation (OPDES). There, a simulator must process events while respecting logical temporal event-ordering constraints; to extract parallelism, an OPDES simulator may speculatively execute events and only *rollback* execution when event-ordering violations occur [4]. In this context, the ability to perform rollback by running time- and space-efficient P and P^-, rather than saving and restoring large amounts of state, can make OPDES more practical. Synthesizing inverses also appears in numerous other software engineering contexts, such as debugging [1], synthesizing "undo" code, or even generating decompressors automatically given only lossless compression code [10]. The challenge in any of these contexts is that constructing program inverses manually is a tedious, time-consuming, and error-prone task.

The program P will generally contain non-invertible statements, such as a destructive assignment. However, in these cases it may still be possible to create an inverse. In particular, we may create an instrumented version of P, called

[1] As a sub-project of the ROSE compiler Backstroke can be downloaded here: http://www.rosecompiler.org

the *forward program*, P^+, with semantics-preserving changes so that it becomes possible to construct P^- from P^+. We then replace executions of P with P^+. For instance, suppose P overwrites a variable v. We may construct P^+ so that it saves the value of v prior to overwriting it. Then, P^- need only restore the saved value to recover v's prior value. In this case, P^+ produces extra outputs, which we denote by S. Indeed, even if P is theoretically reversible without requiring extra output, we may nevertheless need to generate S due to fundamental technical limits on program analysis.

This paper extends our prior compiler-based program inversion framework, called Backstroke [3], to handle the case of programs with loops. As a concrete introductory example, consider the following program on the left, which takes as input an integer $n \geq 0$ and produces the output, $s \leftarrow \sum_{i=0}^{n} i$:

```
s = 0;                          n = 0;
while (n > 0) {                 while (s > 0) {
    s = s + n;                      n = n + 1;
    n = n - 1;                      s = s - n;
}                               }
```

Our goal is to construct an inverse P^- that recomputes n given s, as shown above on the right. Our initial work on Backstroke proposed two new intermediate program representations, which we call the *value search graph* and *route graph* representations. However, these representations could not represent loop structure and hence could not generate the inverse shown above. The method of this paper can; additionally, it can recognize certain special cases where synthesis of an explicit forward program P^+ can be avoided, as is the case in this example. Such special cases are often a requirement in general software engineering (as opposed to OPDES) contexts.

The above example is special in that the loop has single-entry and single-exit points. Consequently, in the usual control-flow graph (CFG) analysis used inside a compiler, the loop's inputs and outputs are easy to identify and program analysis becomes simpler, because the compiler may analyze the loop in relative isolation from the rest of the program. To handle more complex loops, such as those with multiple exits via `break` or `return`, we show how we can modify the CFG to reduce it to the preceding simpler form (Section 3.2). Thus, our method readily applies to the class of so-called *natural* loops.

Note that we use the terms, "program inverse" and "program inversion," even though strictly speaking an inverse for P (as opposed to the instrumented forward program, P^+) may not exist. Nevertheless, this terminology is standard in our OPDES context, so we adhere to it in this paper [11].

2 Prior Foundations: Value Search and Route Graphs

Our work on program inversion for loops builds on a program analysis and synthesis framework that we developed in our prior work. As noted previously, the framework comprises two novel intermediate program representations, which we refer to as *value search graph* and *route graph* [3]. This section summarizes the key ideas behind these representations, explaining how we use them to construct both forward and reverse programs. (Please see our earlier paper for all the formal details [3].) Section 3 describes our extensions for loops.

The basic program inversion workflow in our framework is as follows. Given P, we first translate the program into a standard compiler intermediate program representation known as single static assignment (SSA) form [2]. From the SSA form, we construct a value search graph (VSG) [3]. The problem of finding a forward or reverse program becomes a combinatorial search problem on the VSG. The result of this search is a subgraph of the VSG, which we call a route graph (RG) [3]. There may be many such search results, each of which is a particular forward or reverse program. Lastly, from the RG we synthesize the actual code that implements the forward or reverse program. The process is illustrated in Figure 1. We elaborate on the process and discuss the example next.

The VSG essentially expresses *equality* relations between values in the program. Given these relations, we can determine how values from the input I eventually relate to the values produced during the execution of the forward program P^+, such as the values in the output O. To get the relations, we first transform the program into SSA form. The SSA form is semantically equivalent to the original program but has the special property that each variable is defined only once. In the VSG, nodes represent constants, variables from the SSA, and operators (e.g., "+" or "−" operations); directed edges represent either equality or operand-operator relationships. Edges are also annotated with information about the control-flow paths on which the particular equality relation exists, allowing us to handle conditional branches. Lastly, if there is no way to retrieve a desired value from computational operations alone, we will need to save that value during the execution of P^+ so we can later retrieve it in P^-. Such a *state saving* operation becomes an additional type of node in the VSG. Since state saving may incur both time and space overheads to P^+, we can add a suitable cost to each edge incident to the state saving node.

Given the VSG, we locate *target nodes*, which contain all values we wish to compute. For instance, if we want to build P^- and reconstruct a particular value from the original input I, the corresponding node for that value in the VSG becomes a target node. During the analysis of the VSG, some nodes will be considered *available*. For example, when building P^-, nodes containing constants, the state saving node, and final outputs O of the original program are available. Starting from the targets, we perform a path search through the VSG looking for available nodes. The result of this search is a subgraph of the VSG, which we call a *route graph* (RG). The search algorithm works in such a way that it guarantees each value is retrieved only once for each control flow graph (CFG) path. (The search for a RG that minimizes state-saving cost is NP-Complete, which our prior paper both proves and provides heuristics to find low-cost solutions [3].)

Finally, we generate P^+ and P^- from the RG. In the RG, for each edge pointing to the state saving node, we will instrument P with a state saving statement storing the corresponding value. Also, we use a bit vector to record the control flow paths in P. Then P^+ is generated by instrumenting P with statements performing state savings and CFG path recording. To generate the reverse program P^-, we build a CFG for the reverse function from the RG, and P^- is generated from the CFG.

```
int a, b;
void foo() {
  if (a == 0)
    a = 1;
  else {
    b = a + 10;
    a = 0;
  }
}
```

(a)

```
void foo_forward() {
  int trace = 0;
  if (a == 0) {
    trace |= 1;
    a = 1;
  }
  else {
    store(b);
    b = a + 10;
    a = 0;
  }
  store(trace);
}
```

(b)

```
void foo_reverse() {
  int trace;
  restore(trace);
  if ((trace & 1) == 1)
    a = 0;
  else {
    a = b - 10;
    restore(b);
  }
}
```

(c)

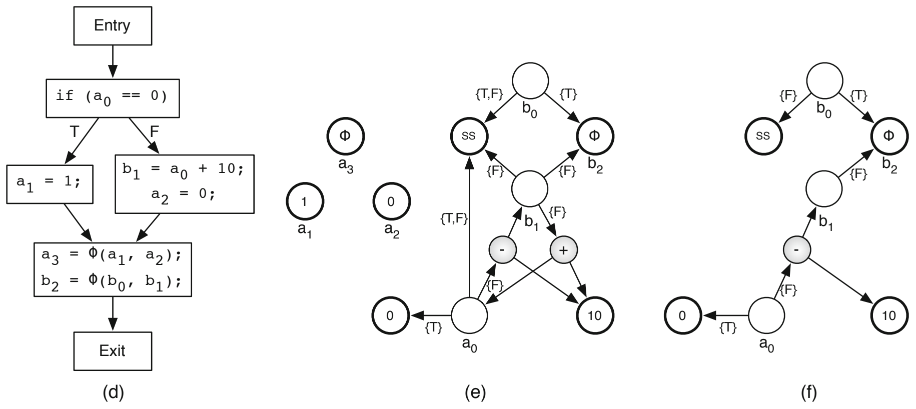


Fig. 1. (a) The original program. (b) The forward program. (c) The reverse program. (d) The CFG in SSA form. (e) The VSG. Nodes in bold are available nodes and all outgoing edges are removed from them. (f) The RG.

Figure 1 illustrates the entire process in a loop-free example. The original program is the function `foo`. The variables a and b are both inputs and outputs. The CFG in SSA form is shown in Figure 1(d). In SSA form, the input of this program are a_0 and b_0, and the output are a_3 and b_2; note that the original variables have subscripts in SSA form, which are referred to as *versions* of the original variables. Observe that versioned variables are in the static program assigned only once. (Programs with loops will need special treatment and extension.) From the SSA CFG, we then build the VSG shown in Figure 1(e). The "SS" node is a special state-saving node. All outgoing edges from each available node shown in bold are removed since the search always ends at available nodes.

Each equality relation (edge) is constrained by a set of CFG paths. Since there are only two paths in the program, we use T and F to represent the path passing through the true and false bodies, respectively. Our goal is to retrieve a_0 and b_0, which is done by searching the VSG to find a way to get its value for each CFG path. The search result, which is the RG, appears in Figure 1(f). We build the forward and reverse programs, Figures 1(b) and (c), respectively, from the RG. (For details on this process, refer to our prior paper [3].)

3 Handling Loops

Unmodified, our prior method as described in Section 2 cannot handle loops for two key reasons. First, a loop results in cyclic paths in the CFG, whereas our prior analysis relies on paths being acyclic. Acyclic paths make it easy to check that the reverse program restores any desired input value no matter what path the forward program takes. Secondly, our prior VSG and RG cannot represent loop control structure. Therefore, it is simply not possible to synthesize, for example, a loop in the reverse code from the RG. Nevertheless, we *can* reuse most of the prior method by decomposing the problem suitably. In particular, we keep the basic framework of "SSA to VSG to RG." Our extension replaces SSA with a loop-enabled variant, and then extends our VSG and RG representations and algorithms to deal with cycles, thereby addressing the two aforementioned issues.

Let us first assume that each loop to be reversed is a single-entry, single-exit while loop (we will explain what is a while loop later). We explain in Section 3.2 how to convert other kinds of loops into this form. We also assume that each loop must terminates at run-time so that we can always get an output. Given an input while loop, there are three steps to build a VSG.

1. We temporarily collapse each while loop into a single abstract node in the CFG, thereby creating a logically loop-free CFG from which we can build a VSG by directly applying our prior method. This "transformation" is for program analysis purposes only. We denote this loop-collapsed VSG by G_P.
2. Similarly, we directly apply our prior method to build a VSG for each loop *body*, which may be treated as another loop-free program. (If the body contains nested loops, these are similarly collapsed as in Step 1 above.) Note that path information in these loop body VSGs are local to the loop body. We denote this VSG for the loop body by G_L.
3. At this point, G_P and G_L are disconnected. Therefore, we introduce new special edges to connect them, thereby resulting in a single connected VSG. These connecting edges are a new type of edge and constitute the main extension to our prior VSG in order to support loops. The new edges connect each input (or output) of a loop to the input (or output) of the loop's body. These new edges serve as markers: when we search the VSG and produce an RG containing these edges, then we know we need to synthesize a loop.

Since Steps 1 and 2 use our prior VSG construction, we need not discuss them further here. What changes is the third step, as detailed below, including new

VSG searching rules and new procedures for synthesizing loops from the search result (i.e., the RG).

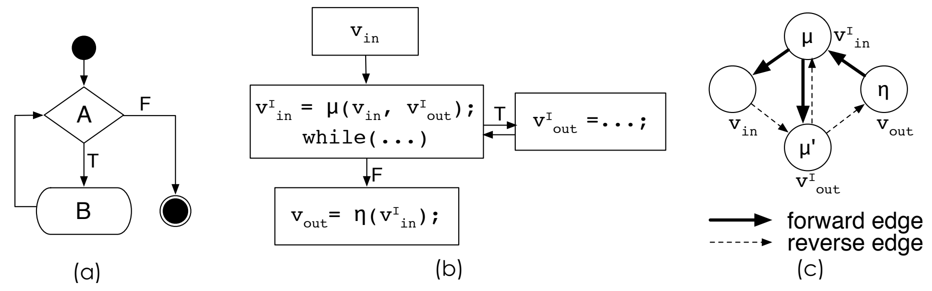


Fig. 2. (a) The diagram of a while loop. (b) The CFG in loop-closed SSA form for a variable v modified in the loop. (c) Forward and reverse edges.

3.1 Dealing with While Loops

We first consider a while loop with the diagram shown in Figure 2(a). We further assume that A has no side-effects and that there are no escapes from B. Thus, the loop only exits from its entry.

Given such a while loop, we transform it into the *loop-closed SSA form* [9], illustrated in Figure 2(b). Loop-closed SSA differs from conventional loop-free SSA as follows. In conventional SSA, a special marker called a ϕ *function* is placed in the CFG at the first program point where two distinct versions (definitions) of a variable, computed along different program paths, meet. In loop-closed SSA, if a value is defined inside of a loop and used outside of it, we place a special single entry ϕ function at the *exit* of the loop. To distinguish this type of loop-specific ϕ function from a conventional ϕ function as used in loop-free programs, we denote the loop-specific form by the term η function, by convention [7]. Additionally, suppose a definition of a variable from outside the loop and a definition coming from a back-edge of the loop meet at a program point. Again, we create a ϕ function marker here, and to distinguish it, we refer to it as a μ function.

To see how these markers work, consider a variable v modified by a while loop; we now describe the corresponding loop-closed SSA form, which Figure 2(b) illustrates. Let v_{in} denote the input value of v before the loop executes, and v_{out} the output value of v after the loop executes. Next, let the input to the loop *body* be v^I_{in} and the output v^I_{out}. (The superscript I is intended to remind the reader that these are values associated with an *iteration* of the loop, as opposed to the values before and after the loop.) Then, v^I_{in} is defined by a μ function as $v^I_{\text{in}} = \mu(v_{in}, v^I_{out})$, and v_{out} is defined by a η function as $v_{\text{out}} = \eta(v^I_{in})$. That is, $v^I_{\text{in}} = \mu(v_{in}, v^I_{out})$ indicates the program point at which v has either the initial value before the loop executes or the value produced by some iteration of the loop; and $v_{\text{out}} = \eta(v^I_{in})$ indicates the program point at which v has the final value once the loop completes.

From this loop-closed SSA form, we wish to build a VSG that will express equality relations among the four SSA values, v_{in}, v_{out}, v^I_{in}, and v^I_{out}. This VSG result is shown in Figure 2(c). Recall that nodes in the VSG represent values, and edges the equality relations. There are four value nodes. The nodes v_{in} and v_{out} are part of the loop-collapsed G_P, and v^I_{in} and v^I_{out} belong to the loop body's G_L. The μ and η functions indicate how to connect G_P and G_L. In particular, the three solid bold edges are associated with the dependences induced by executing the loop in the forward direction; we call these the *forward edges*, and a μ node is incident to all three. The presence of these edges make it possible to obtain v_{out} by some path passing through G_L, and simultaneously indicate that a loop is present for subsequent code generation. Similarly, the three dashed edges are *reverse edges* associated with dependences induced in the reverse direction. These edges make it possible to obtain v_{in} by some path through G_L. Note that the reverse edges form a symmetry to the forward edges. From this symmetry, we define the node incident to all three reverse edges as a μ' node. Later we will show how the search traverses these edges.

Having built the CFG, the next step is to search it, producing the RG result. Recall from Section 2 that we are given a set of target nodes whose values we wish to eventually compute from a starting set of available nodes. We search for a path from available nodes to target nodes; the subgraph representing paths is the RG, which is not necessarily unique. Our algorithm is similar to the one we have described previously [3], but for loops we need three additional search rules:

– During a search for a value, once a forward/reverse edge is selected, all edges in the other category cannot be chosen. This is because either a forward or a reverse loop will be built to retrieve the value.
– When the search reaches a μ or μ' node, it will be split into two sub-searches, in G_P and G_L, respectively, through the two outgoing forward or reverse edges. For example, in Figure 2(c), if the search reaches v^I_{in}, the algorithm begins two sub-searches beginning with v_{in} and v^I_{out}.
– During the search, the algorithm may form a directed cycle only in G_L; furthermore, such a cycle must contain a forward or reverse edge between a μ and μ' node. Once a cycle is formed, the search in G_L is complete.

We build a while loop as either a forward or a reverse loop. Synthesizing such a while loop consists of synthesizing its body and predicate.

Building the Loop Body. The loop body in the reverse program is generated from the search result in G_L. For each variable we remove the edge between the μ and μ' nodes and hence remove the cycles, so that we can generate the loop body using our prior code generation algorithm [3].

Building the Loop Predicate. To guarantee that the generated loop has the same iterations at runtime as the original loop, we need to build a proper loop predicate. We propose three approaches to building a correct loop predicate. To illustrate those approaches, we temporarily introduce the following loop example. We assume that the omitted statements modify neither `A[]` nor `i`.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

- **Approach 1:** Building the same loop predicate as that in the original loop. To build this predicate, we need to retrieve each value in the predicate. A new search is needed to acquire those values, and the search result will be combined into the RG generated above. For the example above, we can build a loop below that has the same number of iterations as the original one. The omitted statements will be substituted by the loop body built above.

```
i = 0;
while (A[i] > 0) {
    /* ... */
    i = i + 2;
}
```

- **Approach 2:** Building the loop predicate from a variable updated in the loop. Given a variable v and its four definitions: v_{in}, v^I_{in}, v^I_{out}, and v_{out}, if $v^I_{\text{in}} \neq v_{\text{out}}$ in each iteration except the last definition of v^I_{in} (which is actually v_{out}), and if we can retrieve v_{in} and v_{out} before the loop (hence we cannot retrieve them through the loop), and v^I_{out} in the loop, we can use them to build a while loop as:

$$u := v_{in};\ while(u \neq v_{out})\ \{\ /*\ update\ u\ */\ \}$$

Similarly, if $v^I_{\text{out}} \neq v_{\text{in}}$ in each iteration, and v_{in} and v_{out} can be retrieved before the loop, and v^I_{in} can be retrieved in the loop, we can use them to build a while loop as:

$$u := v_{out};\ while(u \neq v_{in})\ \{\ /*\ update\ u\ */\ \}$$

In general, it is difficult to detect all variables satisfying the properties above. However, there are some special cases. One case is that of *monotonic variables* [12], which are monotonically strictly increasing or decreasing in each iteration. Another is that of induction variables, which are special monotonic variables that are relatively easier to recognize. In the above example, `i` is an induction variable. Assume its final value after the loop is `i1` that is known, and then we can build the following loop with the predicate using `i`.

```
i = 0;
while (i != i1) {
    /* ... */
    i = i + 2;
}
```

– **Approach 3:** Instrumenting the original loop with a counter counting the number of iterations. The counter has the initial value zero and is incremented by one on each back edge of the loop. The final value of the counter is stored in the forward program and restored in the reverse program as the maximum value of another loop counter. This approach generally works if either of the above two approaches fail. However, it requires instrumentation (the counter), and therefore forces generation of a forward program. Below we show the instrumented loop in the forward program (left) and the generated loop in the reverse program (right) for the above example.

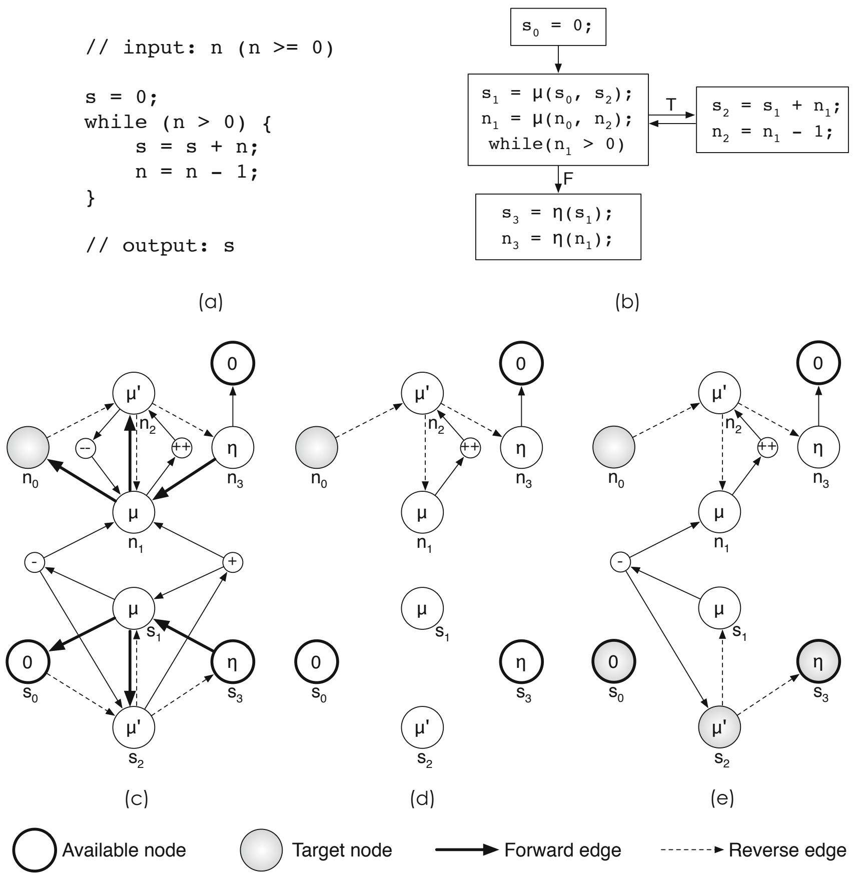
```
i = count = 0;                  restore(count);
while (A[i] > 0) {              while (count > 0) {
    /* ... */                       /* ... */
    i = i + 2;                      count = count - 1;
    count = count + 1;          }
}
store(count);
```

We prioritize these approaches as follows. Applicability and state-saving cost are our main criteria. We prefer Approach 1 and 2 over 3. When either 1 or 2 apply, if no state-saving is required, we apply them. Otherwise, we try Approach 3 and choose the overall approach with the least cost.

As an example, suppose we apply this algorithm to the loop in Figure 3(a). Figure 3(b) shows its CFG in loop-closed SSA. The input is n_0 and the output s_3. Our goal is to generate a reverse program that takes s_3 as input and produces n_0. We build the VSG shown in Figure 3(c), with forward and reverse edges shown as bold and dashed edges, respectively. Note that the equality between n_3 and 0 is acquired from solving constraints, a standard compiler technique, as discussed in Section 3.3.[2] The search result for value n_0 is shown in Figure 3(d), from which we can build the loop body as { `n = n + 1;` }.

Next, we build the loop predicate. In our example, because we wish to retrieve the initial value of n, we cannot use it to build the loop predicate. We can discover that s is a monotonic variable, and that both the initial and final values of s, which are 0 and s_3, respectively, are available. To get s_2, we search its value on the VSG and the search result is shown in Figure 3(e). As a result, we build the loop predicate from s and the reverse program is generated as below.

[2] For clarify, we remove the equality $n_1 = s_2 - s_1$, as this relation will not be used during the search.



```
// input: n (n >= 0)

s = 0;
while (n > 0) {
    s = s + n;
    n = n - 1;
}

// output: s
```

```
s0 = 0;

s1 = μ(s0, s2);
n1 = μ(n0, n2);
while(n1 > 0)

s2 = s1 + n1;
n2 = n1 - 1;

s3 = η(s1);
n3 = η(n1);
```

Fig. 3. (a) The program of our example. (b) The CFG in loop-closed SSA form. (c) The VSG. (d) The RG for retrieving n_3. (e) The RG for retrieving n_0 and s_2.

```
n = 0;
while (s != 0) {
    n = n + 1;
    s = s - n;
}
```

3.2 Dealing with Loops other than While Loops

In practice, the vast majority of loops have a single entry, which are called *natural loops* [5]. Loops with more than one entry are quite rare and can in fact be transformed into natural loops [5]. However, it is quite common that a loop has several exits. For example, in C/C++ we may exit a loop early through `break`, `return`, or `goto` statements. Nevertheless, given a non-while natural loop, we can transform it to separate the last iteration from the loop;

then, the remaining iterations form a new while loop, and the last iteration will not belong to the loop and hence can considered with the control flows outside of the loop. We then process the new while loop as previously described. Note that this "transformation" is only applied to the CFG during the analysis, and not to the original program. As such, in the forward program P^+ the last iteration and other iterations of each loop continue to share the same code.

Figure 4(a) shows a loop in a CFG, with a header (node 1) and two back edges (4→1 and 5→1). There are two different exits from this loop, which are nodes 6 and 7. Figure 4(b) shows the CFG of the transformed loop. This transformation is performed as follows.

In a natural loop, only the last iteration takes the exit, and any other iteration goes back to the loop header. Therefore, if the last iteration is peeled off from the loop, this loop will turn into a while loop. To implement this transformation, we create a new branch node with an unknown predicate that returns *true* if the next iteration is not the last one and *false* otherwise. Note that we will not build this predicate in the forward program. The new branch node turns over all in-edges of the loop header. Its *true* labeled out-edge will point to the loop header of a copy of the loop (node $1'$) with back edges but without exit edges, and all back edges are redirected to this new branch node making it a new loop header. Note that after removing exit edges it is possible that a previous branch node becomes a non-branch node (node $3'$, for example), which is fine because the removed branch edge will not be taken. Then, we can remove the (side effect free) predicates from those nodes. The edge labeled with *false* from the new branch node will point to the original loop header (node 1) and all back edges in the original loop are removed, since the last iteration won't take the back edge. The nodes from which the exit of the program is not reachable due to the back edge removal are removed (node 4 and 5, for example). Again the predicate is removed from a node once it is not a branch node anymore (node 2 and 3).

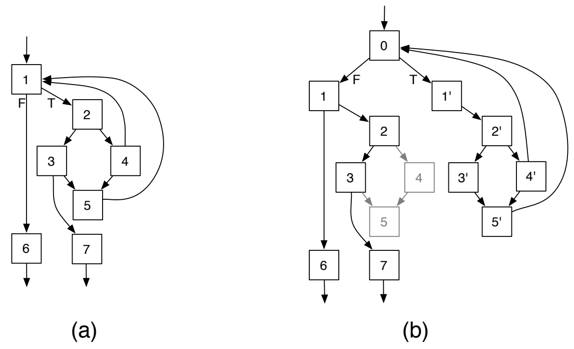


Fig. 4. (a) A loop in CFG with two back edges and two exits. (b) The CFG of the transformed loop.

After the transformation, all loops in the program become while loops and our method applies.

3.3 Discussion

Equality from Solving Constraints. We use constraint solving to obtain any needed equalities. For example, if $a \geq b$ and $a \leq b$, then $a = b$. This method is useful to get the final value of a loop counter. A typical example is shown below:

```
i = 0; while (i < N) { ...; i = i + 1; }
```

where i is a loop counter incremented by one in each iteration, and $N \geq 0$. In Floyd-Hoare logic [6], the partial correctness of a while loop is governed by the following rule of inference [8]:

$$\frac{\{C \wedge I\}\ \text{body}\ \{I\}}{\{I\}\ \textbf{while}\ (C)\ \text{body}\ \{\neg C \wedge I\}}$$

where C is the while condition, and I is a loop invariant, which is informally defined as a statement of the conditions that should be true on entry into a loop and are guaranteed to remain true on every iteration of the loop. In this example, we choose $i \leq N$ as a loop invariant. After replacing C and I with $i < N$ and $i \leq N$, the postcondition at the end of the loop $\{\neg C \wedge I\}$ becomes $\neg(i < N) \wedge i \leq N$, from which we get $i = N$.

Rebuilding Control Flows for the Reverse Program. In our prior work [3], we record the runtime control flow paths in the forward program using a bit vector. Specifically, a bit is used to record which path is taken at each two-way branch node. The bit vector is stored at the end of the forward program and is used to rebuild the control flows in the reverse program. This method has both low time and space overhead. However, for program with loops, recording control flows in each iteration of a loop may bring considerable space overhead.

To avoid this overhead, we found that we could calculate the control flows instead of storing and restoring them. Basically, there are two ways to do that. First, for a predicate in the original program, we can recover all values used in the predicate in the reverse program, then use those values to produce the result of the predicate. Second, if there is a ϕ function defined at a join node in the original program as $v_2 = \phi(v_0, v_1)$, and v_0 and v_1 cannot have the same value, then if we can get the value range of v_0 or v_1 and retrieve the value of v_2 , we can build a predicate by checking if the value of v_2 is in the value range of v_0 or v_1. More details will be introduced in our future publications.

4 Conclusion and Future Work

With our loop handling methods, Backstroke can now handle a variety of programs that operate on scalar variables. The next major direction for this work are to handle array programs and programs that manipulate complex data structures, such as linked data structures. However, our work to date lays the critical foundations for such extensions; for example, the index of an array is a scalar that can be retrieved in a loop using the method described in this paper. Our future work will focus on synthesizing reverse programs that use arrays and object accesses. We are particularly interested in whether our techniques can be used to reverse programs known to be reversible by computation, such as lossless compression and decompression, encryption and decryption, among numerous others.

Acknowledgements. This work was supported in part by the National Science Foundation (NSF), under CAREER award number 0953100, and grants from the U.S. Department of Energy (DOE) through Lawrence Livermore National Laboratory (LLNL) LDRD project 10-ERD-025. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of NSF, DOE, or LLNL.

References

1. Biswas, B., Mall, R.: Reverse execution of programs. ACM SIGPLAN Notices 34(4), 61–69 (1999)
2. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
3. Hou, C., Vulov, G., Quinlan, D., Jefferson, D., Fujimoto, R., Vuduc, R.: A New Method for Program Inversion. In: O'Boyle, M. (ed.) CC 2012. LNCS, vol. 7210, pp. 81–100. Springer, Heidelberg (2012)
4. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)
5. Muchnick, S.S.: Advanced Compiler Design Implementation. Morgan Kaufmann Publishers (1997)
6. Pratt, V.R.: Semantical consideration on floyo-hoare logic. In: 17th Annual Symposium on Foundations of Computer Science (1976)
7. Ballance, R.A., Maccabe, A.B., Ottenstein, K.J.: The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Language. In: PLDI 1990 (1990)
8. Roşu, G., Ellison, C., Schulte, W.: Matching Logic: An Alternative to Hoare/Floyd Logic. In: Johnson, M., Pavlovic, D. (eds.) AMAST 2010. LNCS, vol. 6486, pp. 142–162. Springer, Heidelberg (2011)
9. Pop, S., Jouvelot, P., Silber, G.-A.: In and Out of SSA: A Denotational Specification. In: Static Single-Assignment Form Seminar (2009)
10. Srivastava, S., Gulwani, S., Chaudhuri, S., Foster, J.S.: Path-based inductive synthesis for program inversion. In: PLDI 2011. ACM Press (2011)
11. Vulov, G., Hou, C., Vuduc, R., Quinlan, D., Fujimoto, R., Jefferson, D.: The backstroke framework for source level reverse computation applied to parallel discrete event simulation. In: Winter Simulation Conference (2011)
12. Wolfe, M.: Beyond Induction Variables. In: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, PLDI (1992)

Frugal Encoding in Reversible $\mathcal{MOQA}$: A Case Study for Quicksort

Diarmuid Early, Ang Gao, and Michel Schellekens

Centre for Efficiency Oriented Languages
University College Cork
Ireland*
{a.gao,m.schellekens}@cs.ucc.ie

Abstract. $\mathcal{MOQA}$ is a high-level data structuring language, designed to allow for modular static timing analysis. In essence, $\mathcal{MOQA}$ allows the programmer to determine the average running time of a broad class of programmes directly from the code in a (semi-)automated way. The $\mathcal{MOQA}$ language has the property of randomness preservation which means that applying any operation to a random structure, results in an output isomorphic to one or more random structures, which is the key to systematic timing. The language, its implementation and the associated timing tool have been reported on in the literature. Randomness preservation is key in ensuring modular timing derivation. A degree of reversibility in turn is a key aspect of ensuring randomness preservation. All operations of the $\mathcal{MOQA}$ language can be made reversible with minimal additional bookkeeping. A challenge in achieving this encoding in a frugal way is to ensure subsets of data can be stored without excessive overheads. The paper focuses on illustrating such an encoding for the case of the Quicksort algorithm. Similar encodings are explored to ensure efficient reversibility of all $\mathcal{MOQA}$ operations. The paper is self contained, i.e. no prior knowledge of the $\mathcal{MOQA}$ language is needed to follow the encoding argument. We show how to efficiently encode the information needed to reverse the split of a list into two sublists. The code for reversible Quicksort is provided and an example illustrates the algorithm's reverse execution.

Keywords: Reversible computing, Encoding, Algorithms, Quicksort, Sorting, Data structures, Partial orders, Random Structures, Time analysis, $\mathcal{MOQA}$ language.

1 Introduction

[Sch08] introduces $\mathcal{MOQA}$, a high-level data structuring language, designed to allow for modular static timing analysis. In essence, $\mathcal{MOQA}$ allows the programmer to determine the average running time of a broad class of programs

* This work was supported by the Science Foundation Ireland (SFI) Grant under Grant number: 07/IN.1/I977.

directly from the code in a (semi-)automated way. For further discussion of the mechanics of the process, see [Sch08, Hic08, SHB04]. As pointed out in [Sch08], the modularity property brings a strong advantage for the programmer. The capacity to combine parts of code, where the average-time is simply the sum of the times of the parts, is a very helpful advantage in static analysis, something which is not available in current languages. Moreover, re-use is a key factor in the $\mathcal{MOQA}$ approach: once the average time is determined for a piece of code, then this time will hold in any context. Hence it can be re-used and the timing impact is always the same. Modularity also improves precision of static average-case analysis, supporting the determination of accurate estimates on the average number of basic operations of $\mathcal{MOQA}$ programs. Reversible $\mathcal{MOQA}$ discussed in [Sch10] and [Ear10] complements traditional applications of reversibility with a new application domain, that of average-case cost analysis (where cost can be running time or power usage) of reversible $\mathcal{MOQA}$ programs. We provide here the frugal encoding for the reversible $\mathcal{MOQA}$ split operation and illustrate the approach via a reversible version of the well-known Quicksort algorithm.

Reversibility traditionally plays a role in hardware design, with implications for low power design [Lan61,Ben73,Tof80]. A few exceptions focus on high-level reversible languages, including the language JANUS and the work discussed in [YG07]. Most reversible approaches remained at hardware level. As observed, the use of $\mathcal{MOQA}$ as a high level reversible language brings a new type of application to the area of reversible computing. As pointed out in [Sch08] a sufficient condition for algorithms to be analyzable in a modular way is that they are random bag preserving. Not all algorithms are random bag preserving though, a case in point being the traditional heapsort algorithm [Sch08]. As shown in [Sch10], random bag preservation can typically be guaranteed by ensuring a "locally" one-to-one mapping, e.g. a mapping guaranteed to be one-to-one on each of the parts of a partition of the inputs[1]. $\mathcal{MOQA}$'s random bag preserving programs are ensured to allow for a greatly simplified average-case analysis. The key to understanding $\mathcal{MOQA}$ as a new application domain for reversible computing is that its programs, with little additional book-keeping become fully reversible [SEPV09, Sch10, Ear10]. Hence we establish a link between reversibility and the capacity for modular (i.e. semi-automated) average-case analysis. Of course, general algorithms typically can be subjected to average-case analysis techniques. The key point is that some algorithms, like heapsort, are not random bag preserving and hence either require complicated (non-automatable) techniques or escape average-case analysis by current techniques. As a degree of reversibility lies at the heart of random bag preservation [Sch10], reversibility has the potential to play a fundamental role in the design of modularly predictable algorithms. Since with a little more bookkeeping, $\mathcal{MOQA}$ becomes a fully reversible language, the exploration of its reversible properties is worthwhile, in particular since the reversible programs in turn allow for an exact prediction of average-case computation time. Hence we can predict in a static way, the cost of computing forward and backward in the language.

[1] For example, the $\mathcal{MOQA}$ product operation as discussed in [Sch08], Theorem 5.1.

The reversible aspects of $\mathcal{MOQA}$ open up possibilities to apply $\mathcal{MOQA}$ to determine the average-case power usage but possibly also to use $\mathcal{MOQA}$ to achieve power optimization based on traditional reversible approaches [TDJ10]. We continue with a self contained introduction to $\mathcal{MOQA}$ data structures.

The primary objects in $\mathcal{MOQA}$ are finite labelled partial orders, or *LPO*s, and random structures.

Definition 1. *A **labelled partial order**, or LPO, is a triple* $(A, \sqsubseteq, l)$*, where* A *is a set,* $\sqsubseteq$ *is a partial order on* A *(that is, a binary relation which is reflexive, anti-symmetric, and transitive), and* l *is a bijection from* A *to some totally ordered set* C *which is increasing with respect to the order* $\sqsubseteq$*. If* l *is not increasing, we call* $(A, \sqsubseteq, l)$ *a **weakly labelled partial order**, or WLPO.*

Definition 2. *Given a finite partial order* $(A, \sqsubseteq)$ *and a totally ordered set* C *with* $|C| = |A|$*, the **random structure*** $R_C(A, \sqsubseteq)$ *is the set of all LPOs* $(A, \sqsubseteq, l)$ *with* $l(A) = C$*.*

$\mathcal{MOQA}$ operations are comparison-based, and so the choice of the set C is unimportant, and we will generally write a random structure as $R(A, \sqsubseteq)$, without the subscript[2].

The $\mathcal{MOQA}$ language has the property of 'randomness preservation', which means that applying any operation to each *LPO* in a random structure results in an output isomorphic to one or more random structures, which is the key to systematic timing. We will give a basic example below to illustrate these concepts. Four of the main random bag preserving operations are the random product operation, the random delete operation, the random projection operation and the random split operation. We refer the reader to [Sch08] for further information. The control flow of $\mathcal{MOQA}$ programs is governed by strict rules, given in [Sch08], chapter 7. In essence $\mathcal{MOQA}$ programs involve tightly controlled higher level operations that involve the basic operations listed above. Higher level operations include conditional statements (with restricted conditional), recursion (over so-called series-parallel data structures) and for-loops.

For the purpose of this paper, it suffices to introduce the $\mathcal{MOQA}$ split operation, i.e. the key operation in the Quicksort algorithm. This is the topic of Section 2.

2 Background

In this section, we focus on a simple 'randomness preserving' operation split. For a complete description of the $\mathcal{MOQA}$ operations, designed to capture traditional data structuring operations in a randomness preserving fashion, we refer the reader to [Sch10] or Springer book [Sch08] for the complete description of $\mathcal{MOQA}$.

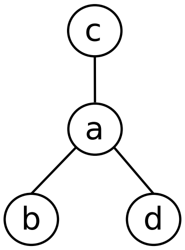
[2] More formally, we can consider the random structure to be the quotient of the set of all *LPO*s on the partial order $(A, \sqsubseteq)$ with respect to a natural isomorphism. See [Ear10] for a full discussion.

The classical algorithms Quicksort and Quickselect are both based on a split operation, which takes a list and a pivot (which is an element of the list) as arguments. We use a simpler version to reduce technicalities. The pivot for split is chosen to be the first element of the list. This choice is again irrelevant. Other choices will result in similar random structures with minor technical modifications.

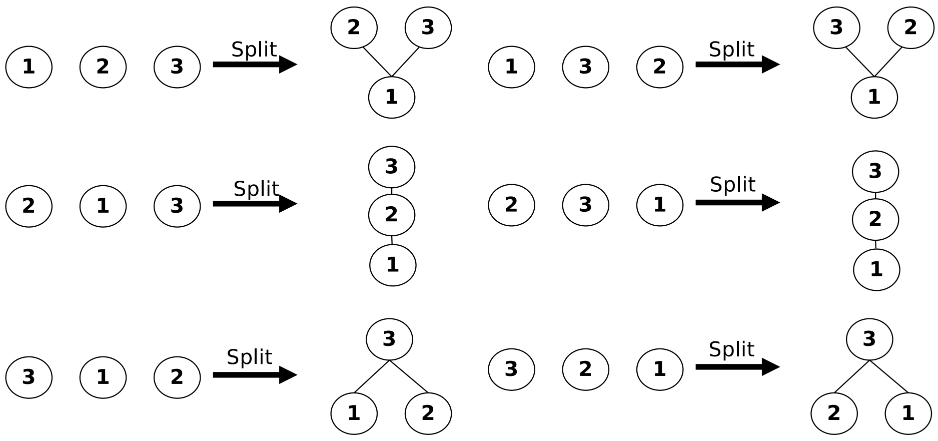
Split proceeds on a list of size n by comparing, in left to right order and starting at the second element, each label of the i-th element, $i \in \{2 \ldots n\}$, with the pivot label. In cases where the label of the i-th element is greater than the pivot label, these elements and their labels are placed above the pivot. Otherwise they are placed below the pivot. In fact, the classical split puts the i-the element to the left or the right of the pivot. The $\mathcal{MOQA}$ split however puts it below or above the pivot, a minor technical difference.

For example, if we have a random list a, b, c, d, one possible LPO for this list is shown in Example 1. In this example $a > b$, $a > d$ and $a < c$. In the follow context, we will call upper part Y_1, middle pivot Y_2 and bottom part Y_3. A more complete example is shown in Example 2. We use the label set $C = \{1, 2, 3\}$ to simplify the example

Example 1. One example apply Split on random list $[a, b, c, d]$.



Example 2. Split operation on random list size 3.



We shows the effect of Split acting on each of the six discrete *LPO*s of a size 3 random list. It is easy to see that the first two outputs form a 3 element V-shaped random structure, denoted by $\vee_3$; the middle two outputs consist of two copys of a linear order random structure of size 3, denoted by S_3, and the last two form a 3 element wedge-shaped random structure, denoted by $\wedge_3$. Thus we say that Split transforms the 3 element random list labelling into $\{(R(\vee_3), 1), (R(S_3), 2), (R(\wedge_3), 1)\}$.

The result of the operation can be generalized to random list of size n, and we refer the interested reader to [Sch08]. In the context of this paper, the resulting random structures and their multiplicities are not crucial and the contents are self contained.

3 Efficient Encoding

First, we show that how to efficiently encode the information needed to reverse the split of a list into two sublists (upper part and bottom part). We assume that all list are ordered.

Clearly, if we know the position of each item in the upper sub-list in the original list, this is enough. For example, if the two sub-lists are (f, g, q, p, z) and (m, s, b, t), and if we know that the elements of the upper list were initially in position $1, 3, 4, 8$ and 9, then the original list except pivot element must be:

$$\begin{array}{ccccccccc} (f, & m, & g, & q, & s, & b, & t, & p, & z) \\ 1 & & 3 & 4 & & & & 8 & 9 \end{array}$$

So, given positive integers $k < n$, suppose we have two lists of length k and $n-k$. To combine them in the original order, we need k distinct integers between 1 and n. We write these as $\{x_i\}_{i=1}^k$, where the x_i are in ascending order. Clearly there are $\binom{n}{k}$ different sets with these properties.

One way to encode a subset of size k from a set of size n is with a binary string of length n, whose i^{th} bit is 1 if and only if the i^{th} element of the set is included in the subset. However, if k is very small or very large (relative to n), this encoding is very inefficient. For example, we can encode a one-element subset with a number between 1 and n, or $log_2(n)$ bits, whereas this method would require n bits. Using this method to reverse the split operation would give a worst-case reversal overhead for Quicksort of $n!2^{O(n^2)}$, while we will show that a more frugal encoding can achieve the same result with a maximum overhead of $n!$.

The following lemma shows how to encode position indices $\{x_i\}_{i=1}^k$.

Lemma 1. *Given a positive integer n and an integer $k \in [0, n]$, the function $f(\{x_i\}_{i=1}^k) = \sum_{i=1}^k \binom{x_i - 1}{i}$ is a one-to-one mapping from the k-element subsets of the first n integers (in ascending order) to the set of integers from 0 to $\binom{n}{k} - 1$.*

Proof. First, we prove that no two subsets map to the same value(i.e. f is an injection). Suppose that $f(\{x_i\}_{i=1}^k) = f(\{y_i\}_{i=1}^k)$, and that $x_i \neq y_i$ for some

$i \in [1, k]$. Let j be the largest value of i for which $x_i \neq y_i$. We can assume that $x_j > y_j$. Now for all $i \leq j$, $y_i \leq x_j - 1 - j + i$(note: $x_j > y_j$, $y_j > y_i$, both x_i and y_j are integers in range [1,n]) and so:

$$\sum_{i=1}^{j} \binom{y_i - 1}{i} \leq \sum_{i=1}^{j} \binom{x_j - 2 - j + i}{i} = -1 + \sum_{i=0}^{j} \binom{x_j - 2 - j + i}{i} \text{(change of index)}$$
$$= -1 + \binom{x_j - 1}{j} < \binom{x_j - 1}{j}$$

Where the last equation follows from the hockeystick lemma [Zei99].
So:

$$f(\{y_i\}_{i=1}^{k}) = \sum_{i=1}^{k} \binom{y_i - 1}{i} = \sum_{i=1}^{j} \binom{y_i - 1}{i} + \sum_{i=j+1}^{k} \binom{x_i - 1}{i}$$
$$< \sum_{i=j}^{k} \binom{x_i - 1}{i} \leq f(\{x_i\}_{i=1}^{k})$$

Which contradicts the assumption. To avoid contradiction, f must be an injection.

Now we prove upper and lower bounds on f. Clearly $f(\{x_i\}_{i=1}^{k}) \geq 0$. To get the upper bound, note that $x_i \leq n - k + i$, and thus:

$$f(\{x_i\}_{i=1}^{k}) \leq \sum_{i=1}^{k} \binom{n - k - 1 + i}{i} = -1 + \sum_{i=0}^{k} \binom{n - k - 1 + i}{i} \text{(change of index)}$$
$$= -1 + \binom{n}{k}$$

And the last equation again follows from the hockeystick lemma [Zei99]. So the range of f is $[0, \binom{n}{k} - 1]$. But now there are $\binom{n}{k}$ distinct subsets, $\binom{n}{k}$ possible outputs and each input maps to a distinct output, so f must be one-to-one. □

We provide a brief intuition for the indexing of the subsets:

- We define an order on the subsets whereby subset A is greater than subset B if the largest element in one set but not the other is in subset A. Consider the number of subsets less than a given subset, which contains the elements of ranks $x_1, x_2, \cdots, x_k$ in the full set.
- If the element of rank x_p in the overall set is the largest that is not common to both subsets, then all the larger elements are in common, and the smaller subset can have any p elements from among the smallest $x_p - 1$ in the set, a total of $\binom{x_p - 1}{p}$ possibilities.

– But now, for any pair of distinct subsets, there is only one largest element in one but not the other, and so any subset smaller than the given one must match this pattern for some $p \in [1, k]$. So the total number of smaller subsets is $\sum_{i=1}^{k} \binom{x_i-1}{i}$.
– Now, assigning each subset an index which is the number of smaller subsets gives each subset a unique index between 0 and $\binom{n}{k} - 1$.

We also brief outline an algorithm for extracting the sequence $\{x_i\}_{i=1}^{k}$ given $f(\{x_i\}_{i=1}^{k})$ (and also the value of n and k)

Algorithm 1. Extracting the sequence $\{x_i\}_{i=1}^{k}$ given $f(\{x_i\}_{i=1}^{k})$

```
Input: N -- $f(\{x_i\}_{i=1}^{k})$, n -- size of original list, k -- sublist size.
Output: S (a set $\{x_i\}_{i=1}^{k}$)

$Extract(N, n, k)$ :
$j \leftarrow k$
$S \leftarrow \emptyset$
for $i \leftarrow n$ to 1 do
    if $N \geq \binom{i-1}{j}$ then
        $S \leftarrow \{i\} \cup S$
        $N \leftarrow N - \binom{i}{j}$
        $j \leftarrow j - 1$
    end if
end for
return S
```

Example 3. Suppose we split $(f, m, g, q, s, b, t, p, z)$ into (f, g, q, p, z) and (m, s, b, t). Then $(x_1, x_2, x_3, x_4, x_5) = (1, 3, 4, 8, 9)$.

$$\begin{aligned} f(\{x_i\}_{i=1}^{5}) &= \binom{9-1}{5} + \binom{8-1}{4} + \binom{4-1}{3} + \binom{3-1}{2} + \binom{1-1}{1} \\ &= 56 + 35 + 1 + 1 + 0 = 93 \end{aligned}$$

Now, given $N = 93$, n = 9, k = 5, we set j = 5 and run the algorithm:
$i = 9, j = 5, 93 > \binom{9-1}{5}$ so $S = \{9\}, j = 4, N = 93 - \binom{9-1}{5} = 37$
$i = 8, j = 4, 37 > \binom{8-1}{4}$ so $S = \{8, 9\}, j = 3, N = 37 - \binom{8-1}{4} = 2$
$i = 7, j = 3, 2 < \binom{7-1}{3}$ so skip
$i = 6, j = 3, 2 < \binom{6-1}{3}$ so skip
$i = 5, j = 3, 2 < \binom{5-1}{5}$ so skip
$i = 4, j = 3, 2 > \binom{4-1}{3}$ so $S = \{4, 8, 9\}, j = 2, N = 2 - \binom{4-1}{3} = 1$
$i = 3, j = 2, 1 = \binom{3-1}{2}$ so $S = \{3, 4, 8, 9\}, j = 1, N = 1 - \binom{3-1}{2} = 0$
$i = 2, j = 1, 0 < \binom{2-1}{1}$ so skip
$i = 1, j = 1, 0 = \binom{1-1}{1}$ so $S = \{1, 3, 4, 8, 9\}, j = 0, N = 0 - \binom{1-1}{1} = 0$

Thus for a $\mathcal{MOQA}$ split operation we can keep track the encoding for the upper elements in resulting LPO and restore their original position with the help of *Extract* algorithm.

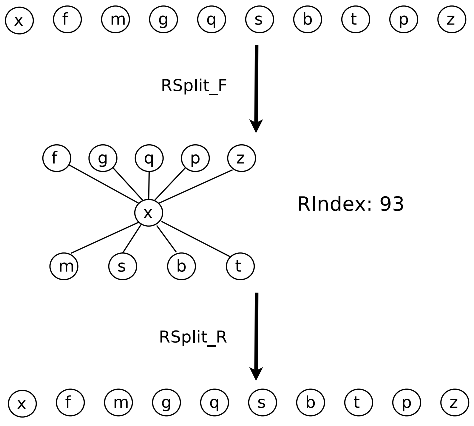
Algorithm 2. Reversible Split algorithm: Forward computing

```
Input: a discrete $LPO$ $L$.
Output: a three-layered series $LPO$ $(Y_1, Y_2, Y_3)$ and reversal index : $RIndex$.

$RSplit\_F(L)$ :                                  ▷ Using the book notation, the top part is $Y_1$,
$(Y_1, Y_2, Y_3) \leftarrow Split(L)$             ▷ pos maps element $y_i$ to its position $x_i$ in $L$
$RIndex \leftarrow f(\{pos(y_i) - 1\}_{y_i \in Y_1})$       ▷ $f$ defined in Lemma 1
return $(Y_1, Y_2, Y_3), RIndex$
```

Algorithm 3. Reversible Split algorithm: Reverse computing

```
Input: a three-layered series $LPO$ $(Y_1, Y_2, Y_3)$ and reversal index : $RIndex$.
Output: a discrete $LPO$ $L$.

$RSplit\_R((Y_1, Y_2, Y_3), RIndex)$ :
$n \leftarrow |Y_1| + |Y_2| + |Y_3|$
$X = Extract(RIndex, n, |Y_1|)$
$L \leftarrow [Y_2]$
for $i \leftarrow 2$ to $n$ do
    if $i - 1 \in X$ then
        $L \leftarrow L + Y_1[1]$
        $Del(Y_1[1])$                           ▷ Remove first element from $Y_1$
    else
        $L \leftarrow L + Y_3[1]$
        $Del(Y_3[1])$                           ▷ Remove first element from $Y_3$
    end if
end for
return $L$
```

Example 4. Forward and reverse split on random list $[x, f, m, g, q, s, b, t, p, z]$.



4 Quicksort

We define a reversible Quicksort algorithm, Q', which takes a discrete *LPO* L as an argument and returns a linear *LPO* L^* and an integer between 1 and $|L|!$. Given L^* and the integer, we can recover L.

Algorithm 4. Reversible Quicksort algorithm: Forward computing

```
Input: a discrete $LPO$ $L$
Output: a linear $LPO$ $L^*$ and reversal index

$Q'(L)$ :
if $|L| \leq 1$ then
    return $(L, 1)$
else
    $(Y_1, Y_2, Y_3), C_0 \leftarrow RSplit\_F(L)$    ▷ Using the book notation, the top part is $Y_1$,
                                                    ▷ the pivot is $Y_2$ and the bottom part is $Y_3$.
    $(Y_1, C_1) \leftarrow Q'(Y_1)$                  ▷ Let the code returned be $C_1$
    $(Y_3, C_2) \leftarrow Q'(Y_3)$                  ▷ Let the code returned be $C_2$
    return $([Y_1 : Y_2 : Y_3],\ |Y_3|(|L|-1)! + C_0|Y_1|!|Y_3|! + (C_1 - 1)|Y_3|! + C_2)$
end if
```

We note that $C_2 \in [1, |Y_3|!]$ by assumption, $C_1 \in [1, |Y_1|!]$ by assumption. $C_0 \in [0, \binom{|L|-1}{|Y_1|} - 1]$ from lemma 1, and $|Y_3| \in [0, n-1]$ from the definition of *Split*, and so the min and max values of the code returned are 1 and

$$
\begin{aligned}
&(|L|-1)(|L|-1)! + (\binom{|L|-1}{|Y_1|} - 1)|Y_1|!|Y_3|! + (|Y_1|! - 1)|Y_3|! + |Y_3|! \\
&= (|L|-1)(|L|-1)! + (\frac{(|L|-1)!}{|Y_1|!|Y_3|!} - 1)|Y_1|!|Y_3|! + (|Y_1|! - 1)|Y_3|! + |Y_3|! \\
&= |L|! - (|L|-1)! + (|L|-1)! - |Y_1|!|Y_3|! + |Y_1|!|Y_3|! - |Y_3|! + |Y_3|! = |L|!
\end{aligned}
$$

as expected.

Note that this encoding is the most efficient possible, since all n! different unsorted lists are mapped to the same sorted output.

We briefly outline the intuition for the reversal index returned.

In order to reverse Quicksort, using the recursive structure of the algorithm, we need three pieces of information: (i) the location of the pivot, which is encoded by $|Y_3|$, the number of elements placed below the pivot; (ii) the order in which the nodes above and below the pivot originally appeared, which is encoded by C_0 as outlined in the previous section; and (iii) the information needed to reverse each of the two recursive calls on the upper and lower parts, which are encoded in C_1 and C_2 respectively. We would like to store these four numbers in a way that allows us to recover each of them.

We could store them as a quadruple (a, b, c, d), but then the recursion would mean that c and d were tuples themselves, and the final n-tuple could be very large — so we need to combine them. The way we do this is similar to the different digits in a number. To store 4 numbers a, b, c, and d between 0 and 9, we can compute $N = a*10^3 + b*10^2 + c*10 + d$, and easily extract each one. In the same way, if a, b, c, and d are non-negative and less than some different upper bounds A, B, C, and D (where in the previous case $A = B = C = D = 10$), then we can encode the combination as $N = a*BCD + b*CD + c*D + d$. This is the essence of how we have encoded the reversal index (with some adjustments for codes that range between 1 and n instead of 0 and $n-1$).

We can then extract the digits using floors and ceilings. For example, to extract b from N above, we can use $b = \lfloor N/CD \rfloor - B * \lfloor N/BCD \rfloor$. The first floor expression gives $a*B+b$, because $c/BC + d/BCD$ is the fractional part of N/CD, and similarly the second one gives a. Again, the technique needs to be adapted slightly for codes that range between 1 and n instead of 0 and $n-1$, but the essential idea is the same. We now show how we can use this information to construct a reverse Quicksort algorithm.

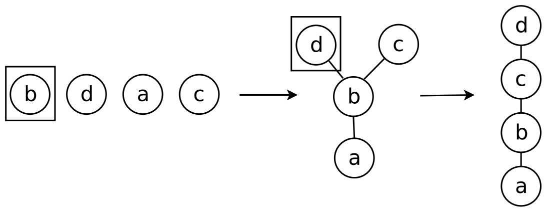
5 Reversing Quicksort

Given a sorted list L^* and a reversal index N, we can reverse Q' to get the input LPO as follows:

Algorithm 5. Reversible Quicksort algorithm: Reverse computing

```
Input: a sorted list $L^*$ and a reversal index $N$
Output: a discrete $LPO$ $L$

$\overline{Q}'(L^*, N)$ :
if $|L^*| \leq 1$ then
    return $L^*$
else
    $k \leftarrow \lceil \frac{N}{(|L^*|-1)!} \rceil$        ▷ $k^{th}$ smallest label in $L^*$ is the first pivot.

    $Y_2 \leftarrow L^*[k]$ $Y_1 \leftarrow L^*[1 : k-1]$ $Y_3 \leftarrow L^*[k+1 : end]$
                                        ▷ : $L^*[a : b]$ means elements from $a$ to $b$

    $C_2 \leftarrow N - (k-1)!\lfloor \frac{N-1}{(k-1)!} \rfloor$ $C_1 \leftarrow 1 + \frac{N - C_2 - (k-1)!(|L^*|-k)!\lfloor \frac{N-1}{(k-1)!(|L^*|-k)!} \rfloor}{k-1!}$
                                        ▷ compute reversal index for upper and bottom parts.

    $Y_1 \leftarrow \overline{Q}'(Y_1, C1)$        ▷ Reverse upper part
    $Y_3 \leftarrow \overline{Q}'(Y_3, C2)$        ▷ Reverse bottom part
    $C_0 \leftarrow \frac{N - (|L^*|-1)!(k-1) - (C_1-1)(k-1)! - C_2}{(k-1)!(|L^*|-k)!}$
    $L \leftarrow RSplit\_R((Y_1, Y_2, Y_3), C_0)$
    return $L$
end if
```

Example 5. Suppose we Quicksort this list $[b, d, a, c]$ as follows:



5.1 Forward Computing

Let's start with Q' first. A split operation applied on the list $[b, d, a, c]$ will partition it into two sublists $[d, c]$ and $[a]$, and the according positions in the original list for sublist $[d, c]$ is $[1, 3]$, thus $C_0 = f(\{x_i\}_{i=1}^2) = \binom{3-1}{2} + \binom{1-1}{1} = 1$. Recursively we look at the upper part, two sublists $[c]$ and empty list. So $C_1 = 1 \times (2-1)! + \binom{1-1}{1} \times 0! \times 1! + (1-1)1! + 1 = 2$. For the down part, because it only has one element, so $C_2 = 1$. Thus the final code returned for this sorting is: $1 \times (4-1)! + 1 \times 2! \times 1! + (2-1)1! + 1 = 6 + 2 + 1 + 1 = 10$. So Q' will give us sorted list $[d, c, b, a]$ and reversal index 10. Next let's reverse the sorting.

5.2 Reverse Computing

First find the first pivot, $k = \lceil \frac{N}{(|L^*|-1)!} \rceil = \lceil \frac{10}{(4-1)!} \rceil = 2$, thus the second smallest label is the pivot, which is b. So Y_1 contains $[d, c]$, Y_3 contains $[a]$. $C_2 = N - (k - 1)! \lfloor \frac{N-1}{(k-1)!} \rfloor = 10 - (2-1)! \lfloor \frac{10-1}{(2-1)!} \rfloor = 1$ and $C_1 = \frac{N - C_2 - (k-1)!(|L^*|-k)! \lfloor \frac{N-1}{(k-1)!(|L^*|-k)!} \rfloor}{k-1!}$ $+1 = \frac{10-1-(2-1)!(4-2)! \lfloor \frac{10-1}{(2-1)! * (4-2)!} \rfloor}{1!} + 1 = \frac{10-1-2\lfloor \frac{9}{2} \rfloor}{1} + 1 = 2$.

Recursively we reverse $Y_1 = [d, c]$ with reversal index 2: pivot $k = \lceil \frac{N}{(|L^*|-1)!} \rceil = \lceil \frac{2}{(2-1)!} \rceil = 2$, thus second smallest element d is the pivot in the sorted sublist $[d, c]$. $C_2 = 2 - (2-1)! \lfloor \frac{2-1}{(2-1)!} \rfloor = 1$, $C_1 = 1 + \frac{2-1-(2-1)!(2-2)! \lfloor \frac{2-1}{(2-1)!(2-2)!} \rfloor}{(2-1)!} = 1$. For its recursive cases, only containing a single element and the empty element, we restore the sorted sublist$[d, c]$ with $C_0 = \frac{2-(2-1)!(2-1)-(1-1)(2-1)!-1}{(2-1)!(2-2)!} = 0$. The final reversed sublist places pivot d in first position, places upper elements (empty) according to the extracted sequence from C_0 (still empty), combined with reversed bottom elements (the single element c). We obtain the reversed unsorted sublist $[d, c]$ for this recursive call.

Similarly, we restore Y_3 with reversal index 1 and obtain the reversed sublist $[a]$.

Finally we combine the two reversed sublists $[d, c]$ and $[a]$ with $C_0 = \frac{10-(4-1)!(2-1)-(2-1)(2-1)!-1}{2!} = 1$. According to Algorithm 1, we can obtain original position indices for $[d, c]$, which is $[1, 3]$ as can be verified in section 5.1. Thus finally we reversed Quicksort and restored the input sequence $[b, d, a, c]$.

6 Conclusion and Future Work

We have shown the frugal encoding underpinning of the reversible $\mathcal{MOQA}$ split operation, where the encoding can be achieved through the bookkeeping of a single number. The applicability of the encoding has been demonstrated via reversible Quicksort.

As mentioned in the introduction, the paper serves to illustrate the principles underpinning the encoding, the generalization of which will be explored to ensure frugal encodings for all of $\mathcal{MOQA}$'s operations, where reversibility of each of the operations has been established in prior work ([SEPV09],[Sch10],[Ear10]). The encoding lifts $\mathcal{MOQA}$ from a language underpinning static average-case analysis to a reversible language, capable of exact average-cost predictions. Future work will focus on extracting the benefits of both aspects, and, on exploring the interesting novel connection between guaranteeing a modular derivation of the average computation cost (a key requirement to develop static timing tools) and reversibility of language operations.

References

[Ben73] Bennet, C.: Logical reversibility of computation. IBM Journal of Research and Development 6, 525–532 (1973)

[Ear10] Early, D.: A Mathematical Analysis of the $\mathcal{MOQA}$ Language. Ph.D. thesis, University College Cork (2010)

[Hic08] Hickey, D.: Tracking data structures for automated average time analysis. Ph.D. thesis, University College Cork (2008)

[Lan61] Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5, 183–191 (1961)

[Sch08] Schellekens, M.: A Modular Calculus for the Average Cost of Data Structuring, 240 p. Springer (2008)

[Sch09] Schellekens, M.: A random bag preserving product operation. Electronic Notes in Theoretical Computer Science 225 (2009)

[Sch10] Schellekens, M.: $\mathcal{MOQA}$; Unlocking the Potential of Compositional Static Average-Case Analysis. Journal of Logic and Algebraic Programming 79(1), 61–83 (2010)

[SEPV09] Schellekens, M., Early, D., Popovici, E., Vasudevan, D.: A high level reversible language for modular average-case analysis. Preliminary Proceedings of the Reversible Computing Workshop-RC 2009, A Satellite Workshop of ETAPS 2009 (2009)

[SHB04] Schellekens, M., Hickey, D., Bollella, G.: $\mathcal{MOQA}$, a Linearly-Compositional Programming Language for (semi-) automated Average-Case Analysis. In: WIP Proceedings, 25th IEEE International Real-Time Systems Symposium, Lisbon, Portugal (2004)

[Tof80] Toffoli, T.: Reversible Computing. In: Proceedings of the 7th Colloquium on Automata, Languages and Programming, pp. 632–644 (1980)

[TDJ10] Ye, T., Vasudevan, D., Chen, J., Popovici, E., Schellekens, M.: Static Average Case Power Analysis for Block Ciphers. In: The 13th EUROMICRO Conference on Digital System Design, EUROMICRO-DSD 2010, Lille, France (2010)

[YG07] Yokoyama, T., Glueck, R.: A Reversible Programming Language and its Invertible Self- Interpreter. In: Proc. ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM, pp. 144–153 (2007)

[Zei99] Zeitz, P.: The Art and Craft of Problem Solving. John Wiley & Sons, Inc. (1999)

Towards a General-Purpose, Reversible Language for Controlling Self-reconfigurable Robots

Ulrik Pagh Schultz

Modular Robotics Lab, University of Southern Denmark
ups@mmmi.sdu.dk
http://www.mmmi.sdu.dk/~ups

Abstract. Self-reconfigurable, modular robots are distributed mechatronic devices that can autonomously change their physical shape. Self-reconfiguration from one shape to another is typically achieved through a specific sequence of actuation operations distributed across the modules of the robot. Automatically reversing the sequence of operations brings the robot back to its initial shape, as has been experimentally demonstrated using the DynaRole reversible language. DynaRole however only allows simple sequences of operations to be reversed, which is suitable for reversing self-reconfiguration sequences but lacks the generality needed to implement more complex behaviors.

In this paper we present initial ideas on generalizing the DynaRole language to support a wider range of modular robot control scenarios, while retaining the possibility of reversing distributed sequences. Reversibility is investigated as a practical feature, reducing the programming task of the programmer, and allowing error recovery by backing out of an error state using reverse execution.

Keywords: modular robots, distributed control, reversible language.

1 Introduction

Modular robotics is an approach to the design, construction and operation of robotic devices aiming to achieve flexibility and reliability by using a reconfigurable assembly of simple subsystems [1]. Robots built from modular components can potentially overcome the limitations of traditional fixed-morphology systems because they are able to rearrange modules automatically on a need basis, a process known as self-reconfiguration, and are able to replace unserviceable modules without disrupting the system's operations significantly. Programming reconfigurable robots is however complicated by the need to adapt the behavior of each of the individual modules to the overall physical shape of the robot and the difficulty of handling partial hardware failures in a robust manner.

In earlier work, we have investigated the distributed execution of a pre-specified self-reconfiguration sequence in a modular robot [2]. A sequence is specified using a simple, centralized scripting language, which either could be

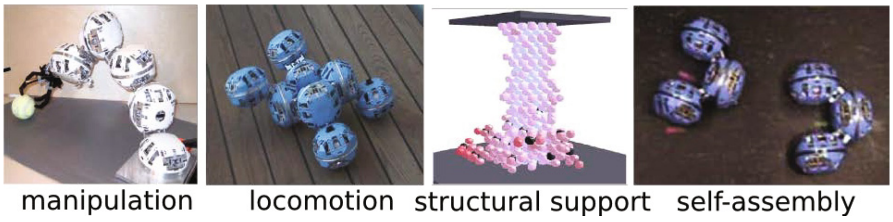


Fig. 1. The ATRON modular robot used for various applications

the outcome of a planner or be hand-coded. The distributed controller generated from this language allows for parallel self-reconfiguration steps and is highly robust to communication errors and loss of local state due to software failures. Furthermore, the self-reconfiguration sequence can automatically be reversed: any self-reconfiguration process described in the language is reversible, subject to physical constraints. The distributed scripting facility is however limited to specifying straightforward sequences of actions, there is no support for specifying when sequences should execute, nor is there support for e.g. conditionals and local state. These limitations prompt the development of an improved language which provides a more general notion of robust, reversible execution.

This paper presents a work-in-progress on providing a generalized notion of reversibility in a programming language for modular robots. The key focus is on providing a robust, distributed execution model that facilitates programming modular robots in the presence of partial hardware failures. Reversibility serves both a practical role in terms of code reuse (the reverse sequence is automatically derived from the forwards sequence) and as a potential improvement in robustness since reverse execution presumably can be used to "back out of" an execution sequence that cannot advance due to partial hardware failures. Concretely, execution sequences are extended with simple pre- and post-conditions over the module structure and context, and an extended representation of the execution state is used to allow sequences to be robustly reversed at any point in their execution. The overall utility of the approach is tested experimentally using a prototype reversible language for modular robots, also described in this paper.

The rest of the paper is organized as follows: we first provide background material on modular robots and the DynaRole language (Sec. 2), then provide a more in-depth discussion of how reversibility could be used more generally in modular robots (Sec. 3), and afterwards present our proposal for general principles for program reversal for modular robots (Sec. 4). Last, we present our proposed language design, explained using simple obstacle avoidance experiment (Sec. 5), and conclude with a discussion of perspectives and future work (Sec. 6).

2 Background: Modular Robots

There are numerous different kinds of modular robots [1]. The ATRON self-reconfigurable modular robot (Fig. 1) is our primary experimental platform.

The ATRON is a 3D lattice-type system [3]. Each unit is composed of two hemispheres, which rotate relative to each other, giving the module one degree of freedom. Connection to neighboring modules is performed by using its four actuated male and four passive female connectors, each positioned at 90 degree intervals on each hemisphere. The likewise positioned eight infrared ports are used to communicate among neighboring modules and to sense distance to nearby objects. Each module is in principle an autonomous robot, but for the robot as a whole to provide useful functionality the actions of the individual modules must be coordinated.

2.1 Self-reconfiguration

Self-reconfiguration concerns the spatial transformation of the robot morphology from one shape to another [1]. It is typically viewed as a sequence of operations performed by the robot; in some cases self-reconfiguration could be the only operation performed e.g. if performing locomotion based on self-reconfiguration. Off-line planning of self-reconfiguration sequences robots has been studied for a large number of different robotic systems [4–13], but is largely complementary to the concerns addressed in this paper: we are interested in providing automatic reversal of self-reconfiguration sequences for the dual purpose of code reuse and increased robustness. On-line, distributed self-reconfiguration algorithms [14–20] allow self-reconfiguration to be done automatically given a target shape. Nevertheless, on-line algorithms are only feasible for modules with relatively few motion constraints, making them less useful for a robot such as the ATRON [12].

As mentioned earlier, the ATRON robot consists of simple one-degree-of-freedom modules. To compensate for the limited motion capabilities, control can be based on *metamodules* composed of more than one module [21]. Metamodules are a logical construct that emerge from other modules, move on the surface of other modules through continuous self-reconfiguration, and stop at a new position. The flow of metamodules, from one place to another on the structure of modules, realizes the desired self-reconfiguration. Programming metamodules has been done using a combination of local actions executed by specific metamodules and gradient information propagated throughout the structure [22].

2.2 The DynaRole Language

DynaRole is a role-oriented language that allows the programmer to use roles to declaratively specify how programs are deployed in the modular robot as a function of its spatial layout and how each module responds to sensor inputs and communication [23]. The use of roles allows behaviors to be organized into units that again are organized into an inheritance hierarchy, providing both reuse and behavioral specialization. To enable DynaRole to be used for self-reconfiguration, we extended the language to support robust execution of distributed sequences of operations [2]. Specifically, self-reconfiguration sequences are compiled to a robust and efficient implementation based on a distributed state machine that continuously shares the current execution state between the modules of the robot.

Dependencies between operations are explicitly stated to allow independent operations to be performed in parallel while enforcing sequential ordering between actions that are physically dependent on each other. The language is *reversible* meaning that for any self-reconfiguration sequence the reverse one is automatically generated, reversal is however subject to physical constraints. As a concrete example, consider a sequence of operations [2]:

```
M0.retract(0)& M3.retract(4); M3.rotate(0,324,0); M4.rotate(0,108,1);
```

Here, modules `M0` and `M3` retract their connectors in parallel, once both operations are complete, module `M3` rotates counterclockwise (indicated by the third argument) from position 0 to position 324 (note that the starting position is not actually checked at runtime), and once this is complete module `M4` rotates clockwise to position 108. The reverse sequence is automatically generated by the DynaRole compiler, as follows:

```
M4.rotate(108,0,0); M3.rotate(324,0,1); M3.extend(4)& M0.extend(0);
```

Now the rotation operations execute first, rotating to position 0 in the opposite directions. Afterwards, the connectors are extended in parallel (extension is the reverse of retraction). As is the case for reversible general-purpose languages, reversibility is enabled using reversible operations [24], which is why the rotation operation also includes the position the actuator is assumed to be starting in.[1]

The continuous diffusion of the state of each module to its neighboring modules provides a high degree of robustness towards partial failures: one-way communication links still serve to propagate state throughout the ensemble, and modules that are reset (e.g., due to hardware issues or by a watchdog-based timer) are automatically restored from the neighboring modules. Nevertheless, the distributed sequences are extremely simple, there are no conditionals, loops, or propagation of any state except how far the sequence has executed.

3 The Role of Reversibility

In earlier work, our basic observation was that for a given scenario, even if a planner was available that could generate the initial self-reconfiguration sequence, it can be useful to work with the sequence from a programming point of view to perform parallelization or to adapt it to constraints in the physical environment. A sequence serves to transform the robot from one configuration to another, and it is usually also relevant to transform the robot back again to the original configuration. This reverse transformation can be accomplished automatically when the sequence language is reversible, as is the case for DynaRole.

[1] As a generalization, in a language having only reversible operations, one could also imagine an operation having only a target position and direction, but returning the current position as determined by the encoder of the actuator, e.g. `p=rotate(108,0);` which can be reversed using the value of `p`.

We hypothesize that reverse execution can also be used to back out of distributed execution sequences that have become blocked, due to unexpected features of the environment, conflicting operations (one module blocking the movements of another), or partial hardware failures. This approach is similar to the notion of reversible computing from Zuliani [25] and Stoddart et al. [26], in particular when conditionals are introduced in the language, as discussed in Sec. 4. In general, reversibility could be used to physically search a state space where one of many possible self-reconfiguration sequences lead to a specific, desired result. We expect that the abstractions from Stoddart et al. will be a useful starting point for such a generalization, this is however left as future work.

Not all physical operations performed by the modules are reversible: The surrounding environment may have changed after a forwards operation, physically blocking the reverse operation from taking place. A single module can only lift two other modules against gravity, but could lower a larger number of modules (although not in a controlled fashion). Completely disconnecting modules from the rest of the robot can cause them to become misaligned, making it impossible to reconnect without a special procedure for actively realigning the modules. Such irreversible operations raise the question of whether a pure reversible model is appropriate. As an alternative, one could for example consider an approach such as bidirectional programming, where state changes are an integral part of the model [27]. A bidirectional transformation would then map one state of the robot to another, and changes to the current state of the robot would implicitly modify the corresponding reverse state reachable by the transformation. Such an approach would work well for control of a mathematical model of the ATRON robot in a known environment, but since the set of potential spurious failures in general is open-ended, and the use of the aforementioned physically irreversible operations is rare, there does not appear to be an obvious advantage to this approach over the pure reversible model.

As a concrete example of using reverse execution, the "structural support" scenario from Fig. 1 uses ATRON modules grouped as metamodules. To move around in the structure, metamodules essentially perform short distributed sequences of operations, depending on their context and the attraction point they are moving towards (a cylinder shape in this example). Collisions between metamodules are common in larger-scale scenarios, and are handled by reversing the current operation sequence to bring the metamodule back to a known configuration. For metamodules this reverse execution was implemented manually, and DynaRole could not have been used since it only supports off-line reversal of complete sequences, not on-line reversal in the middle of the execution of a sequence, as would be required for error handling.

In terms of expressiveness, we are interested in generalizing the language to support adapting the execution of distributed sequences to changes in the environment (other modules in the robot or the surrounding physical environment). We believe this can be done conveniently by providing a mechanism for conditional start of a sequence, conditional evaluation of operations during the sequence, and shared state during sequence execution.

```
(if (and (@LeftWheel (module-nearby 1))        ; pre-condition:
         (@RightWheel (module-nearby 3)))      ; modules nearby
    (begin                                     ; then: dock
       (@LeftWheel (extend-connector 1))
       (@RightWheel (extend-connector 3)))
    'nop                                       ; else: ignore
    (and (@LeftWheel (is-connected 1))         ; post-assertion:
         (@RightWheel (is-connected 3))))      ; modules connected
```

Fig. 2. Docking operation

4 General Principles

As a first step, we propose that a distributed sequence for an ensemble can be written as a behavior with a precondition and a post-assertion. Specifically, a distributed sequence is an *ensemble behavior* that activates when a *precondition* on the state and spatial context are fulfilled, performs a sequence of operations optionally prefixed by the name of the module that performs the operation, and in the end in certain requirements being fulfilled, as specified by a *post-assertion.* (This approach is similar to and inspired by conditionals in Janus [24].)

As a concrete example of requirements and results, consider the program in Fig. 2, which is an (overly simplified) example of docking two ATRON cars (as in "self-assembly" of Fig. 1). Here, the precondition is that the wheel modules detect that a module has been placed between them, in which case they extend their connectors, resulting in them both being in the connected state (as expressed by the fourth element in the if form). Logically, the reverse operation ("undocking") shown in Fig. 3 requires the modules to be in the connected state, and is performed by retracting the same connectors, which results in the modules being nearby. (This is however not guaranteed, after undocking the other vehicle could move away before the post-assertion can be checked, or the sensors might simply report incorrect information.) This approach appears to work for self-reconfiguration in general, and can be seen as a distributed generalization of rule-based self-reconfiguration which has proven to be quite useful in practice, albeit difficult to express concisely using simple rules [28].

We wish to support the execution of sequences that manipulate shared variables and that can be reversed at any point. DynaRole supports execution of a single sequence (without variables) using a shared state tuple (i, p), where i is an invocation counter incremented when a new sequence is started, and p is a program counter for the currently executing sequence. Robust execution requires that a set of modules that were temporarily cut off from the execution but that reestablish contact can unambiguously merge their current execution state with that of the remaining modules, which is ensured by the combination of invocation and program counter. Both counters are monotonic, this scheme however obviously breaks if the sequence starts to execute in reverse. As a generalization, we propose to represent the state of a given distributed sequence using a tuple

```
(if (and (@LeftWheel (is-connected 1))        ; pre-condition
         (@RightWheel (is-connected 3)))      ; modules connected
    (begin                                    ; then: undock
       (@RightWheel (retract-connector 3))
       (@LeftWheel (retract-connector 1)))
    'nop                                      ; else: ignore
    (and (@LeftWheel (module-nearby 1))       ; post-assertion
         (@RightWheel (module-nearby 3))))    ; modules nearby
```

Fig. 3. Hypothetical reversed docking (reverse of Fig. 2)

(i, s, p, d) where i is a per-module invocation counter for the sequence, s is a *step counter* designating how many operations have been performed in the current evaluation of the sequence, p is the program counter, and d is the *execution direction* (forwards or backwards). The currently executing sequence can thus be robustly merged even after on-line reversal in one part of the structure (e.g., due to a local error), since the state with the highest step count must be the newest (barring arithmetic overflow, at which point this scheme will fail). The implementation of shared variables is a significant challenge in distributed programming; we refer to a companion paper for a proposed solution that provides a useful notion of shared variables across modules, similar to global variables but integrated with the concept of role-based control [29]. We expect that these variables will be compatible with reversal similarly to global variables in Janus [24].

The pre-condition on a sequence is simply a conditional that triggers a sequence of distributed steps. Generally, conditionals could be used at any point in the sequence, prefixed with the name of the module evaluating the conditional. Such conditionals however raise a philosophical question: what is the reverse behavior of a robot in a given environment? To provide a preliminary answer to this question, we investigate a simple example of a reversible robot controller written in a minimal, reversible distributed programming language for modular robots.

5 Experiments in μrRoCE

The language μrRoCE (micro reversible robust collaborative ensembles) is intended as a minimalistic, reversible language for programming modular robot systems. The language is currently under development as an embedded DSL in Scheme integrated with the USSR simulator for modular robots [30]; it is based on the RoCE language also currently under development [29, 31], but can nonetheless serve to study the idea of general-purpose reversible programs for modular robots. Distributed control flow in μrRoCE can be based on the tuple-based state representation presented in the previous section, and distributed state sharing can be based on the continuous propagation of updated values from module to module in the physical structure proposed for RoCE [29]. The implementation presented in this paper however works only with simulated robots since it uses a simple, centralized representation of control flow and state that merely simulates the proposed semantics.

5.1 Informal Language Description

The BNF for the μrRoCE language is shown in Fig. 4 (the implementation syntax is more verbose, we use a cleaner syntax for readability). A program is a number of entities: *ensembles* that represent distributed scope and distributed execution across a number of modules, and *roles* that describe individual state and behavior for single modules. Both kinds of entities have members: requirements, state variables, and expressions. At a given point in time, all entities for which all requirement expressions are satisfied on a given module are activated on that module. State variables represent the state of the entity, and are always stored locally on a module; for roles the variables are private to modules on which they are active, for ensembles assigning a variable on one module will eventually (over time and space) propagate that value to the other modules where the same ensemble is active [29].

```
PROGRAM  ::= ENSEMBLE* ROLE*
ENSEMBLE ::= (define-ensemble E MEMBER*)
ROLE     ::= (define-role R (E*) MEMBER*)
MEMBER   ::= (require EXP) | (var V VALUE) | EXP
EXP      ::= (F EXP*) | VALUE | (@R EXP) | X.V
           | (if EXP EXP EXP EXP? ) | (timed VALUE EXP)
           | (begin EXP*) | (transition E.V EXP EXP)
VALUE    ::= NUMBER | 'SYMBOL
```

E∈ENSEMBLE, R∈ROLE, X∈ENSEMBLE∪ROLE, V∈VARIABLE, F∈FUNCTION

Fig. 4. BNF for reversible programming of modular robots (μrRoCE)

Expressions evaluate continuously on all modules where the corresponding entity or role is active. An expression can be the application of a function, a value, the restriction of an expression to only execute on a module of a given role (the enclosing expression can only proceed by evaluating the expression on a module playing the given role), access to a variable from an entity or role (must be active on the module), a conditional with optional post-assertion, timed execution of an expression (after evaluating the expression, execution of the entity blocks for the designated amount of time), an expression sequence, or the transition of a state variable from one value to another (an explicitly reversible update to the state).

As a concrete example of a μrRoCE program, the docking code fragment previously discussed in Sec. 4 is shown implemented[2] in Fig. 5. The ensemble

[2] The simulator currently does not have reliable support for docking, for which reason this program has not been tested in simulation.

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

Fig. 5. Docking of cars in μrRoCE

`DockingCar` has an expression that triggers on a module playing the role `Front`; this expression tests if other modules are close to the wheel modules, and if so extends the connectors (and otherwise does nothing). The post-assertion for the conditional states that the wheels should be connected. As was argued in Sec. 4, this program can be executed in reverse to achieve an undocking behavior: For an ensemble, requirements on activation and state variables function in the same way during reverse execution, but each of the expression are evaluated in reverse using semantics similar to Janus [24].

5.2 Obstacle Avoidance Example

As a more elaborate example, consider obstacle evasion for the small two-wheeler car, shown in Fig. 6. The program consists of an ensemble for the whole car, controlling the overall coordination, and specific roles for each of the modules of the robot. The ensemble `Car` defines the state variables `obstacle` and `drive`, which are used to represent sensory information and the overall driving behavior of the robot. The single expression in `Car` instructs the ensemble as a whole to drive forwards if there are no obstacles, or to perform an evasion behavior for 5 seconds if there is an obstacle. The role `Front` is responsible for reading sensors and updating the shared sensory information correspondingly, whereas the two wheel roles actuate to achieve either forwards or backwards motion depending on the desired overall driving behavior. The flow of information is from the role `Front` (the center of the car) through the state variables `obstacle` and `drive`, that propagate to the wheel modules. The `Car` ensemble executes on all the modules and hence the transition of the `drive` state can happen on one module or on all of the modules in parallel.

The dataflow and actuation during the execution is illustrated as a sequence of steps in Fig. 7. Execution starts with the forward-moving initial state where `Car.obstacle` = `None`, `Car.drive` = `Forward`, and both wheel modules are rotating in their respective forward directions at full speed. Once an obstacle is detected the value of `obstacle` changes, causing `drive` to be updated. This information eventually propagates to the wheel modules, which then start backing up. Once the timed block completes, evaluation continues and restores to normal, forward movement.

```
(define-ensemble Car
  (var obstacle 'None)
  (var drive 'Forward)
  (if (eq? Car.obstacle 'None)
      (transition Car.drive 'Evade 'Forward)
      (timed 5 (transition Car.drive 'Forward 'Evade))))

(define-role Front (Car)
  (require (eq? (connected COMPASS-ANY) 2))
  (if (or (is-proximity FRONT-LEFT)
          (is-proximity FRONT-RIGHT))
      (transition Car.obstacle 'None 'Some)
      (transition Car.obstacle 'Some 'None)))

(define-role LeftWheel (Car)
  (require (and (eq? (connected COMPASS-ANY) 1)
                (eq? (connected COMPASS-EAST) 1)))
  (if (eq? Car.drive 'Forward)
      (rotateContinuous 100)
      (rotateContinuous -100)))

(define-role RightWheel (Car)
  (require (and (eq? (connected COMPASS-ANY) 1)
                (eq? (connected COMPASS-WEST) 1)))
  (if (eq? Car.drive 'Forward)
      (rotateContinuous -100)
      (rotateContinuous 75)))
```

Fig. 6. Obstacle evasion in μrRoCE

None of the conditions have post-assertions. The semantics in this case is that the condition is also used as post-assertion, meaning that both forwards and reverse execution uses the same condition. In this specific program all these conditions are essentially about responses to sensory information or other stimuli, which seems to result in such simplified conditionals. For example, for the role `Front`, the condition essentially tests a sensor and updates a variable correspondingly. The post-assertion could be on the value of the variable, but running the program in reverse would in this case most likely not be meaningful, since it would imply a causal connection between the value of a variable and what information will be read from a sensor. Alternatively, the condition in the ensemble `Car` could perhaps more naturally have taken the state of the variable `drive` into account in a post-assertion, e.g.:

```
  (if (and (eq? Car.obstacle 'None) (eq? Car.drive 'Evade))
      (transition Car.drive 'Evade 'Forward)
      (timed Car 5 (transition Car.drive 'Forward 'Evade))
      (and (eq? Car.obstacle 'None) (eq? Car.drive 'Forward))))
```

	Car		forward/backward?	
step	`obstacle`	`drive`	`LeftWheel`	`RightWheel`
1. normal	`None`	`Forward`	fwd(100)	fwd(100)
2. detect	`Some`	`Forward`	fwd(100)	fwd(100)
3. react	`Some`	`Evade`	fwd(100)	fwd(100)
4. propagate	`Some`	`Evade`	bck(100)	bck(75)
5. evading	`None`	`Evade`	bck(100)	bck(75)
6. continue	`None`	`Forward`	bck(100)	bck(75)
7. propagate	`None`	`Forward`	fwd(100)	fwd(100)

Fig. 7. Sequence of steps in forwards control of the car ensemble. The ordering of the propagate step can differ between executions.

With this implementation, we however see a degenerate behavior during reverse execution: When there are no obstacles, the initial value of `drive` is "forward", and thus changes to "evade", but then immediately afterwards changes to "forward" and suspends evaluation in that state for 5 seconds, after which it repeats itself.

The lack of a test on the value of `drive` before the state transition however causes the problem that a transition may be executed from "evade" to "forward" even when `drive` was already in the "forward" state. For now we allow such transitions to take place: we ignore a transition when the variable already holds the value of the target state. (This is a pragmatic solution that allows simple, reversible controller programs to be written; exploring the implications of this design choice is left as future work.)

5.3 Reverse Obstacle Avoidance

Forwards execution of this controller program causes the robot car to drive forwards until it encounters an obstacle, at which time it moves backwards while turning for 5 seconds, and then resumes its behavior. A natural question is whether executing this controller program in reverse makes sense. What should be the meaning of reversing this program, such that we get a "reverse behavior" for the robot as a whole? A robot uses sensors to observe the world, the controller program decides what actions to perform, and the actuators to (attempt to) perform those actions. Given a sequence of stimuli that cause the robot to take a corresponding sequence of actions, when using reverse controller execution we could naturally expect that presenting the same stimuli in reverse order would cause the robot to reverse the actions.

Concretely, the reversed obstacle avoidance algorithm will cause the vehicle to drive backwards while turning, except when it detects an object, in which case it drives into the object. In more detail, the dataflow and actuation of this reverse execution is shown in Fig. 8. Initially the state is the same as before, but immediately switches to "evasion", until an obstacle is detected at which time the state switches to "forward" and the robot eventually moves forwards, until there are no obstacles at which time it resumes the evasion behavior. Keeping

	Car		forward/backward?	
step	obstacle	drive	LeftWheel	RightWheel
0. initial	None	Forward	fwd(100)	fwd(100)
1. normal	None	Evade	bck(100)	bck(75)
2. detect	Some	Evade	bck(100)	bck(75)
3. react	Some	Forward	bck(100)	bck(75)
4. propagate	Some	Forward	fwd(100)	fwd(100)
5. attacking	Some	Forward	fwd(100)	fwd(100)
6. continue	None	Evade	fwd(100)	fwd(100)
7. propagate	None	Evade	bck(100)	bck(75)

Fig. 8. Sequence of steps in reverse control of the car ensemble. The ordering of the propagate step can differ between executions.

in mind the conceptual framework of Braitenberg vehicles [32], the forwards behavior could be said to be "exploring while evading obstacles", whereas the reverse behavior could be said to be "evading but attacking obstacles".

Interestingly, at least in this specific example, there are two different yet similar ways of achieving a simple reversed behavior: (1) reverse execution with Janus-like semantics applied to each expression (as just described), and (2) forwards execution but with inverted physical operations (rotation in the opposite direction). The resulting behaviors are almost identical: both exhibit the "evade but attacking obstacles" behavior. Reverse execution goes backwards while turning but moves straight forward when sensing an object in front of it, whereas the inverted operation version goes backwards straight and moves forwards while turning when sensing an object. Both the forwards and the two different kinds of reverse behaviors were tested in simulation, for forwards execution static obstacles were used, whereas for reverse execution the "attacking" behavior was triggered by obstacles that were interactively dropped in front of the robot while it was moving backwards.

6 Conclusion and Future Work

Program reversal has previously been shown to be interesting for modular robots, as a means of automatically deriving reverse self-reconfiguration sequences. Moreover, there are obvious applications for error recovery, allowing a self-reconfiguration sequence to back out of a halfway completed sequence. Given that the sequence may be controlled by a general-purpose program, this raises the issue of what it means to reverse general-purpose controllers for modular robots specifically but to some extent also for robotics in general. This paper provides general principles for reversible control of modular robots, namely the use of conditions and post-assertions (as seen elsewhere in reversible computing), and the distributed execution support for general-purpose robust and reversible execution. In addition, based on a simple experiment implemented in the μrRoCE language, we show how reversible execution can provide interesting behaviors for a simple obstacle avoidance controller for a modular robot.

The principles and ideas presented in this paper merely provide initial directions for research in reversible computation for modular robots and robot controllers in general. The immediate future work is to experiment with different controllers implemented in μrRoCE, to see if the same approach to writing reversible controllers works for other robots in other scenarios. On the longer term we are working on an implementation of the complete RoCE language; we however expect that providing a formal semantics for μrRoCE will be a useful step in this direction.

References

1. Yim, M., Shen, W.M., Salemi, B., Rus, D., Moll, M., Lipson, H., Klavins, E., Chirikjian, G.S.: Modular Self-Reconfigurable Robot Systems (Grand Challenges of Robotics). IEEE Robot. Automat. Mag. (March 2007)
2. Schultz, U.P., Bordignon, M., Stoy, K.: Robust and reversible execution of self-reconfiguration sequences. Robotica 29, 35–57 (2011), `http://modular.mmmi.sdu.dk`, accompanying video available at `http://www.youtube.com/watch?v=SYizuooEs7s`
3. Østergaard, E., Kassow, K., Beck, R., Lund, H.: Design of the ATRON lattice-based self-reconfigurable robot. Autonomous Robots 21(2), 165–183 (2006)
4. Pamecha, A., Ebert-Uphoff, I., Chirikjian, G.S.: Useful metrics for modular robot motion planning. IEEE Transactions on Robotics and Automation (13), 531–545 (1997)
5. Kotay, K., Rus, D.: Algorithms for self-reconfiguring molecule motion planning. In: Proc. of the Int. Confe. on Intelligent Robots and Systems, IROS 2000 (2000)
6. Yoshida, E., Murata, S., Kamimura, A., Tomita, K., Kurokawa, H., Kokaji, S.: Motion planning of self-reconfigurable modular robot. In: Proc. of the Int. Symp. on Experimental Robotics (2000)
7. Brandt, D.: Comparison of A$*$ and RRT-connect motion planning techniques for self-reconfiguration planning. In: Proc. of the 2006 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2006), Beijing, China, pp. 892–897 (October 2006)
8. Asadpour, M., Sproewitz, A., Billard, A., Dillenbourg, P., Ijspeert, A.: Graph signature for self-reconfiguration planning. In: 2008 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2008), pp. 863–869 (2008)
9. Asadpour, M., Ashtiani, M.H.Z., Sproewitz, A., Ijspeert, A.: Graph signature for self-reconfiguration planning of modules with symmetry. In: The 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), St. Louis, USA (October 2009)
10. Prevas, K., Unsal, C., Efe, M., Khosla, P.: A hierarchical motion planning strategy for a uniform self-reconfigurable modular robotic system. In: Proceedings of the IEEE International Conference on Robotics and Automation, Washington, DC, vol. 1, pp. 787–792 (October 2002)
11. Ünsal, C., Khosla, P.: A multi-layered planner for self-reconfiguration of a uniform group of I-cube modules. In: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Maoui, Hawaii, vol. 1, pp. 598–605 (2002)

12. Brandt, D., Christensen, D.J.: A new meta-module for controlling large sheets of atron modules. In: Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems, San Diego, California (November 2007)
13. Yim, M., Goldberg, D., Casal, A.: Connectivity planning for closed-chain reconfiguration. In: Proceedings of Sensor Fusion and Decentralized Control in Robotics Systems III, Bellingham, WA, vol. 4196, pp. 402–412. SPIE (2000)
14. Yoshida, E., Murata, S., Kurokawa, H., Tomita, K., Kokaji, S.: A distributed method for reconfiguration of a three-dimensional homogeneous structure. Advanced Robotics (13), 363–379 (1999)
15. Ünsal, C., Kiliccöte, H., Khosla, P.K.: A modular self-reconfigurable bipartite robotic system: Implementation and motion planning. Autonomous Robots (10), 23–40 (2001)
16. Butler, Z., Rus, D.: Distributed planning and control for modular robots with unit-compressible modules. The International Journal of Robotics Research (22), 699–715 (2003)
17. Rosa, M.D., Goldstein, S., Lee, P., Campbell, J., Pillai, P.: Scalable shape sculpting via hole motion: Motion planning in lattice-constrained modular robots. In: Proc. of the 2006 IEEE Int. Conf. on Robotics and Automation, ICRA 2006 (2006)
18. Murata, S., Kurokawa, H., Kokaji, S.: Self-assembling machine, pp. 441–448 (1994)
19. Yim, M., Zhang, Y., Lamping, J., Mao, E.: Distributed control for 3d metamorphosis. Auton. Robots 10(1), 41–56 (2001)
20. Shen, W.M., Salemi, B., Will, P.: Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots. IEEE Transactions on Robotics and Automation 18, 700–712 (2002)
21. Christensen, D., Støy, K.: Selecting a meta-module to shape-change the ATRON self-reconfigurable robot. In: Proceedings of IEEE International Conference on Robotics and Automations (ICRA), Orlando, USA, pp. 2532–2538 (May 2006)
22. Christensen, D.J.: Experiments on fault-tolerant self-reconfiguration and emergent self-repair. In: Proceedings of Symposium on Artificial Life Part of the IEEE Symposium Series on Computational Intelligence, Honolulu, Hawaii (April 2007)
23. Bordignon, M., Stoy, K., Schultz, U.P.: A Virtual Machine-based Approach for Fast and Flexible Reprogramming of Modular Robots. In: Proc. IEEE Int. Conf. on Robotics and Automation (ICRA 2009), Kobe, Japan, May 12-17, pp. 4273–4280 (2009)
24. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: CF 2008: Proc. of the 2008 Conference on Computing Frontiers, pp. 43–54. ACM, New York (2008)
25. Zuliani: Logical reversibility. IBM Journal of Research and Development (6), 807–818 (2001)
26. Stoddart, B., Lynas, R., Zeyda, F.: A virtual machine for supporting reversible probabilistic guarded command languages. Electronic Notes in Theoretical Computer Science 253(6), 33–56 (2010); Proceedings of the Workshop on Reversible Computation (RC 2009)
27. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. ACM Transactions on Programming Languages and Systems (3) (2007)
28. Brandt, D., Ostergaard, E.: Behaviour subdivision and generalization of rules in rule based control of the ATRON self-reconfigurable robot. In: Proceeding of the International Symposium on Robotics and Automation (ISRA), Queretaro, Mexico, pp. 67–74 (September 2004)

29. Schultz, U.: Towards a robust spatial computing language for modular robots. In: Proceedings of the 2012 Workshop on Spatial Computing, Spain (June 2012)
30. Christensen, D.J., Brandt, D., Stoy, K., Schultz, U.P.: A Unified Simulator for Self-Reconfigurable Robots. In: Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2008), France, pp. 870–876 (2008)
31. Schultz, U.: Programming language abstractions for self-reconfigurable robots (poster). Accepted for publication in SPLASH Companion. ACM (October 2012)
32. Braitenberg, V.: Vehicles: Experiments in Synthetic Psychology. MIT Press (1986)

Reversible and Quantum Circuit Optimization: A Functional Approach

Zahra Sasanian and D. Michael Miller

Department of Computer Science
University of Victoria
Victoria, BC
Canada V8W 3P6
{sasanian,mmiller}@uvic.ca

Abstract. The circuits produced by reversible and quantum synthesis approaches are not often optimal and post synthesis optimizations are beneficial. This paper introduces a functional approach for the optimization of reversible and quantum circuits that uses a recently introduced structure for semi-classical quantum circuits called Decision Diagram for a Matrix Function (DDMF). Experimental results are given that show that using DDMFs leads to more optimizations than are found using existing approaches.

1 Introduction

The synthesis of reversible and quantum circuits has been studied extensively over the last two decades [1,4–6,10,17,23]. Synthesis tools often generate circuits that are not optimal and can be improved by post synthesis optimizations [2, 11,22]. Optimization methods are usually based on gate rearrangement which is restricted by gate moving rules. The conventional moving rule for reversible and quantum gates was proposed by Maslov [8] and was further improved by Sasanian *et al* [18]. The method in [18] uses labels to keep track of function changes between gates to be used by the modified moving rule. Although this approach is fast, effective and straightforward, it cannot find all the possible reductions in a circuit.

In this paper, we improve the approach in [18] to use a functional description instead of labels to represent functions on circuit line segments. The functional representation that we use is a decision diagram structure called Decision Diagram for a Matrix Function (DDMF) [27,28]. The limitation of this representation is that it is only applicable to Boolean reversible circuits and Semi-Classical Quantum Circuits(SCQC). In this paper, we show that the proposed approach can find more reductions than the approach in [18]. However, due to its limitation, the approach can only be used for reversible circuits such as Mixed-Polarity Multiple-Control Toffoli (MPMCT) circuits and non-entangled quantum circuits. The qubits of a quantum circuit are in an entangled quantum state if it is impossible to separate the contributions of the states of the individual qubits from the state of the whole system [7]. Non-entangled NCV (NOT, CNOT, V, V^+, controlled-V, and controlled-V^+) circuits are achieved by using non-entangled quantum realizations of MPMCT gates in mapping MPMCT circuits to quantum circuits. Removing the entanglement adds extra quantum costs that are sometimes comparable to the improvements achieved by the new method.

The rest of the paper is organized as follows. Section 2 gives the necessary background. Section 3 outlines the basic optimization method. The functional approach to circuit optimization is introduced in section 4. Experimental results are given in Section 5 and section 6 concludes the paper.

2 Background

A multiple-output Boolean function is called ***reversible*** if it maps each input assignment to a unique output assignment. A reversible function is realized by a cascade of reversible gates with no fan-out or feedback [16]. A completely or incompletely-specified irreversible function can be embedded into a reversible function, often requiring added constant inputs and/or garbage outputs, and then realized by a reversible circuit [14].

A ***Mixed-Polarity Multiple-Control Toffoli (MPMCT)*** gate with ***target line*** x_t, ***positive controls*** $\{x_{i_1}, x_{i_2} \cdots x_{i_k}\}$ and ***negative controls*** $\{x_{i_{k+1}} \cdots\ x_{i_m}\}$ maps x_t to $(x_{i_1} x_{i_2} \cdots x_{i_k} \overline{x_{i_{k+1}}} \cdots \overline{x_{i_m}}) \oplus x_t$. The ***size*** of an MPMCT gate is the number of controls plus one. An MPMCT gate with no control is the well-known ***NOT*** gate. An MPMCT gate with a single positive control is called a ***Controlled-NOT (CNOT)*** gate. An MPMCT gate with two positive controls is the original gate introduced by Toffoli [25] We use $T(C;t)$ to denote the MPMCT gate with C being the set of controls and t being the target. If not specified, controls are positive. We use an over bar to indicate a negative control. To draw an MPMCT gate, we use the conventional notation $\bigoplus$ to indicate the target line, a $\bullet$ to show a positive control connection and a $\circ$ to indicate a negative control connection.

A ***controlled-V gate*** applies the transformation defined by the matrix in Equation 1 (a) to the target line if its control line is set to 1. A ***controlled-V^+ gate*** changes the target line using the transformation shown in Equation 1 (b) if its control line is 1. Gates V and V^+ are referred to as ***controlled-square-root-of-NOT*** gates since $\mathbf{V}^2 = (\mathbf{V}^+)^2 = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. Note that for this work, the controls for V and V^+ gates are always positive. For drawing V and V^+ gates, a box containing the appropriate symbol is placed on the target line.

$$\mathbf{V} = \frac{1+i}{2}\begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} \qquad (a) \tag{1}$$

$$\mathbf{V}^+ = \frac{1-i}{2}\begin{pmatrix} 1 & i \\ i & 1 \end{pmatrix} \qquad (b)$$

The so-called ***NCV library*** contains the NOT, CNOT, V, V^+, controlled-V, and controlled-V^+ gates with positive controls. All gates in the NCV library are taken to have unit cost. The ***quantum cost*** of a circuit of NCV gates is thus the number of gates in the circuit.

Property 1. MPMCT gates are self-inverse and two consecutive identical MPMCT gates cancel each other and yield the identity mapping. V and V^+ gates with the same target and the same control line are the inverse of each other.

Property 2. Given a cascade of quantum gates $G_1 G_2 \ldots G_k$ realizing the reversible function F, the cascade $G_k^{-1} \ldots G_2^{-1} G_1^{-1}$ realizes the function F^{-1}, where G_i^{-1} is the inverse gate for G_i.

Property 3. Since an MPMCT gate is self-inverse applying Property 2 to a realization of the gate yields an alternate realization for the same gate which is called the ***reverse*** realization.

A circuit line not used as the target or as a control of a gate is an *ancillary line* for that gate. Such lines are used in realizing a MPMCT gate using quantum gates, see [3,13].

A ***Semi-Classical Quantum Circuit (SCQC)*** is a quantum circuit in which if all the initial input quantum states of the circuit are in the base states $|1\rangle$ or $|0\rangle$ (classical values), the quantum states at all gate controls in the circuit are also in the base states $|1\rangle$ or $|0\rangle$ [27]. Entanglement does not occur in SCQCs as long as their inputs are initialized to classical values.

A ***matrix function*** with n Boolean variables $x_1, \ldots, x_n$ is a mapping from $\{0,1\}^n$ to 2×2 unitary matrices. A matrix function $mf(x_1, \ldots, x_n)$ is called ***a constant matrix function*** if $mf(x_1, \ldots, x_n)$ is the same matrix (M) for all assignments to $x_1, \ldots, x_n$ and is denoted by $CM(M)$ [27].

Let mf_1, mf_2 and mf_3 be matrix functions with respect to $x_1, \ldots, x_n$. Then $mf_1 \oplus mf_2$ is defined as a matrix function mf such that $mf(x_1, \ldots, x_n) = mf_1(x_1, \ldots, x_n) \cdot mf_2(x_1, \ldots, x_n)$ where $\cdot$ means normal matrix multiplication. Also, let f be a Boolean function with respect to $x_1, \ldots, x_n$. Then $f * mf_3$ is a matrix function which equals $mf_3(x_1, \ldots, x_n)$ when $f(x_1, \ldots, x_n) = I$ and equals I when $f(x_1, \ldots, x_n) = 0$. For more information regarding matrix functions please refer to [27,28].

3 Basic Optimization Approach

Most optimization methods use gate rearrangement to find possible reductions in reversible and quantum circuits [2,9,12]. So far, the gate rearrangements have been performed under the limitations imposed by the traditional moving rule [8] according to which gate $T(C_1, t_1)$ can be interchanged with gate $T(C_2, t_2)$ in a reversible circuit iff $C_1 \cap t_2 = \oslash$ and $C_2 \cap t_1 = \oslash$. In [18,21], gate rearrangement constraints were modified to allow gate movements beyond the traditional moving rule limits. The following property defines the new moving rule proposed in [18,21].

Property 4. ***Generalized Moving Rule:*** A gate can be moved from one end of a cascade of gates to the other end if its controls are on lines that have the same functionality at each end of the cascade and its target is on a line that has no control connection within the cascade.

Using the *Generalized Moving Rule* requires knowing the functionality for each segment of the circuit lines. In [18, 21], the functions at different spots on a circuit are represented by labels that are assigned using a *Line Labeling Procedure* (Procedure 1 in [18]). This labeling is such that if two segments on a circuit line have the same label, those segments realize the same function. The procedure uses stacks to keep track of identity structures. For more details please refer to [18].

We here describe the optimization procedure used in [20, 22] to optimize a given circuit using the generalized moving rule and Line Labeling Procedure for gate rearrangement. In this procedure, the circuit is considered as a cascade of gates $G_1, G_2, ..., G_N$ where N is the number of gates in the circuit.

Procedure 1 Basic Optimization

For $p = 1$ to N apply the following steps:

1. Label the circuit line segments after the gate G_p using the Line Labeling Procedure.
2. Create an empty list called *ReductionList* and add G_p to it.
3. For $q = p - 1$ to 1, if G_p can be made adjacent to G_q using the generalized moving rule (Property 4) and the labels from step 1:
   (a) If G_p and G_q are identical MPMCT gates or both belong to the NCV library, add G_q to the *ReductionList*.
   (b) If G_p and G_q are both MPMCT gates and can be reduced using the reduction rules 1 to 5 in [22], keep a record of G_q.
4. If $len(ReductionList) > 1$, remove the gates that are in the *ReductionList* from the circuit and substitute the earliest gate with an equivalent optimal cascade, set $p = q$ and go to 1
5. Otherwise, if any MPMCT gate has been found in step 3.(b), remove G_p from the circuit, find its equivalent optimized sub-circuit using the reduction rules 1 to 5 in [22], and substitute G_q with the resulting sub-circuit. Set $p = q$ and go to 1.

The optimization method just described, uses labels to encode the functionality for each line segment in a circuit. Despite being simple and fast, this approach is able to find most of the possible reductions in a circuit as is shown in the experimental results below. However, the procedure is not guaranteed to find all possible reductions and there are cases where identities cannot be found using this method. In the following, two examples are shown where using the Line Labeling Procedure does not find all possible reductions.

Example 1. Consider the circuits shown in Fig. 1. Two consecutive identical MPMCT gates are shown in Fig. 1 (a). They realize the identity according to Property 1. In Fig. 1 (b), the first MPMCT gate is replaced by its five gate realization from the NCV library and since controls can have any order in MPMCT gates, the second MPMCT gate is replaced by the reverse realization (Property 3) with the controls interchanged. In the resulting circuit, $V^+(a; c)$ from the first realization can be canceled with $V(a; c)$ from the second realization and $V^+(b; c)$

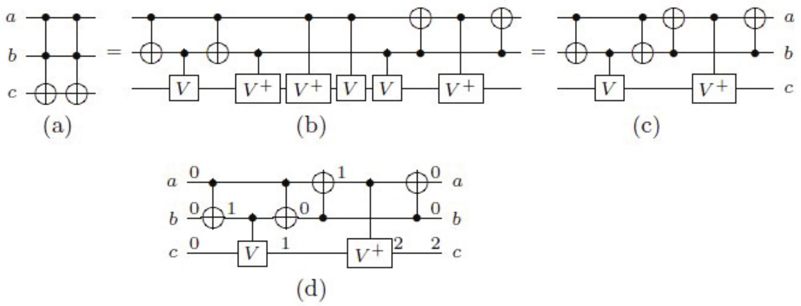


Fig. 1. Example 1: (a) Two MPMCT gates realizing the identity (b) After substituting the NCV realizations (c) After removing the four redundant gates in the middle (d) After applying the Line Labeling procedure

from the first realization can be canceled with $V(b;c)$ from the second realization. This yields the circuit shown in Fig. 1 (c). Since the circuit in Fig. 1 (c) is equivalent to the circuit in Fig. 1 (a), it realizes the identity. Fig. 1 (d) shows the result of applying the Line Labeling Procedure to the circuit of Fig. 1 (c). Note that only changes on the line labels are shown. As is shown, the labels on line c are not the same at the two ends of the circuit although they represent the same function.

Example 2. Consider the two consecutive Swap gates shown in Fig. 2 (a) realizing the identity function. Fig. 2 (b) shows the circuit resulting from substituting these gates with NCV realizations in two ways. As shown in Fig. 2 (c), applying the Line Labeling Procedure leads to different labels at the two ends of the circuit while both ends represent the same function.

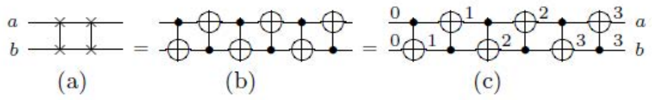


Fig. 2. Example 2: (a) Two Swap gates realizing the identity (b) After mapping to NCV gates (c) After applying the Line Labeling procedure

These two examples show that the basic optimization method that uses the Line Labeling Procedure to find the identities cannot find all the possible reductions in a circuit. In the next section, we discuss an alternative approach that uses functional representations instead of labels and allows more effective implementation of the generalized moving rule.

4 DDMF-Based Optimization Method

It was shown in the previous section that the basic optimization approach cannot find all possible reductions in a circuit because labels do not uniquely represent the functionality of line segments. One way to address this problem is to use a full representation of the functionality for each line segment in the circuit. In this section, an alternative to the basic optimization method is presented that uses a decision diagram structure called Decision Diagram for a Matrix Function (DDMFs) [27,28] for this purpose. DDMFs are compact structures and they are well suited to represent functionality at each circuit line segment. The only issue with this representation is that it only applies to SCQC circuits. However, it is still a good choice since in our application, all of the reversible circuits and most of the NCV circuits are SCQC.

4.1 DDMFs for Reversible and Quantum Circuits

Before discussing the new optimization approach, we briefly introduce DDMFs and show how they are built for reversible and quantum gates and circuits.

A DDMF is a directed acyclic graph with three types of nodes: (1) A single terminal node corresponding to the identity matrix I, (2) a root node with an incoming edge having a weighted matrix M, and (3) a set of non-terminal (internal) nodes. Each internal and the root node are associated with a Boolean variable x_i, and have two outgoing edges which are called 1-edge (solid line) leading to its 1-child node and 0-edge (dashed line) leading to its 0-child node. Every edge has an associated matrix. A DDMF is ordered, i.e., variables appear in the same order on each path from the root node to the terminal node.

The matrix function represented by a node is defined recursively by the following three rules:

(l) The matrix function represented by the terminal node is the constant matrix function $CM(I)$.

(2) The matrix function represented by an internal node (or the root node) whose associated variable is x_i is defined as $x_i{*}(CM(M_1){\oplus}mf_1){\oplus}\overline{x_i}{*}(CM(M_0){\oplus} mf_0)$, where mf_l and mf_0 are the matrix functions represented by the 1-child node and the 0-child node respectively, and M_1 and M_0 are the matrices of the 1-edge and the 0-edge, respectively.

(3) The root node has one incoming edge that has a matrix M. The matrix function represented by the whole DDMF is $CM(M) \oplus mf$, where mf is the matrix function represented by the root node. (See an illustration of this structure in Fig. 3)

Similar to conventional binary decision diagrams (BDD), a canonical form can be defined for DDMFs. For more information on DDMFs refer to [27,28].

We use DDMFs to represent matrix functions on all line segments in a circuit. To this end, first, for each primary input x_i a DDMF representing that variable is built. Then, gates are considered one at a time from the primary inputs towards the outputs, and DDMFs for the functions corresponding to the circuit lines on

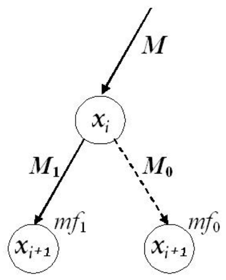


Fig. 3. An internal DDMF node

the output side of the gate are constructed from DDMFs on the input side. Let D_i^j be the DDMF for the $i-$th line after the $j-$th gate and $F(D)$ be the matrix function represented by a DDMF D. Then the DDMFs after the $j-$th gate are built as follows:

1. If the $i-$th line is not the target bit of the $j-$th gate, assign $D_i^j = D_i^{j-1}$.
2. If the $i-$th line is the target of the $j-$th gate, assign $D_i^j = D_i^{j-1} \oplus D_{gate}$ where D_{gate} is constructed by the following steps:
   (a) Assume the $j-$th gate has positive controls $p_1, p_2, \ldots, p_k$ and negative controls $n_1, n_2, \ldots, n_l$. The matrix function for the controls (g) is constructed by $g = F(D_{p_1}^{j-1}) \cdot F(D_{p_2}^{j-1}) \ldots F(D_{p_k}^{j-1}) \cdot \overline{F(D_{n_1}^{j-1})} \cdot \overline{F(D_{n_2}^{j-2})} \ldots \overline{F(D_{n_l}^{j-1})}$. Because of the restriction of SCQCs, all the matrix functions for the controls are classical Boolean functions and the above expression is simply the logical AND.
   (b) The D_{gate} is constructed by $D_{gate} = (DDMF$ for $g) * (DDMF$ for $CM(U))$, where U is a unitary matrix associated with the $j-$th gate.

For more information regarding the DDMFs for reversible and quantum circuits, please refer to [27].

4.2 The Functional Optimization Method

The following procedure describes the DDMF-based optimization approach. In this procedure, the circuit is considered as a cascade of gates $G_1, G_2, ..., G_N$ where N is the number of gates in the circuit.

Procedure 2 DDMF-Based Optimization

For $p = 1$ to N apply the following steps:

1. Build the DDMFs for the circuit lines after the G_p using the method described in the previous section.
2. If the DDMFs on the circuit lines after G_p match DDMFs on the primary inputs, remove the cascade of gates $G_1 \ldots G_p$ from the circuit, set $p = 1$ and go to 1.

3. For $q = 1$ to $p - 1$, if the DDMFs on the circuit lines after G_q match the DDMFs on the corresponding lines after G_p, remove the cascade of gates $G_{q+1} \dots G_p$ from the circuit, set $p = q + 1$ and go to 1.
4. Create an empty list called *ReductionList* and add G_p to it.
5. For $q = p - 1$ to 1, if G_p can be made adjacent to G_q by applying the generalized moving rule (Property 4) using DDMFs for comparing functions:
   (a) If G_p and G_q are identical MPMCT gates or both belong to the NCV library, add G_q to the *ReductionList*.
   (b) If G_p and G_q are both MPMCT gates and can be reduced using the reduction rules 1 to 5 in [22], keep a record of G_q.
6. If $len(ReductionList) > 1$, remove the gates that are in the *ReductionList* from the circuit and substitute the earliest gate with an equivalent optimal cascade, set $p = q$ and go to 1
7. Otherwise, if any MPMCT gate has been found in step 5.(b), remove G_p from the circuit, find its equivalent optimized sub-circuit using the reduction rules 1 to 5 in [22], and substitute G_q with the resulting sub-circuit. Set $p = q$ and go to 1.

In Steps 2 and 3 of the procedure, identity blocks are removed from the circuit. As a result, the circuits of Examples 1 and 2 are handled properly by this procedure. In Steps 4 to 7 of the procedure, more reductions are searched by moving gates using the generalized moving rule and DDMFs.

5 Experimental Results

Prototype implementations of the basic and the DDMF-based optimization procedures have been implemented using the Python and C++ programming languages respectively. The experiments were run on a system with a 3.2 GHz i5-650 CPU and 3.0 GB RAM. The programs use REVLIB [26] *real* circuit format.

Our first test suite consists of 22 MPMCT circuits that are produced by the QMDD synthesis method in [23,24] and optimized by the ESOP-based optimization EXORCISM-4 [15] as reported in [22]. Our second test suite consists of three unstructured reversible functions (urf) taken from the REVLIB web site [26].

To compare the basic and DDMF-based optimization procedures, both optimization methods were applied to the MPMCT circuits in the test suites. The results are mapped to NCV circuits and the optimization procedures were again applied to the NCV circuits. For mapping MPMCT to NCV circuits, we used the mapping procedure introduced in [19] with an extension to generate non-entangled NCV circuits to comply with the limitation of using DDMFs (*i.e.* being applicable on the SCQCs only). To this end, the target of an MPMCT gate is never used as an ancillary as it is mapped to an NCV circuit. The resulting realizations are somewhat more expensive than the entangled realizations reported in [19]. Tables 1 and 2 show the non-entangled NCV costs of MPMCT gates with 1 and $n - 3$ ancillaries respectively. Space does note allow showing the tables for the case of $1 < a < n - 3$ ancillaries but our approach makes full use of those cases when applicable.

Table 1. Non-entangled NCV cost of MPMCT gates of size $n = 4 \ldots 16$ with 1 ancillary

Controls	Number of Negative Controls															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	14	14	16	18												
4	20	20	20	22	24											
5	42	42	42	44	46	48										
6	54	54	54	54	56	58	60									
7	72	72	72	72	76	78	80	82								
8	84	84	84	84	84	86	88	90	92							
9	108	108	108	108	108	110	112	114	116	118						
10	132	132	132	132	132	132	134	136	138	140	142					
11	156	156	156	156	156	156	158	160	162	164	166	168				
12	180	180	180	180	180	180	180	182	184	186	188	190	192			
13	204	204	204	204	204	204	204	206	208	210	214	216	218	220		
14	228	228	228	228	228	228	228	228	230	232	234	238	240	242	244	
15	252	252	252	252	252	252	252	252	254	256	258	260	264	268	270	272

Table 2. Non-entangled NCV cost of MPMCT gates of size $n = 4 \ldots 16$ with $n - 3$ ancillary lines

Controls	Number of Negative Controls															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	14	14	16	18												
4	20	20	20	22	24											
5	32	32	32	34	36	38										
6	44	44	44	44	46	48	50									
7	56	56	56	56	58	60	62	64								
8	68	68	68	68	68	70	72	74	76							
9	80	80	80	80	80	82	84	86	88	90						
10	92	92	92	92	92	92	94	96	98	100	102					
11	104	104	104	104	104	104	106	108	110	112	114	116				
12	116	116	116	116	116	116	116	118	120	122	124	126	128			
13	128	128	128	128	128	128	128	130	132	134	136	138	140	142		
14	140	140	140	140	140	140	140	140	142	144	146	148	150	152	154	
15	152	152	152	152	152	152	152	152	154	156	158	160	162	164	166	168

We here consider three different approaches: (1) the basic method for entangled circuits, (2) the basic method for non-entangled circuits and (3) the DDMF-based method for non-entangled circuits. The results of applying these methodologies are summarized in Table 3. The first column of the table gives the REVLIB circuit name and file identification number. The column labeled *Initial* shows the non-Entangled NCV cost of the initial MPMCT circuits found by summing the costs of the individual gates. The next 12 columns give the result of applying the basic optimization on entangled circuits, the basic optimization on non-entangled circuits and the DDMF-based optimization on non-entangled circuits. For each method, the results are reported in four columns:

(a) The NCV cost of the circuits after applying the corresponding optimization method on the initial MPMCT circuits again found by summing the NCV costs of the individual gates.
(b) The NCV cost of the circuits after mapping MPMCT circuits to NCV circuits using the method in [19].
(c) The NCV costs after applying the corresponding optimization method on the circuits of column (b).
(d) The total elapsed time in seconds for steps (a)-(c).

Table 3. The results of applying the basic and DDMF-based optimization methods on MPMCT circuits

		Basic Entangled				Basic Non-Entangled				DDMF					
Circuits	Initial	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)	Δ_1	Δ_2
adr4_197	535	161	161	161	1.351	161	161	161	1.056	161	161	158	0.923	3	3
clip_206	4061	2640	2485	2465	12.368	2640	2485	2465	11.983	2640	2485	2449	16.922	16	16
cm152a_212	165	165	143	139	0.330	165	143	139	0.283	165	143	139	0.454	0	0
cm42a_207	226	226	200	194	0.565	226	200	194	0.534	226	200	194	0.439	0	0
cm85a_209	1976	502	486	482	1.441	502	486	482	1.620	502	486	475	1.517	7	7
cycle10_2_110	1398	852	850	818	0.711	916	914	880	0.744	916	914	877	1.045	3	-59
dc1_220	239	239	206	200	0.706	239	206	200	0.629	239	206	200	0.798	0	0
dc2_222	1412	1274	1130	1116	5.274	1274	1130	1116	5.369	1274	1130	1111	5.328	5	5
max46_240	2874	2182	1994	1987	5.236	2568	2198	2192	6.397	2568	2198	2182	7.437	10	-195
misex1_241	638	625	539	531	2.365	625	539	531	2.224	625	539	526	2.109	5	5
plus63mod4096	749	665	651	638	1.252	749	705	701	1.299	749	705	699	1.298	2	-61
plus63mod8192	955	849	809	794	1.252	955	884	866	1.502	955	884	864	1.642	2	-70
radd_250	536	226	208	208	0.965	226	208	208	0.965	226	208	204	1.047	4	4
rd73_252	823	788	692	674	6.962	788	692	674	6.671	788	692	661	6.859	13	13
rd84_253	1693	1577	1440	1417	9.364	1577	1440	1417	9.294	1577	1440	1405	10.436	12	12
root_255	2248	1866	1719	1689	5.814	1866	1719	1689	5.533	1866	1719	1677	6.702	12	12
sqn_258	1128	940	876	862	2.107	940	876	862	2.262	940	876	858	2.626	4	4
sqrt8_260	500	456	404	391	1.348	456	404	391	1.270	456	404	389	1.391	2	2
squar5_261	301	292	263	263	1.115	292	263	263	1.269	292	263	263	2.058	0	0
sym9_193	3859	3159	2928	2904	18.998	3645	3191	3156	16.177	3645	3191	3141	22.765	15	-237
wim_266	186	171	159	157	0.426	171	159	157	0.300	171	159	157	0.281	0	0
z4_268	521	412	406	400	2.433	412	406	400	2.227	412	404	390	2.359	10	10
urf1_278	16130	15687	15585	15525	195.042	15791	15689	15629	196.025	15787	15685	15469	33703.485	160	56
urf2_277	6711	6525	6475	6445	48.147	6555	6505	6475	46.992	6555	6505	6473	1031.906	2	-28
urf5_280	13613	13295	13213	13165	112.336	13439	13357	13309	111.624	13435	13353	13212	13020.905	97	-47

The last two columns of Table 3 show the difference between the three approaches in terms of the quantum cost. Δ_1 is the difference between the ninth column (Basic Non-Entangled (c)) and the thirteenth column (DDMF (c)). In fact, Δ_1 gives the improvement that can be achieved by using the DDMF-based optimization method rather than the basic optimization method for non-entangled circuits. Δ_2 gives the difference between the fifth column (Basic Entangled (c)) and the thirteenth column (DDMF (c)). It compares the two options of using the basic method on entangled circuits and the DDMF-based method on non-entangled circuits.

The results are interesting in that they show that using a functional approach does not in general add significant improvement to what is achieved by the basic method. However, in some cases (*e.g.* the clip_206 circuit) the DDMF-based method outperforms the basic entangled method despite using the expensive non-entangled cost function. The CPU time of the DDMF-based method is significantly more than the basic method as expected.

6 Conclusion

We introduced an optimization approach that uses a functional description to implement the generalized moving rule as opposed to the labels in [18]. The results presented show that using a functional approach does not result in considerable cost savings while it is computationally expensive. However, the trade-offs between the techniques presented can be considered depending on the target circuits. For example, for small SCQCs, the DDMF-based approach is definitely the best choice while for larger circuits the basic method is more practical.

Our experiments to date have been for MPMCT circuits mapped to NCV circuits using a particular approach to the mapping of each MPMCT gate. It may be that the DDMF method will yield more improvement for alternative mapping approaches or for SCQC found by other approaches. This is a consideration for ongoing research.

Acknowledgements. This work was partially supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. The authors wish to thank S. Yamashita for providing the source code for his DDMF package.

References

1. Alhagi, N., Hawash, M., Perkowski, M.: Synthesis of reversible circuits with no ancilla bits for large reversible functions specified with bit equations. In: Proc. Int'l Symp. on Multiple-valued Logic, pp. 39–45 (2010)
2. Arabzadeh, M., Saeedi, M., Zamani, M.S.: Rule-based optimization of reversible circuits. In: Proc. ASP Design Automation Conf., pp. 849–854 (2010)
3. Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D.P., Margolus, M., Shor, P., Sleator, T., Smolin, J.A., Weinfurter, H.: Elementary gates for quantum computation. Physical Review A 52(5), 3457–3467 (1995)

4. Golubitsky, O., Falconer, S.M., Maslov, D.: Synthesis of the optimal 4-bit reversible circuits. In: Proc. Design Automation Conf., pp. 653–656 (2010)
5. Kerntopf, P., Perkowski, M., Podlaski, K.: Synthesis of reversible circuits: A view on the state-of-the-art. In: Proc. IEEE Int'l Conf. on Nanotechnology (2012)
6. Lukac, M., Perkowski, M., Kameyama, M.: Evolutionary quantum logic synthesis of Boolean reversible logic circuits embedded in ternary quantum space using structural restrictions. In: Proc. Int'l Conf. on Evolutionary Computation, pp. 1–8 (2010)
7. Marinescu, D.C., Marinescu, G.M.: Approaching Quantum Computing. Pearson Education (2004)
8. Maslov, D.: Reversible Logic Synthesis. Ph.D. thesis, University of New Brunswick (2003)
9. Maslov, D., Dueck, G.W., Miller, D.M.: Simplification of Toffoli networks via templates. In: Symp. on Integrated Circuits and System Design, pp. 53–58 (2003)
10. Maslov, D., Dueck, G.W., Miller, D.M.: Synthesis of Fredkin-Toffoli reversible networks. IEEE Trans. on VLSI Systems 13, 765–769 (2005)
11. Maslov, D., Saeedi, M.: Reversible circuit optimization via leaving the Boolean domain. IEEE Trans. on CAD 30(6), 806–816 (2011)
12. Maslov, D., Young, C., Miller, D.M., Dueck, G.W.: Quantum circuit simplification using templates. In: Proc. Design, Automation and Test in Europe, pp. 1208–1213 (2005)
13. Miller, D.M.: Lower cost quantum gate realizations of multiple-control Toffoli gates. In: IEEE Pacific Rim Conference, pp. 308–313 (2009)
14. Miller, D.M., Wille, R., Dueck, G.W.: Synthesizing reversible circuits from irreversible specifications using Reed-Muller spectral techniques. In: Proc. Reed-Muller Workshop, pp. 87–96 (2009)
15. Mishchenko, A., Perkowski, M.: Fast heuristic minimization of exclusive sum-of-products. In: Proc. 5th Int'l Reed-Muller Workshop, pp. 242–250 (2001)
16. Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information. Cambridge Univ. Press (2000)
17. Saeedi, M., Wille, R., Drechsler, R.: Synthesis of quantum circuits for linear nearest neighbor architectures. Quantum Information Processing 10(3), 355–377 (2011)
18. Sasanian, Z., Miller, D.M.: Mapping a multiple-control Toffoli gate cascade to an elementary quantum gate circuit. In: Proc. Workshop on Reversible Computation, pp. 83–90 (2010)
19. Sasanian, Z., Miller, D.M.: NCV realization of MCT gates with mixed controls. In: Proc. Pacific Rim Conf. on Communications, Computers and Signal Processing, pp. 567–571 (2011)
20. Sasanian, Z., Miller, D.M.: Transforming MCT Circuits to NCVW Circuits. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 77–88. Springer, Heidelberg (2012)
21. Sasanian, Z., Miller, D.M.: A new methodology for optimizing quantum realizations of reversible circuits (in preparation, 2012)
22. Soeken, M., Sasanian, Z., Wille, R., Miller, D.M., Drechsler, R.: Optimizing the mapping of reversible circuits to four-valued quantum gate circuits. In: Proc. Int'l Symp. on Multiple-valued Logic, pp. 173–178 (2012)
23. Soeken, M., Wille, R., Hilken, C., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: Proc. Workshop on Reversible Computation, pp. 59–70 (2011)
24. Soeken, M., Wille, R., Hilken, C., Przigoda, N., Drechsler, R.: Synthesis of reversible circuits with minimal lines for large functions. In: Proc. ASP Design Automation Conf., pp. 85–92 (2012)

25. Toffoli, T.: Reversible computing. Tech. Memo LCS/TM-151, MIT Lab. for Comp. Sci. (1980)
26. Wille, R., Große, D., Teuber, L., Dueck, G.W., Drechsler, R.: RevLib: An online resource for reversible functions and reversible circuits. In: Int'l Symp. on Multi-Valued Logic, pp. 220–225 (2008), RevLib is available at www.revlib.org
27. Yamashita, S., Minato, S., Miller, D.M.: DDMF: An efficient decision diagram structure for design verification of quantum circuits under a practical restriction. IEICE Trans. on Fundamentals E91-A(12), 3793–3802 (2008)
28. Yamashita, S., Minato, S., Miller, D.M.: Synthesis of semi-classical quantum circuits. In: Proc. Workshop on Reversible Computation, pp. 93–99 (2010)

Properties of Quantum Templates

Md. Mazder Rahman and Gerhard W. Dueck

Faculty of Computer Science, University of New Brunswick, Canada

Abstract. Identity circuits are the basis for rewriting rules in the process of optimizing reversible and quantum circuits. Rewriting rules are also known as templates. It has been shown that templates can play an important role in optimizing quantum circuits. This paper presents an in-depth study of the properties of such templates. It is shown that all optimal realizations, within certain limitations, are embedded in templates. The properties presented here, lead to a systematic method of generating all templates with a given number of lines. It is proven that, if the complete set of templates is available, template matching results in optimal circuits.

Keywords: Logic Synthesis, Reversible Logic, Quantum Circuit, Entangled State, Quantum Template.

1 Introduction

Synthesis of quantum logic has gained significance due to the potentials of quantum computation. Logic operations in quantum circuits are inherently reversible and an infinite state space can be found in quantum computation [1]. Direct synthesis of binary reversible logic into quantum circuits is intractable and the non-binary states of information in quantum circuits may result in entangled states. Therefore, researchers are motivated to synthesize reversible logic into classical reversible circuits [2–4] and then transform them into quantum circuits by using decompositions of Multiple-Control-Toffoli (MCT) gates [5] – for example. However, quantum circuits obtained from decomposing MCT circuits are most likely not optimal, even when the MCT circuits are realized with a minimum number of gates. Therefore, the obtained quantum circuits can be optimized by post synthesis methods. In this paper, optimality is determined by the number of gates in a circuit.

The optimization heuristic $Template\ Matching$ was first introduced for MCT circuits in [6] and later adapted to quantum circuits in [7]. Templates are based on circuits that realize the identity function. If a sequence of gates in the circuit to be optimized matches more than half the number of gates in a template, then the matched sequence is replaced with the inverse of the unmatched part of the template. Further, the reconfigured template approach proposed in [8] produces improved results in quantum circuits. The set of quantum templates published in [7] is incomplete and no algorithm for finding templates is given.

In this paper, we present a new definition of templates such that it is possible to obtain optimal circuits if a complete set of templates is given. Several properties of templates are presented. Based on these properties a template generation process is developed.

In the remainder of the paper, the proposed method and the significance of quantum templates are described by the following structure: Section 2 provides the basics of reversible logic, quantum logic, and quantum computations. Section 3 briefly describes the limitations of previous work on quantum templates as well as the motivation for revisiting the template definition. In Section 4, a new definition of quantum templates with some properties is given. The basic idea of generating quantum templates is outlined. This section ends with a description of the impact of reconfigured templates. Section 5 shows that all optimal binary realizations can be obtained from templates. The paper is concludes in Section 6 by giving some directions for future work.

2 Preliminaries

To keep the paper self-contained, the background of reversible circuits and logic operations in quantum circuits are briefly overviewed in this section.

A logic function $f : B^n \to B^n$ is said to be reversible if there is a one-to-one and onto mapping between input and output vectors. A reversible function can be realized by cascading reversible gates such as Multiple Control Toffoli (MCT) [9], Peres [10], and Fredkin [11] gates. The synthesis of reversible logic based on MCT gates is prevalent and the resulting circuits are referred to as MCT circuits. The logic operations within the MCT circuits are performed with binary values. However, the logic operations in quantum computation are quite different from logic operations in classical computation. The fundamental unit of information in quantum computation is the qubit – it can be represented by a state vector. The states $|0\rangle$ or $|1\rangle$ are known as the computational basis states. An arbitrary qubit is described by the following state vector

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad (1)$$

where α and β are complex numbers that satisfy the constraint $|\alpha|^2 + |\beta|^2 = 1$. The measurement of a qubit results in either 0 with probability $|\alpha|^2$, that is, the state $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ or in 1 with probability $|\beta|^2$, that is, the state $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. On the other hand, a classical bit has a state either 0 or 1 which is analogous to the measurement of a qubit state either $|0\rangle$ or $|1\rangle$ respectively. The fundamental difference between bits and qubits is that a bit can be either in state 0 or 1 whereas a qubit can be in a superposition of both states $|0\rangle$ and $|1\rangle$.

Similarly a two qubit system has four basis states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ can be represented by the state vector

$$|\psi\rangle = \lambda_1|00\rangle + \lambda_2|01\rangle + \lambda_3|10\rangle + \lambda_4|11\rangle = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \end{pmatrix} \tag{2}$$

where $\lambda_1\lambda_4 = \lambda_2\lambda_3$. If $\lambda_1\lambda_4 \neq \lambda_2\lambda_3$ then the state $|\psi\rangle$ is referred to as an entangled state that is not separable as the tensor product of two single qubits. In addition to that, since it is possible to form linear combination of states – a so called superposition, a two qubit state can be formed by the tensor product of two single qubit states $\alpha_1|0\rangle + \beta_1|1\rangle$ and $\alpha_2|0\rangle + \beta_2|1\rangle$ represented as

$$\begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} \otimes \begin{pmatrix} \alpha_2 \\ \beta_2 \end{pmatrix} = \begin{pmatrix} \alpha_1\alpha_2 \\ \alpha_1\beta_2 \\ \beta_1\alpha_2 \\ \beta_1\beta_2 \end{pmatrix} \tag{3}$$

where $\alpha_1\alpha_2\beta_1\beta_2 = \alpha_1\beta_2\beta_1\alpha_2$, otherwise the resulting state is entangled.

The NCV gate library includes the elementary quantum gates NOT, $CNOT$, Controlled-V, and Controlled-$V^\dagger$ which have been prevalent for the synthesis of binary reversible functions. The elementary quantum gates, also known as quantum primitives, are represented by their unitary matrices that may include complex elements. The logic operations in quantum gates are performed by matrix vector multiplication that result in output state vectors, where the matrix and the vector represent the quantum gate and the qubit state respectively. The operation of a 2-qubit gate is said to have unit cost. However, when a sequence of quantum gates are acting on the same two qubits, their operations can be considered as unit cost in some technologies [12]. However, we will not use this cost metric in this paper.

The single-qubit NOT and the two-qubits $CNOT$ are self-inverse gates. Controlled-V and Controlled-$V^\dagger$ gates are also known as the Controlled-*sqrt*-of-NOT gates and therefore, Controlled-V and Controlled-$V^\dagger$ are inverse of each other. The Controlled-V and the Controlled-$V^\dagger$ gates can be formed as the cascading of other two two-qubits primitives, as shown in Fig. 1. These are referred to as **splitting rules**. Moreover, the Controlled-V gate and the Controlled-$V^\dagger$ gate can be replaced with each other in Fig. 1 (a) and (b) resulting in two more splitting rules shown in Fig. 1 (c) and (d) respectively. The inverse of the splitting rules are referred to as **merge rules**.

For the synthesis of binary reversible functions using quantum gates, four different qubit states $|0\rangle$, $|1\rangle$, $|v_0\rangle$ and $|v_1\rangle$ are necessary [13] where the state vectors are $|v_0\rangle = \frac{(1+i)}{2}\binom{1}{-i}$ and $|v_1\rangle = \frac{(1+i)}{2}\binom{1}{i}$. If the state of a qubit is $|v_0\rangle$ or $|v_1\rangle$ (also referred to as intermediate signals in the literature) and this is applied to the control of a two-qubit gate, then the resulting output vector is

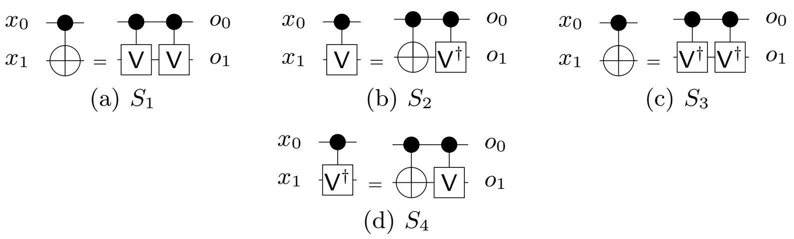


Fig. 1. Splitting and merging of two-qubit quantum primitives

entangled. Therefore, some cascades of quantum gates may produce entangled states. Such states are not permitted here. Note, this restriction only applies to binary reversible circuits — not quantum circuits. Moreover, when a binary two-qubit state either $|10\rangle$ or $|11\rangle$ is applied to a Controlled-V gate where control input state is $|1\rangle$ then the output state either $|v_0\rangle$ or $|v_1\rangle$ is generated in the target qubit, therefore, quantum circuits realizes not only binary reversible functions but non-binary reversible functions as well. The reversible Toffoli-3 gate and its optimal quantum realization are shown in Figure 2(a) and (b) respectively.

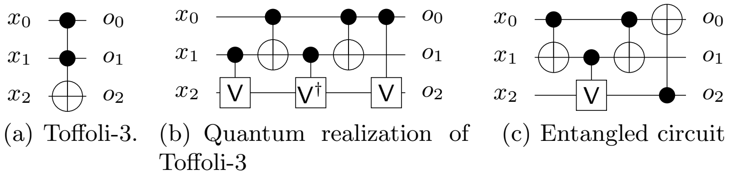


Fig. 2. Cascades of reversible gates

A cascade of quantum primitives may result in an invalid circuit. If a cascades of quantum primitives generates any output state that is not separable as the tensor product of single qubit states then the circuit can no longer be used to realize a binary reversible function. Note that, direct synthesis methods of quantum circuits using quantum primitives must take into account this condition. However, if a quantum circuit is obtained from the quantum decomposition of a MCT circuit, the entangled state does not arise at any intermediate position and the circuit realizes a binary function.

Definition 1. *If a quantum circuit generates an entangled state for any given binary input state is said to be an entangled circuit.*

Example 1. The cascade of quantum primitives shown in Fig. 2(c) is an entangled circuit because it generates an entangled state for input vector $\langle 1, 1, 1\rangle$ and the resulting outputs are not separable into 3 single-qubit state vectors.

3 Previous Work on Quantum Templates

A **quantum template** has been defined as a quantum identity circuit that can not be reduced by other templates [7]. If a sequence of gates in a circuit to be optimized matches a sequence of gates in a template, then it can be replaced with the inverse of the unmatched sequence of the template. This process is known as **template matching**. According to this template definition we have a complete set of 3-qubits templates with up to 8 gates [14], shown in Fig. 3. However, for a given set of all templates for $2l$ qubits, the template matching algorithm does not give optimal results for any arbitrary circuit with l qubits, due to the fact that some identities are not considered as templates, even though they can be used to reduce the number of gates in circuits. This is illustrated with the following example.

Example 2. Consider the circuit shown in Fig. 4(a). This circuit is to be optimized. All sub-circuits of size 3 in the circuit are optimal. The circuit can not be optimized with template matching, even though all templates of 2-qubits are known (see T_3, T_4, T_5, T_7 and T_9 in Fig. 3). However, the optimal realization of the circuit is embedded in the template T_7 in Fig. 4(a) but the sequence of gates in circuit does not match with any 2-qubit template. Therefore, template matching fails to optimize the circuit. Moreover, another post synthesis method – window optimization [15] – does not consider the whole circuit as a sub-circuit to be optimized. Therefore, this method fails to optimize this small circuit because for all windows of size w, where $w < n$ and n is the size of the circuit, all sub-circuits are optimal. However, the 2-qubits quantum identity shown in Fig. 4(b) is not a template since it is reducible by the template T_5 shown in Fig. 4(a), but if it is used in template matching, results is an optimal circuit as shown in Fig. 4(c).

Therefore, some identities that do not comply with the template definition given in [7] have to be considered as templates. This is one motivation for our proposed approach described in subsequent section.

4 Quantum Templates: Definition and Properties

Motivated by the discussion in Section 3, in particular by Example 2, in this section we propose a new template definition and describe the properties of such templates. The major objectives are to 1) construct templates and 2) prove that optimal circuits can be obtained by template matching. A systematic method for the generation of templates, based on the properties, is presented. We also describe reconfigured templates that can be derived from other templates. A template can be reconfigured dynamically during template matching. Details are described in Section 4.2. Note that the concept of reconfigured templates has already been proposed in [8].

Definition 2. *The* **size** *of a circuit C is defined as the number of its gates and denoted by $|C|$.*

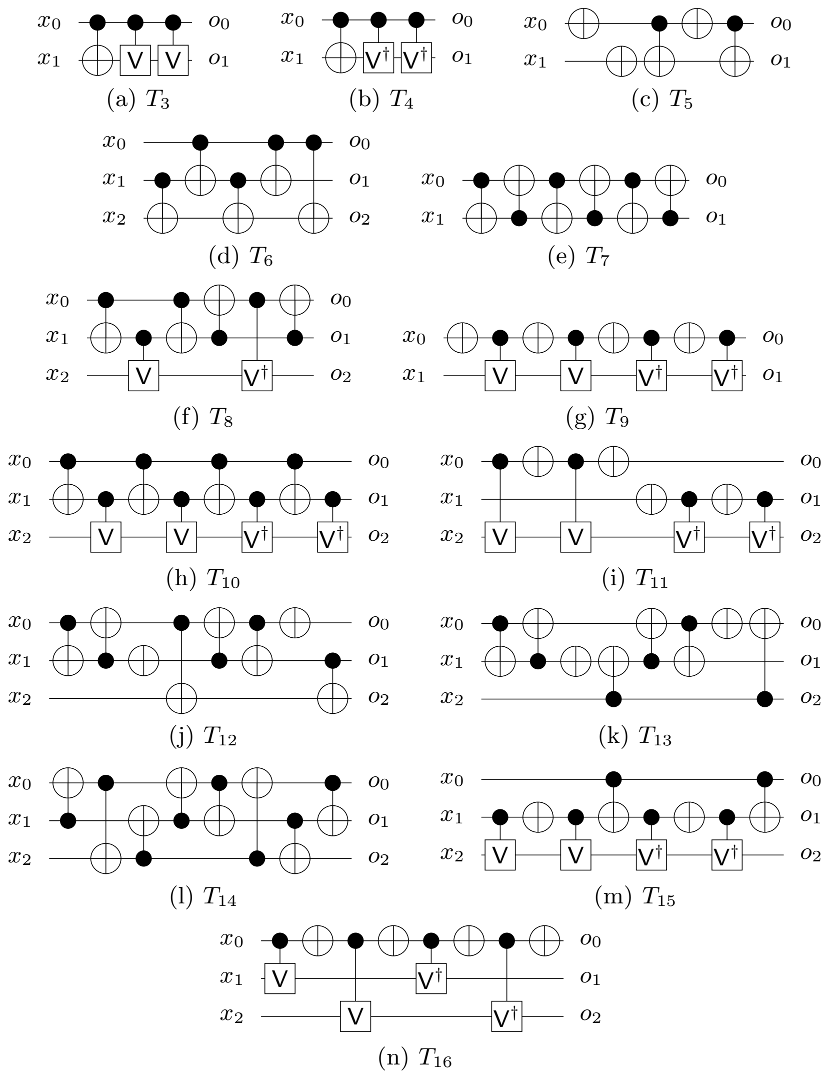


Fig. 3. Quantum templates according to the definition in [7]

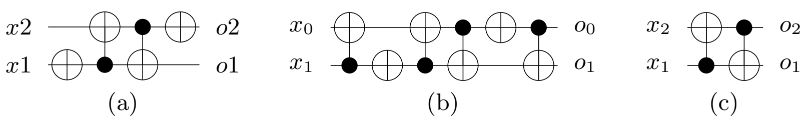


Fig. 4. Quantum circuits of 2-qubit: (a) quantum circuit, (b) Identity circuit and (c) optimal circuit of (a)

Definition 3. *If two circuits C and C' realize the same function then they are said to be* **functional equivalent***. The functional equality of C and C' is denoted by $C = C'$.*

Definition 4. *A circuit C is said to be* **optimal** *if no C' exists such that $C = C'$ and $|C| > |C'|$.*

In light of Example 2, we will use the following definition for templates.

Definition 5. *A* **quantum template** *is an identity circuit with d gates, such that at least one sequence of size $\lfloor \frac{d}{2} \rfloor + 1$ can not be reduced by any other template.*

Theorem 1. *All sub-circuits of an optimal circuit C are optimal.*

Proof. Assume the sub-circuit A of C in not optimal. Since A is not optimal, it is reducible. Therefore, C is reducible – by replacing the sub-circuit A with its optimal – which is a contradiction. □

Theorem 2. *For a given template T there exists at least one sub-circuit of size $\lfloor \frac{|T|}{2} \rfloor$ that is optimal.*

Proof. If no sub-circuits of size $\lfloor \frac{|T|}{2} \rfloor$ is optimal, then all sub-circuits of size $\lfloor \frac{|T|}{2} \rfloor$ in T can be optimized by other templates of size $s < |T|$. Therefore, all sub-circuits of size $\lfloor \frac{|T|}{2} \rfloor + 1$ in T are reducible by other templates, a contradiction. □

Theorem 3. *Given the template $T = S_1S_2$, where $|S_1| = \lfloor \frac{|T|}{2} \rfloor + 1$ and S_1 is not reducible by any other template, then S_1 has a sub-circuit of size $\lfloor \frac{|T|}{2} \rfloor$ that is optimal.*

Proof. If there is a sub-circuit S_1' of size $\lfloor \frac{|T|}{2} \rfloor$ in S_1 and S_1' is not optimal then it can be optimized by other templates. Therefore, S_1 of size $\lfloor \frac{|T|}{2} \rfloor + 1$ is reducible which is a contradiction. □

Theorem 4. *Given the template $T = S_1S_2$, where $|S_1| = \lfloor \frac{|T|}{2} \rfloor + 1$ and S_1 is not reducible by any other template, then S_2 is optimal.*

Proof. Assume that S_2 is not optimal, then $S_2 = S_3$ where $|S_3| < |S_2|$. Therefore, there is an identity realization $I = S_1S_3$ (note $|I| < |T|$) that can reduce the sub-circuit S_1, which is a contradiction. □

Theorem 5. *Given the template $T = S_1S_2$ and $|T|$* ***is even****, where $|S_1| = \lfloor \frac{|T|}{2} \rfloor$ and S_1 is optimal (such an S_1 exists according to Theorem 2), then S_2 is optimal.*

Proof. Suppose S_2 is not optimal, then we have $S_2 = S_3$ where $|S_2| > |S_3|$. Since $S_1 = S_2^{-1}$ we have $S_1 = S_3^{-1}$ (note that $|S_1| > |S_3|$), therefore S_1 is not optimal, which is a contradiction.

□

Theorem 6. *Given the template $T = S_1 S_2$ and $|T|$* ***is odd****, where $|S_1| = \lfloor\frac{|T|}{2}\rfloor + 1$ and $|S_1|$ is not reducible by any other template, then S_2 is optimal.*

Proof. *The proof follows from Theorem 4.* □

Theorem 7. *Let d be the size of the largest optimal circuit with l qubits, then the largest template with l qubits has size no larger than $2d + 1$.*

Proof. *Assume there is a template T such that $|T| > 2d + 1$. According to Theorem 4 this template must have an optimal sub-circuit S_1 such that $|S_1| \geq d + 1$, a contradiction.* □

Definition 6. *A circuit is said to be k optimal if all sub-circuits of k gates are optimal.*

Theorem 8. *Given a k optimal circuit C with l qubits, applying all templates with l qubits of size up to $2(k + 1) + 1$ will result in a $k + 1$ optimal circuit.*

Proof. *Assume a sub-circuit S_1 in C, where $|S| = k + 1$ is not optimal. Let $S_1 = S_2$ where $|S_1| > |S_2|$. This implies an identity realization $I = S_1 S_2^{-1}$. I must be a template (note $|I| < 2(k+1)$), since S_1 is not reducible by any template of size up to $2(k + 1) + 1$, a contradiction.*

□

Theorem 9. *If all templates with l qubits of size up to d are known, then $\lfloor\frac{d-1}{2}\rfloor$ optimality can be achieved for any circuit C with up to l qubits.*

Proof. *By applying the templates of size up to $5, 7, 9, \ldots, d$, $\lfloor\frac{d-1}{2}\rfloor$ optimality is achieved according to Theorem 8.* □

Theorem 10. *If all templates of l qubits are known, then an optimal realization for any circuit C of l qubits can be achieved.*

Proof. *Let d be the size of the largest template. By Theorem 9 we can obtain C' from C such that C' is $\lfloor\frac{d}{2}\rfloor$ optimal. If $|C'| \leq \lfloor\frac{d}{2}\rfloor$ then it is optimal. Assume $|C'| \geq \lfloor\frac{d}{2}\rfloor$. Let S be a sub-circuit of C' such that $|S| = \lfloor\frac{d}{2}\rfloor + 1$. Let $S = S_1$ where S_1 is optimal. Then $I = S_1 S^{-1}$ must be template, a contradiction.* □

4.1 Generation of Quantum Templates from Identities

A template is an identity circuit, therefore, it is clear that $S_T \subseteq S_I$ where S_T the set of templates of l qubits and S_I is the set of identity realization of l-qubits. However, according to the definition of templates it has to be ensured that all templates of size up to $s \leq n$ are already generated while we are looking for another template of size $d \geq n$. Therefore, the generation of new templates is an iterative process.

All quantum primitives are considered to be optimal. Therefore, the generation of identity realizations from optimal realizations can be considered by the following two conditions:

1. If C and C' are optimal quantum circuits where $|C| = |C'|$ and $C = C'$ but they have different arrangement of gates, then we have an identity realization $I = CC'^{-1}$ of even size that satisfies Theorem 5.
2. Suppose C and C' are optimal quantum circuits where $|C| = |C'|$ and $C \neq C'$. If $C_2 = Cg$ where g is a gate such that $C_2 = C'$ and $|C_2| = |C| + 1$ then we have an identity realization $I = C_2C'^{-1}$ of odd size that satisfies Theorem 6.

Since the optimal quantum realization for a given reversible function is not unique, the general idea of finding all identities and templates is described in the following. First find all optimal realizations of the function f as proposed in [16], let this be the set $S_f = \{C_1, C_2, \ldots, C_m\}$ and $|C_i| = n$. Circuits from the sets S_f and $S_{f^{-1}}$ are concatenated to form identities. Note that circuits in $S_{f^{-1}}$ are obtained from the S_f by reversing the order of the gates in each circuit. Template matching is used to determine if a given identity is a template. This will only yield templates with an even number of gates. To find templates with an odd number of gates f^{-1} must be realized with $n + 1$ gates.

The basic quantum identity circuits are comprised of two gates whose controls and targets are acting on the same qubit as shown in Fig. 5 – they can easily be detected. If they can be moved in such a way that they are adjacent, they can be deleted. At the beginning, the basic quantum identities are considered as the initial set of templates. The same data structure used for generating identities based on all 3-qubits optimal circuits proposed in [14] with the new definition of template has been used to find new templates. According to the new definition of template, the complete set of all 17 2-qubit quantum templates have been found.

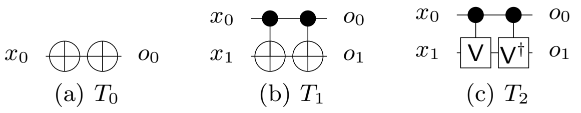


Fig. 5. Basic quantum identity circuits used as initial templates

Example 3. The circuits C and C' shown in Fig. 6(a) and (b) realize the same function $f : \{00, 01, 10, 11\} \rightarrow \{10, 00, 11, 01\}$. According to the condition 1, we have an identity realization CC'^{-1} as shown in Fig. 6(c). The sub-circuit S of gate sequence $\{1, 2, 3, 4, 5\}$(count 0 from left) in Fig. 6(c) where $|S| = \lfloor \frac{|CC'^{-1}|}{2} \rfloor + 1$ cannot be reduced by other templates, therefore, this identity is a template with even size.

Example 4. The circuits C and C' shown in Fig. 7(a) and (b) are both optimal but they realize two different reversible functions. However, the concatenation C in Fig. 7(a) with a CNOT gate results in circuit $C_2 = Cg$ shown in Fig. 7(c) of size 4. The circuits C_2 and C' are functional equivalent, that is, $C_2 = C'$. It is clear that $|C'| \neq |C_2|$ and C_2 is not optimal. Since $C_2 = C'$, we have the identity $C_2C'^{-1}$ as shown in Fig. 7(d). According to the definition of template,

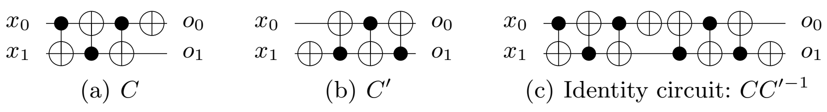


Fig. 6. Quantum circuits

the sub-circuit of gate sequence $\{3, 4, 5, 6\}$ (start from 0) in Fig. 7(d) can not be optimized by other templates. Therefore, it is a new template. However, this is a reconfigured template [8] since by applying gate merge rules, it can be transformed to the circuit shown in Fig. 7(e) which is a template.

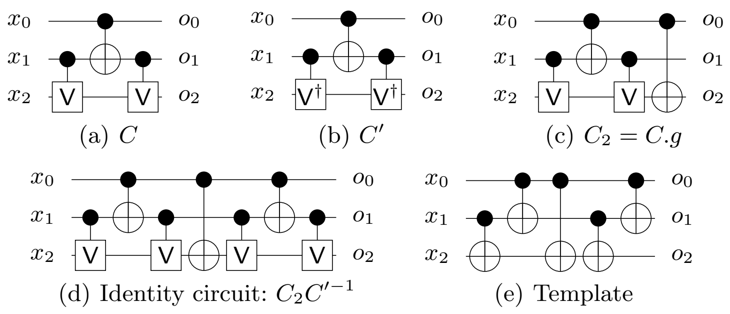


Fig. 7. Quantum circuits

4.2 Impact of Reconfigured Templates

A quantum template is said to be a reconfigured template [8], if it can be derived from another template by using splitting rules shown in Fig. 1. For example, the template in Fig. 8(a) can be reconfigured as Fig. 8(b) and (c).

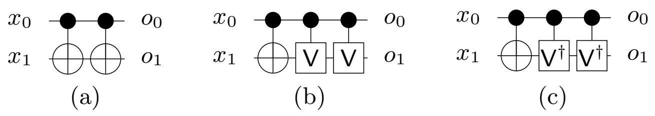


Fig. 8. Template (a) is reconfigured as (b) and (c)

We predict that the number of templates will grow exponentially with respect to the number of qubits. Hence, the number of templates in the set of all templates would be too large to be considered in a template matching procedure. However, reconfigured templates can easily be detected. That is, if any of the quantum gate merge rules as shown in Fig. 1 can be applied, the template falls into the reconfigured category. The deletion of reconfigured templates from the set all templates results a reduced set of templates that is sufficient. Fig. 9 shows all 2-qubit templates except reconfigured templates. The templates T_2, T_4 and T_5 in Fig. 9 can be reconfigured in 2, 4, and 2 ways respectively. How templates can be reconfigured dynamically, is described in [8].

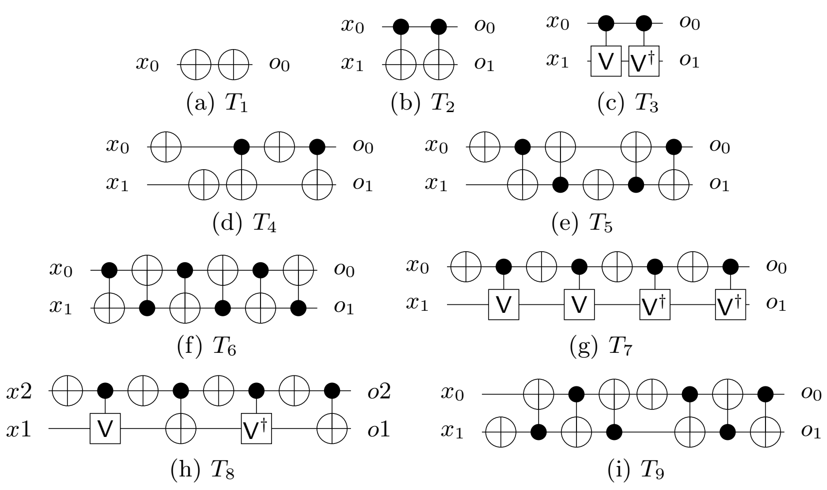


Fig. 9. 2-qubit templates

5 Completeness of Optimality of Binary Realizations from Templates

From the properties of quantum templates it can be seen that all optimal realizations are embedded in the templates. Therefore, given the set of all templates for n qubits, an optimal realization of any reversible function f with n qubits can be found. Reconfigured templates are derived from other templates. Therefore, if an optimal realization of a binary reversible function f is embedded into a reconfigured template then there exists another template in which the optimal realization of that binary reversible function f is also embedded. Table 1 shows all $2^2! = 24$ 2-qubit binary functions, their realizations, and the corresponding templates (as shown in Fig. 9) in which the functions are embedded.

Table 1. All optimal binary realizations of 2-qubits

Function	Realization	Template	Function	Realization	Template
0,1,2,3			0,3,1,2	$t_2(a,b)t_2(b,a)$	T_5, T_6, T_9
2,3,0,1	$t_1(a)$	T_1, T_5, T_7, T_8, T_9	0,2,3,1	$t_2(b,a)t_2(a,b)$	T_5, T_6, T_9
1,0,3,2	$t_1(b)$	T_1, T_5, T_7, T_8, T_9	1,2,0,3	$t_1(a)t_2(a,b)t_2(b,a)$	T_5, T_9
3,2,1,0	$t_1(a)t_1(b)$	T_4	3,0,2,1	$t_1(b)t_2(a,b)t_2(b,a)$	T_9
0,1,3,2	$t_2(a,b)$	T_2, T_4, T_5, T_6, T_8, T_9	0,2,1,3	$t_2(a,b)t_2(b,a)t_2(a,b)$	T_6, T_9
0,3,2,1	$t_2(b,a)$	T_2, T_4, T_5, T_6, T_8, T_9	1,3,0,2	$t_1(a)t_2(a,b)t_2(b,a)t_2(a,b)$	T_9
3,2,0,1	$t_1(a)t_2(a,b)$	T_4, T_5, T_8, T_9	2,0,3,1	$t_1(b)t_2(a,b)t_2(b,a)t_2(a,b)$	T_9
2,3,1,0	$t_2(a,b)t_1(a)$	T_4, T_5, T_8, T_9	3,1,2,0	$t_2(a,b)t_2(b,a)t_1(a)t_2(a,b)$	T_9
3,0,1,2	$t_1(b)t_2(b,a)$	T_4, T_5, T_8, T_9	3,1,0,2	$t_1(a)(b,a)t_2(a,b)$	T_9
1,2,3,0	$t_2(b,a)t_1(b)$	T_4, T_5, T_8, T_9	2,1,3,0	$t_2(a,b)t_2(b,a)t_1(b)$	T_9
1,0,2,3	$t_2(a,b)t_1(b)$	T_4, T_9	1,3,2,0	$t_2(b,a)t_2(a,b)t_1(b)$	T_9
2,1,0,3	$t_1(a)t_2(b,a)$	T_4, T_9	2,0,1,3	$t_1(a)t_2(b,a)t_2(a,b)$	T_5, T_9

Function $\{3,2,0,1\}$ stands for $f : \{00,01,10,11\} \rightarrow \{11,10,00,01\}$

6 Conclusion and Future Work

An new definition of quantum template has been proposed. It has been shown that a complete set of templates has some interesting properties. These properties can be used to generate a complete set for templates for a given number of qubits. One conclusion that can be drawn from the properties, is that the set of templates is very large, since all optimal circuits must be embedded in them. We are currently in the process of generating all 3-qubit templates. Even this task requires significant processing power and smart algorithms. We hope to complete this task soon.

It is clear that the number of templates, even for 3 qubits, will be very large. In fact, it is not expected that all templates could be efficiently considered in a template matching procedure. However, empirical results show that some templates are more often applied than others. This is particularly evident if the circuits were obtained using Barenco's decomposition [5]. The obvious next step (after the complete set of 3-qubit templates has been obtained) is to see which templates are applied most frequently and how good the optimization results are, if a reduced set of templates is applied. The goal is to obtain near-minimal results using reasonable amount of computing time with a limited set of templates.

References

1. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2000)
2. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: Design Automation Conference (2003)
3. Iwama, K., Kambayashi, Y., Yamashita, S.: Transformation rules for designing CNOT-based quantum circuits. In: Design Automation Conference, New Orleans, Louisiana, USA (2002)
4. Mishchenko, A., Perkowski, M.: Logic synthesis of reversible wave cascades. In: International Workshop on Logic Synthesis (2002)
5. Barenco, A., Bennett, C.H., Cleve, R., DiVinchenzo, D., Margolus, N., Shor, P., Sleator, T., Smolin, J., Weinfurter, H.: Elementary gates for quantum computation. The American Physical Society 52, 3457–3467 (1995)
6. Maslov, D., Dueck, G.W., Miller, D.M.: Toffoli network synthesis with templates. Transactions on Computer Aided Design 24, 807–817 (2005)
7. Maslov, D., Young, C., Dueck, G.W., Miller, D.M.: Quantum circuit simplification using templates. In: DATE - Design, Automation and Test in Europe, pp. 1208–1213 (2005)
8. Rahman, M.M., Dueck, G.W., Banerjee, A.: Optimization of Reversible Circuits Using Reconfigured Templates. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 43–53. Springer, Heidelberg (2012)
9. Toffoli, T.: Reversible computing. Tech. memo MIT/LCS/TM-151, MIT Lab. for Comp. Sci. (1980)
10. Peres, A.: Reversible logic and quantum computers. Phys. Rev. A 32, 3266–3276 (1985)
11. Fredkin, E., Toffoli, T.: Conservative logic. International Journal of Theoretical Physics 21, 219–253 (1982)

12. Rahman, M.M., Banerjee, A., Dueck, G.W., Pathak, A.: Two-qubit quantum gates to reduce the quantum cost of reversible circuit. In: Proceedings of the International Symposium on Multiple-Valued Logic, pp. 86–92 (2011)
13. Hung, W., Song, X., Yang, G., Yang, J., Perkowski, M.: Optimal synthesis of multiple output Boolean functions using a set of quantum gates by symbolic reachability analysis. Transactions on Computer Aided Design 25, 1652–1663 (2006)
14. Rahman, M.M., Dueck, G.W.: An algorithm to find quantum templates. In: IEEE Congress on Evolutionary Computation (accepted 2012)
15. Soeken, M., Wille, R., Dueck, G.W., Drechsler, R.: Window optimization of reversible and quantum circuits. In: International Symposium on Design and Diagnostics of Electronic Circuits and Systems (2010)
16. Rahman, M.M., Dueck, G.W.: Optimal quantum circuits of 3-qubits. In: Proceedings of the International Symposium on Multiple-Valued Logic (accepted 2012)

Optimal 4-bit Reversible Mixed-Polarity Toffoli Circuits

Marek Szyprowski[1] and Paweł Kerntopf[1,2]

[1] Institute of Computer Science, Department of Electronics and Information Technology Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland
[2] Department of Theoretical Physics and Informatics, University of Łódź, Pomorska 149/153, 90-236 Łódź, Poland
`{m.szyprowski,p.kerntopf}@ii.pw.edu.pl`

Abstract. Optimal synthesis of reversible circuits is a very hard task. For example, up to year 2009 this problem had not been solved even for 4-bit reversible functions, in spite of intensive research during previous decade. In 2010, a method and a tool of practical usage for finding optimal circuits for any 4-bit reversible specification were finally developed. Namely, with sophisticated optimizations it was possible to find gate count optimal circuits for any 4-bit reversible function built from multi-control Toffoli gates. Last year, we published an extension to the algorithm, which allows to reduce the quantum cost of the resulting circuits. In this paper we present another extension to this approach. Namely, we have extended the reversible gate library to mixed-polarity multi-control Toffoli gates (i.e. with both positive and negative controls). Our experimental results for the known reversible benchmarks show that using mixed-polarity Toffoli gates gives significant savings in gate count. The paper presents results of different computational experiments including optimal 4-bit circuits for the known reversible benchmarks with respect to both gate count and quantum cost criteria.

Keywords: reversible circuits, synthesis, quantum cost.

1 Introduction

A gate (circuit) is called reversible if there is a one-to-one correspondence between its inputs and outputs. Research on reversible logic circuits is motivated by possible applications in quantum computing, low-power design, nanotechnology, optical computing, bioinformatics and cryptography. Therefore, synthesis of reversible logic has been intensively studied for the last decade [15]. The attention has been focused on the synthesis of circuits built from the NCT library of gates consisting of NOT, CNOT and Toffoli gates. Many reversible circuit synthesis algorithms for this library have been proposed. However, they usually generate suboptimal circuits.

For many years few exact optimal circuits have been found for n-variable functions with $n > 3$. For example, it was even not known what is the maximal gate count in optimal circuits implementing 4-bit reversible functions. This problem is very hard because there are $16! \approx 2\cdot10^{13}$ such functions so the search space is extremely large.

In 2010 a very fast tool capable of synthesizing optimal circuits for any 4-bit reversible specification was finally developed [5]. With this tool it was possible to establish that there are 144 4-bit functions requiring 15 gates in their optimal circuits and that there exists none requiring 16 or more gates [5], [9].

Quality of a reversible circuit is usually estimated by gate count (GC) or by a metric called quantum cost (QC). Much less effort has been devoted to minimization of QC in reversible circuits. Usually, minimization of GC was the first step and then an effort to reduce quantum cost was made with the fixed gate count [6].

However, as we showed in [20], finding exact minimal quantum cost circuits requires considering circuits having greater number of gates than the minimal size ones. Synthesis of reversible circuits with gates having mixed-polarity control (i.e. with both positive and negative controls) has been considered in a number of papers [1], [2], [7], [12-14], [16-17], [24-25]. However, exact optimal circuits have been reported only for circuits with up to 7 gates [23].

Last year we developed a tool similar to the one reported earlier in [5] capable of finding a circuit having a minimal number of gates for any 4-bit reversible function. We also programmed and run a method capable of finding *all* circuits implementing a given reversible function *with a specified value of GC*. Now we have extended this tool to synthesis of circuits built with gates having mixed-polarity control. Our experimental results for the known reversible benchmarks show that using mixed-polarity Toffoli gates gives significant savings in gate count. The paper presents results of different computational experiments including optimal 4-bit circuits for the known reversible benchmarks with respect to both gate count and quantum cost criteria. These results have implications in testing synthesis algorithms for reversible mixed-polarity circuits and quantum circuits.

The paper is organized as follows. Section 2 recalls basic concepts of reversible logic. In Section 3 notions of cost functions and optimal reversible circuits are introduced. Section 4 describes a tool [4], [5] for synthesis of 4-bit gate count optimal reversible Boolean circuits. In Section 5 our extensions to this tool are briefly presented. In Section 6 our experimental results are collected and compared to known circuits from benchmark webpages and from the literature. Section 7 summarizes the paper with conclusions and suggestions for further research.

2 Preliminaries

Definition 1. A completely specified n-input n-output Boolean function (referred to as $n*n$ function) is called reversible if it maps each input assignment into a unique output assignment.

There are $2^n!$ reversible $n*n$ Boolean functions. For $n = 3$ this number is equal to 40,320 and for $n = 4$ is greater than $2\cdot 10^{13}$.

Definition 2. An n-input n-output ($n*n$) gate (or circuit) is *reversible* if it realizes an $n*n$ reversible function.

In a reversible circuit fanout of each gate output is always equal to 1. As a consequence $n*n$ reversible circuits can be only built as a cascade of $k*k$ reversible gates ($k \le n$).

Definition 3. A set of reversible gates that can be used to build reversible circuits is called a *gate library*.

Many gate libraries have been examined in the literature. The so called *NCT library* for $n \le 4$ consists of 1*1 NOT, 2*2 CNOT and 3*3 and 4*4 TOFFOLI gates. Below definitions of generalized mixed polarity gates of those types are given.

Definition 4. Let $a_i \in \{0, 1\}$ for $i = 1, 2, 3, 4$, and let $\oplus$ denote XOR operation.

1*1 *NOT*(x_1) *gate* performs the operation

$$(x_1) \rightarrow (x_1 \oplus 1),$$

2*2 ***CNOT***(x_1, x_2) *gate* performs the operation

$$(x_1, x_2) \rightarrow (x_1, (x_1 \oplus a_1) \oplus x_2),$$

3*3 ***TOFFOLI***(x_1, x_2, x_3) *gate* performs the operation

$$(x_1, x_2, x_3) \rightarrow (x_1, x_2, (x_1 \oplus a_1)(x_2 \oplus a_2) \oplus x_3),$$

4*4 ***TOFFOLI4***(x_1, x_2, x_3, x_4) *gate* performs the operation

$$(x_1, x_2, x_3, x_4) \rightarrow (x_1, x_2, x_3, (x_1 \oplus a_1)(x_2 \oplus a_2)(x_3 \oplus a_3) \oplus x_4).$$

The above defined $i*i$ gates, where $i = 1, 2, 3, 4$, (in short, denoted by N, C, T, T4, respectively) invert input x_i if and only if the values of inputs $x_1, x_2, ..., x_{i-1}$ differ from corresponding $a_1, a_2, ..., a_{i-1}$ coefficients, passing these inputs unchanged to corresponding outputs. Signals which are passed unchanged from input to output of the gate are called *control lines*. The signal x_i which can be modified by the gate is called *target*. Each of the N, C, T, T4 gates is invertible, i.e. equal to its own inverse.

Some commonly used names for describing control lines have been introduced. If all $a_1, a_2, ..., a_i$ coefficients equals zero, the gate has *positive-polarity* control lines. Alternatively, if all $a_1, a_2, ..., a_i$ coefficients equals one, the gate has *negative-polarity* control lines. These names come from expanding the expressions for the functions realized by the gate. *Positive-polarity* means that all variables which correspond to control lines directly affects the target line. *Negative-polarity* means that the target line is affected only if the values of control lines are equal 0. The term *mixed-polarity* control lines is used if all values of $a_1, a_2, ..., a_i$ coefficients are allowed to be 0 or 1.

In the literature the term NCT gate library is commonly used as a synonym for the library consisting of N, C, T gates with positive-polarity controls exclusively. We will use the term *mixed-NCT* (*m-NCT* in short) for the generalized NCT library with all gates having mixed-polarity control lines.

3 Cost Functions and Optimal Reversible Circuits

It can easily be noticed that for most of reversible functions there exist more that one reversible circuit implementing it. Thus a cost function has to be defined to evaluate the quality of a circuit. For this purpose additive cost functions are applied. The simplest cost function of a reversible circuit is based on the total number of gates. It is called *gate count* (GC in short).

Other cost functions are also considered. The most widely used, called quantum cost (QC), is based on the cost of elementary quantum gates required to build the circuit using the best known quantum mapping procedure of the reversible gates. It is assumed that the cost of each elementary quantum gate equals 1, so the cost of a reversible gate equals to the total number of elementary quantum gates used. The quantum cost of positive-polarity N, C, T and T4 gates is assumed to be 1, 1, 5 and 13, respectively (see for example [3], [10]). These values have been calculated in assumption that the four types of elementary quantum gates are available: N, C, V/V+ (the latter two implementing square roots of NOT gate) and W/W+ (implementing fourth roots of NOT gate). Quantum cost of mixed-polarity NCT gates is generally assumed the same as positive-polarity with the exception that the quantum cost of all negative-polarity C, T and T4 gates is higher by 1 and equals 2, 6 and 14, respectively [8], [11], [14], [18].

By an *optimal circuit* we mean an implementation having the minimal cost. The set of optimal circuits implementing a reversible function depends on a gate library and a cost function. One can easily notice that a cascade built from arbitrary types of gates might realize more than one reversible function, because the final function is defined by the order of variables and the possible inversion of the circuit [4-5].

Let us define an equivalence class in the set of reversible functions with respect to the types of gates in optimal circuits implementing those functions.

Definition 5. Two reversible Boolean functions are called *equivalent* if they belong to the same equivalence class (called *a conjugacy class* in [4-5]) under simultaneous input/output relabeling and reversible function inversion.

It can be easily shown that two functions which belong to the same equivalent class have the same cost. Such equivalence class can contain maximally $2 \cdot n!$ functions (number of permutations of all variables doubled by the possibility of inversion). In a set of 4-bit reversible functions one equivalence class contains up to 48 functions.

We will be using the following popular vector notation for an n-variable reversible Boolean function f: $[f(0), f(1), \ldots, f(2^n\text{-}1)]$, where each binary vector $f(i)$ will be expressed as a decimal.

Definition 6. The *canonical representative of an equivalence class* is the function whose vector $[f(0), f(1), \ldots, f(2^n\text{-}1)]$ is lexicographically smallest.

4 Gate Count Optimal Synthesis

In 2010 Golubitsky, Falconer and Maslov [4] presented the implementation of an algorithm for finding a gate count optimal reversible circuit for any 4*4 reversible function. The algorithm is based on the fact that the set of all functions that have a gate count optimal circuit up to 9 gates can be effectively stored in memory of nowadays computers. It also relied on the fact that for each equivalent class of reversible functions it is sufficient to store only its canonical representative, so the amount of required memory can be reduced almost 48 times. Reversible functions in such database are stored in hash tables. Authors of [4] constructed a database with canonical representatives of optimal circuits up to 9 gates. With such database, the GC of the optimal circuit can be quickly checked for any reversible function *f*, requiring up to 9 gates by a simple lookup of a canonical representative. To find GC for the functions that require more than 9 gates in an optimal circuit additional processing is needed. Namely, the algorithm relies on the fact that any optimal circuit for the function *f* can be partitioned into two circuits which realize functions *g* and *r*, such that $f = g \circ r$, where symbol $\circ$ denotes composition (cascading of the circuits). The above equation can be transformed into the following: $f \circ r^{-1} = g$, where r^{-1} denotes the inverse of the function *r*.

The algorithm (named *FINDOPT* in [4]) iterates over all functions in the database that have optimal circuits of length *i*, for $i < 9$, computes their inversions and then compose the function *f* with all of them. For the resulting function *g* it checks the length of an optimal circuit by a lookup in the database. If such function *g* with optimal circuit of length *j* has been found in the database, the length of the optimal circuit for the function *f* equals to $i+j$ [4].

The above procedure provides an effective and very fast method of finding the length of a gate count optimal circuit for any 4*4 reversible function. It can also be easily extended to a fast algorithm for finding a gate count optimal circuit [4]. One just needs to store the last gate of the optimal circuit with each function in the database and the complete circuit can be easily constructed with a recursive approach.

Using the computer system with 16 AMD Opteron 2300 MHz processors and 64GB RAM the authors of [4] managed to create a database for 9-gate optimal circuits and synthesize an optimal circuit for any given 4*4 reversible function in about 0.01s on average. However, they considered positive-polarity NCT gates exclusively.

5 Our Extension to the Synthesis Algorithm

In [20-21] we have extended the algorithm presented in [4-5] to an algorithm for finding *all* reversible circuits with a specified gate count for the given reversible function. Our approach is a combination of the original algorithm and depth-first search with database for effective pruning the search tree. Such approach is similar to a generic scheme of heuristic search based algorithms for reversible circuit synthesis described earlier in the literature [19]. In each step a reversible gate is selected one by one. Each such gate is added at the end of the previously analyzed gate cascade and the result is checked if it gives a circuit for the specified reversible function with the selected number of gates. This check is performed by calculating the reversible function for the *to-be-constructed* part of the circuit and calculating the gate count of an optimal circuit for it. Once the final circuit has been constructed, the algorithm backtracks and tries other gates until *all* circuits for the specified reversible function will be found. If adding the selected gate to the previously analyzed gate cascade results in a longer *to-be-constructed* part of the circuit than in the previous step of the algorithm, the search path is abandoned and the algorithm returns to the previous level of the depth-first search procedure. Our approach results in developing an algorithm which is well balanced in both CPU power and memory usage complexity and allows to use the resources of nowadays computer systems in the most efficient way.

Once all reversible circuits for an arbitrary circuit length have been constructed, the second phase begins. Namely, for each of those circuits the quantum cost function is calculated and a circuit with the lowest quantum cost is considered as the final result. The proposed approach allowed us to find 4-bit circuits for the known reversible benchmarks with significantly lower quantum cost (on average 44.7%) than the best circuits published earlier in the literature [20-21].

6 Experimental Results

In our experiments we have focused on mixed-polarity NCT gate library and for such gate library we have constructed a database of 4-bit optimal circuits. Our experiments were performed on the IBM xSeries x3650 with 2x6-core Intel Xeon 6C X5650 2.66GHz CPUs and 92 GiB of RAM computer system and our tools were run on KVM virtualized RedHat RHEL 6 64bit operating system with 8 logical CPUs and 72GiB of RAM. With such computer system we managed to construct a database of gate count optimal mixed-polarity reversible circuits up to 7 gates and quantum cost optimal circuits up to 23 units. The parameters for the gate count and quantum cost optimal circuit databases for the mixed-polarity NCT gates are presented in Table 1 and Table 5, respectively. The first column shows the number of gates in an optimal circuit, the second column shows the total number of functions requiring such number of gates in an optimal circuit and the third one presents the number of canonical representatives of the reversible functions actually stored in the database. The next group of columns presents the parameters of the hash table used to store canonical representatives for circuits of the respective gate count: 'Max Entries' means the total number of entries that can be stored in the hash table of that size, where 'Size' means the amount of memory occupied by the hash table in the memory of the computer system, 'Max Chain Length' shows the length of the chained (conflicting) entries in the hash table and the next column shows the load factor of the hash table. Relatively low values of the maximal chain length compared to the size of the table and high values of the fill ratio show that the selected hash calculation algorithm is correctly selected for this task and the hash tables can be used for very fast lookup operations.

Table 1. Parameters of the gate count optimal circuit database for reversible functions requiring up to seven mixed-polarity NCT gates

GC	# Functions	# Canonical Representatives of Functions	Hash Table Parameters			
			Max Entries	Size	Max Chain Length	Load Factor
1	108	10	16	128B	1	62.5%
2	6,774	244	256	2KiB	51	95.3%
3	313,140	7,292	8,192	64KiB	615	89.0%
4	11,559,793	245,457	262,144	2MiB	1,202	93.6%
5	349,572,560	7,310,186	8,388,608	64MiB	823	87.1%
6	8,585,260,568	179,011,749	268,435,456	2GiB	148	66.7%
7	163,493,840,712	3,406,842,995	4,294,967,296	32GiB	483	79.3%

Using our database with gate count optimal circuits up to 7 gates we are able to construct optimal circuits from m-NCT gates up to 14 gates. From [5] we know that optimal circuits built from positive-polarity NCT gates require at most 15 gates. NCT gate library is a subset of m-NCT gate library, so gate count m-NCT optimal circuits cannot be larger than NCT based ones. We have constructed the m-NCT optimal circuits for all reversible functions requiring exactly 15 NCT gates in the optimal circuit. The results are provided in Table 6. Optimal m-NCT circuits for these functions consist of 10 or 11 gates, what proves that our database contains enough data to construct an m-NCT optimal circuit for any 4-bit reversible function.

Table 2. Distribution of the number of gates in the optimal circuit for 10,000,000 randomly generated reversible functions

GC	# Functions
4	9
5	164
6	4070
7	77748
8	996225
9	5354699
10	3554656
11	12429

Our next experiment consists in finding m-NCT optimal circuit size for 10,000,000 random reversible functions. Results are available in Table 2. Basing on the distribution of the number of gates for the randomly generated functions with uniform distribution we have extrapolated the distribution of gate count m-NCT optimal circuits for all 4-bit reversible functions – see Table 3. For comparison we also provide a distribution of NCT optimal circuits calculated in [4-5]. It can be noticed that optimal circuits built from m-NCT gates are significantly smaller than those built from NCT gates.

Table 3. Number of 4-bit permutations requiring prescribed number of gates for gate count optimal circuits built from NCT and mixed-polarity NCT gate libraries

GC	NCT [5]	m-NCT
0	1	1
1	32	108
2	784	6,774
3	16,204	313,140
4	294,507	11,559,793
5	4,807,552	349,572,560
6	70,763,560	8,585,260,568
7	932,651,938	163,493,840,712
8	10,804,681,959	2.11×10^{12}♣
9	105,984,823,653	1.14×10^{13}♣
10	819,182,578,179	7.54×10^{12}♣
11	4,298,462,792,398	2.64×10^{10}♣
12	10,690,104,057,901	?♣
13	4,959,760,623,552	?♣
14	37,481,795,636	?
15	144	0

♣ - estimation based on the distribution on 10,000,000 random reversible function

This observation has been verified with the next experiments. The first one includes finding optimal circuits for the important class of 4-bit affine functions [4-5]. Results of this experiment are presented in Table 4. For this set of reversible functions the average gate count of the m-NCT gate count optimal circuits is approximately one gate lower than the average gate count of the NCT optimal circuits.

Table 4. Distribution of the number of gates in the gate count optimal circuit for 4-bit affine reversible functions

GC	NCT [5]	m-NCT
0	1	1
1	16	28
2	162	438
3	1206	4340
4	6589	25761
5	26182	82680
6	72062	129016
7	118424	71096
8	84225	9104
9	13555	96
10	138	
Average:	6,88	5,82

The last set of experiments have been performed on 4-bit reversible benchmarks. We applied our extension to the synthesis algorithm and constructed all reversible circuits for the specified functions. For all such circuits we have calculated their quantum cost. We also constructed reversible circuits for the benchmark functions using the database of quantum cost optimal circuits. Results are presented in Table 6. For the obtained set of circuits we selected those having lowest quantum cost in case of gate count synthesis and lowest gate count in case of quantum cost synthesis. The table is organized in the following way. The rows which span across the whole table contain the name of the benchmark and its specification as a list of output vectors (in decimal numbers). The next rows (separated into columns) contain the experimental data. The first column shows the number of gates of the obtained reversible circuits for the specified reversible function. The value in the first row after the reversible benchmark name contains data for gate count optimal circuits (there exist no circuits with less gates). The second column shows the range of quantum cost of all generated gate count optimal reversible circuits or quantum cost value (show in bold) for circuits constructed from database of the quantum cost optimal circuits. The third column shows the total number of the generated circuits. The fourth column contains the information about the time (measured in CPU-seconds) required to generate all the circuits. The last column contains an example of a circuit with the lowest quantum cost which have been found for the specified number of gates.

To present circuits in a compact way we have shortened the names of gates. Notation for the positive-polarity controlled gates is as follows:

- NOT(a) is denoted by Na,
- CNOT(a, b) is denoted by Ca-b,

– TOFFOLI(a, b, c) is denoted by Tab-c,
– TOFFOLI(a, b, c, d) is denoted by Tabc-d.

Negative-polarity controls are marked with prime symbol. For example, negative-polarity CNOT(a, b) gate implementing the function: (a, b) $\rightarrow$ (a, ($a \oplus 1$) $\oplus$ b) is denoted by Ca'-b.

Summary of this experiment and a comparison with earlier results for 4-bit optimal circuit synthesis constructed for NCT library are shown in Table 7. The first column contains names of reversible benchmarks. Column II contains GC and QC parameters of the best circuits under gate count cost taken from the reversible benchmark collections [9], [22] and the literature. Column III contains GC and QC parameters for the circuits which had the lowest quantum cost constructed in [20]. Similarly, column IV contains GC and QC parameters for the gate count optimal circuits constructed in [20]. All these results were achieved for positive-polarity NCT gates. The next column (V) contains GC and QC parameters for the gate count optimal circuits constructed using our database for m-NCT gates. It can be noticed that optimal gate count circuits constructed from m-NCT gates are smaller (on average by 25.7%) than optimal circuits built from NCT gates. The last column (VI) contains GC and QC for the circuits of lowest quantum cost constructed from the quantum cost optimal database for m-NCT gates. Although the circuits presented in this column have not been proven optimal, one can notice significant (about 5 units) quantum cost reduction comparing to the gate count optimal circuits from the previous column.

Table 5. Parameters of the quantum cost optimal circuit database for reversible functions requiring up to 23 units of quantum cost, constructed from mixed-polarity NCT gates

QC	# Functions	# Canonical Representatives of Functions	Hash Table Parameters				
			Max Entries	Size	Max Chain Length	Load Factor	Computation Time [s]
1	16	2	8	64B	1	25.0%	0
2	162	9	16	128B	4	56.3%	0
3	1,206	40	64	512B	5	62.5%	0
4	6,589	176	256	2KiB	29	68.8%	0
5	26,218	623	1,024	8KiB	30	60.8%	0
6	72,794	1,656	2,048	16KiB	56	80.9%	0
7	127,202	2,838	4,096	32KiB	50	69.3%	0
8	159,969	3,567	4,096	32KiB	95	87.1%	0
9	496,839	10,775	16,384	128KiB	41	65.8%	1
10	2,250,392	47,953	65,536	512KiB	87	73.2%	1
11	7,191,293	152,065	262,144	2MiB	64	58.0%	6
12	13,584,693	286,272	524,288	4MiB	55	54.6%	18
13	12,054,088	254,237	262,144	2MiB	13,240	97.0%	40
14	19,177,368	402,503	524,288	4MiB	156	76.8%	52
15	96,053,736	2,008,029	2,097,152	16MiB	4,734	95.8%	93
16	344,816,754	7,199,623	8,388,608	64MiB	804	85.8%	422
17	732,152,368	15,275,671	16,777,216	128MiB	1,556	91.1%	1,686
18	756,302,010	15,774,290	16,777,216	128MiB	3,501	94.0%	4,113
19	641,829,725	13,386,936	16,777,216	128MiB	350	79.8%	5,692
20	2,555,342,563	53,270,576	67,108,864	512MiB	417	79.4%	6,528
21	10,000,899,175	208,426,279	268,435,456	2GiB	317	77.6%	19,289
22	23,856,350,699	497,120,368	536,870,912	4GiB	3,308	92.6%	72,212
23	31,985,693,970	666,494,297	1,073,741,824	8GiB	132	62.0%	186,322

Table 6. Quantum cost optimal implementations with specified gate count for 4-bit benchmark functions [9], [22], [20] constructed from m-NCT gates

GC	QC	#circuits	time[s]	example circuit for QC_{min}
4b15g_1 [1,5,0,8,9,11,2,15,3,12,4,6,10,14,13,7]				
10	55-55	22	183	Ta'b'-c Tb'cd-a Ta'd-b Tb'c'-d Ca-d Tc'd'-a Tab'-c Ta'd'-c Tac'-b Cb'-a
13	**41**	1	-	Tabc-d Tac'-b Nb Cb-d Cb-a Cc-b Tad-b Cd-c Tb'c-d Ca-c Tc'd-a Cb-c Cd-b
4b15g_2 [1,9,0,4,10,8,2,11,3,15,5,12,7,14,13,6]				
11	37-82	146224	6371	Tad'-c Tb'c'-a Cb-c Cc-d Ta'd-c Tc'd-b Tbc'-d Tad-c Cc-a Ca'-d Cc-b
14	**34**	1	-	Tb'd-a Ca-d Cb-c Cc-a Tad'-b Tbc-d Cd-c Cb-a Ca-c Na Tcd-a Ta'b-c Cc-b Cd-a
4b15g_3 [3,1,7,13,11,0,8,15,2,5,10,6,9,14,12,4]				
11	37-91	308210	6600	Ca'-d Tcd'-b Cb-c Tcd-a Ta'd-b Ta'b-d Tbc'-a Ca'-b Tb'd-c Cc-d Cd-b
13	**34**	1	-	Ca-c Cc-b Cd-a Tab-d Tc'd'-a Ca-b Cb-c Cc-d Tad'-c Tbc'-a Cd-c Tab-d Cc-b
4b15g_4 [3,1,11,7,8,0,9,5,2,6,15,13,14,4,10,12]				
11	46-69	17296	3852	Tcd-a Tab'c-d Cd'-c Cb'-d Tcd-b Ca-c Tc'd'-a Ta'b-c Tbc'-d Ca-b Cb-a
13	**38**	1	-	Ta'b'd-c Cd-b Cc-d Tb'd-c Cd-a Ca-d Nd Cd-c Cc-b Cc-a Ta'd-c Tbc-d Cd-b Cb-c
4b15g_5 [3,5,11,1,8,0,9,7,2,6,14,13,10,4,12,15]				
10	37-67	600	96	Cd-a Ta'c-d Tb'd'-c Ca'-c Tb'c'-a Cd-b Ta'b-d Tac'-b Tbd-c Cd-a
13	**34**	1	-	Ca-b Tb'c-d Tad'-c Tb'c-a Ca-b Tb'd-a Cb-c Cc-d Ta'd-c Cd-b Cb-d Cc-a Na Ca-b
4_49 [15,1,12,3,5,6,8,7,0,10,13,9,2,4,14,11]				
9	**30**-78	490	1774	Cc-a Ta'b'-d Tcd-b Tad-c Tbc-a Cd-a Ta'b-d Cd-c Cd-b
9	**30**	1	-	Cc-a Ta'b'-d Tcd-b Tad-c Tbc-a Cd-a Ta'b-d Cd-c Cd-b
decode42 [1,2,4,8,0,3,5,6,7,9,10,11,12,13,14,15]				
6	39-42	10	2	Tc'd'-b Tb'd'-a Ta'b'c'-d Tb'd'-c Cd'-a Tad'-b
9	**29**	1	-	Cd-a Ca-c Tcd'-b Tab'c-d Ca-b Tbd'-a Ca-c Ca-b Na
hwb4 [0,2,4,12,8,5,9,11,1,6,10,13,3,14,7,15]				
10	**22**-50	11776	1798	Cc-a Tbd'-c Cb-d Ca-d Tc'd-b Tab'-c Cd-a Cb-d Ca-b Cc-a
10	**22**	1	-	Cd-b Ta'c-d Cb-c Ca-c Tcd-a Tab-d Cc-b Ca-c Cb-a Cd-b
imark [4,5,2,14,0,3,6,10,11,8,15,1,12,13,7,9]				
6	**19**-27	36	0	Tcd-a Tab-d Cd'-c Cb-c Cd-a Tac'-b
7	**19**	1	-	Tcd-a Tab-d Cd-c Cb-c Cd-a Tac-b Nc
mperk [3,11,2,10,0,7,1,6,f,8,14,9,13,5,12,4]				
8	**17**-35	2087	389	Tcd-b Cd'-c Tac-d Cd-a Cc-a Ca-c Cb-a Ca-b
9	**17**	1	-	Tcd-b Cd-c Tac'-d Cd-a Cc-a Ca-c Cb-a Na Ca-b
oc5 [6,0,12,15,7,1,5,2,4,10,13,3,11,8,14,9]				
9	38-63	2952	734	Cc'-a Tad-b Tbc'-d Ca-c Cc-b Ta'bd-c Cb-a Ta'd-b Tb'c-a
12	**36**	1	-	Cc-b Tb'd-c Cc-a Cb-c Cc-d Na Ca-b Tb'd-a Tac-d Cb-c Cb-a Tabd-c
oc6 [9,0,2,15,11,6,7,8,14,3,4,13,5,1,12,10]				
9	**38**-63	95	53	Ta'bc-d Cd-c Tbc-d Tb'd'-a Cc-b Tab'-c Tc'd-a Ca-d Ca-c
9	**38**	1	-	Ta'bc-d Tb'd'-a Cd-c Cc-b Tb'c-d Tab'-c Tc'd-a Ca-d Ca-c
oc7 [6,15,9,5,13,12,3,7,2,10,1,11,0,14,4,8]				
10	41-70	4097	90	Tad-c Cb-a Tacd-b Cb'-d Cc'-b Tbd-a Tac'-d Ta'd'-c Cb-c Cc-a
11	**40**	1	-	Tad-c Cb-a Tacd-b Nb Cb-d Cc-b Tbd-a Tac'-d Ta'd'-c Cb-c Cc-a

Table 6. *(continued)*

GC	QC	#circuits	time[s]	example circuit for QC_{min}
oc8 [11,3,9,2,7,13,15,14,8,1,4,10,0,12,6,5]				
9	43-81	12308	1517	Cc-a Ta'b'-d Ta'd-b Tbc-d Tcd-a Ta'b'd-c Cb-a Tad'-b Cb'-d
11	**39**	1	-	Tcd-b Ca-b Cb-d Cb-c Nb Ta'bd'-c Tcd-a Ta'd-b Tbc-d Cb-a Ca-c
nth_prime4_inc [0,2,3,5,7,11,13,1,4,6,8,9,10,12,14,15]				
8	56-56	16	408	Ta'cd'-b Tb'd-c Tb'c-d Tad'-b Tab'-c Tb'cd-a Tc'd-b Tbd'-a
12	**32**	1	-	Cd-b Tbc'-d Cc-b Tad'-c Cc-a Cb-c Cd-a Ta'c-b Cd-a Tbd-a Tac-d Cb-c
primes4 [2,3,5,7,11,13,0,1,4,6,8,9,10,12,14,15]				
8	33-55	512	7	Tc'd-b Cb-c Ta'c-d Tcd-a Cd'-b Tbc-d Tac-b Tbd-c
10	**26**	1	-	Tcd'-b Cd-a Cb-d Tad-b Cb-a Nc Cc-b Tbd'-c Cc-d Tb'd-c
mini_alu [0,1,2,3,4,5,14,11,8,6,10,9,12,15,13,7]				
6	30-78	36	0	Tbc-a Tad-c Tad-b Tbc-d Tbc-a Tad-c
8	**24**	1	-	Tbc-a Cb-c Tad-b Cd-a Tbc'-d Ta'd-c Cb-c Cd-a
aj-e11 [1,2,4,8,0,3,5,6,7,9,10,11,12,13,14,15]				
6	39-42	10	0	Tc'd'-b Tb'd'-a Ta'b'c'-d Tb'd'-c Cd'-a Tad'-b
9	**29**	1	-	Cd-a Ca-c Tcd'-b Tab'c-d Ca-b Tbd'-a Ca-c Ca-b Na
mod10_171 [1,2,3,4,5,6,7,8,9,0,10,11,12,13,14,15]				
5	46-49	28	0	Tabd'-c Tad'-b Tab'c'-d Tb'c'd-a Cd'-a
9	**37**	1	-	Tad'-b Ca-c Tc'd-a Tab'-c Tcd-a Ca-c Cd-c Tab'c'-d Na
mod10_176 [1,2,3,4,5,6,7,8,9,0,11,12,13,14,15,10]				
5	33-39	50	1	Tab-c Tac'd-b Tabc'-d Ca-b Na
8	**24**	1	-	Cc-d Tab-c Tc'd-b Tab-d Tc'd-b Ca-b Cc-d Na
4_49+hwb4 [15,2,3,12,5,9,1,11,0,10,14,6,4,8,7,13]				
9	34-34	12	153	Tbc'-d Tcd'-a Ca'-d Tc'd-b Tab-c Tb'd-a Tac'-b Cd-c Cb-a
13	**29**	1	-	Cb-dTc'd-a Cc-b Ta'b-d Cd-a Nd Tc'd-b Ca-c Cc-d Tab-c Cc-b Cd-c Cb-a
msaee [11,3,9,2,7,13,15,14,8,1,4,10,0,12,6,5]				
9	43-81	12308	20	Cc-a Ta'b'-d Ta'd-b Tbc-d Tcd-a Ta'b'd-c Cb-a Tad'-b Cb'-d
11	**39**	1	-	Tcd-b Ca-b Cb-d Cb-c Nb Ta'bd'-c Tcd-a Ta'd-b Tbc-d Cb-a Ca-c
gyang [2,5,3,15,4,13,6,7,8,9,10,11,12,1,14,0]				
5	50-50	4	0	Tabd'-c Tac-d Tad'-c Tbc'd'-a Ta'c'd'-b
11	**39**	1	-	Tab'd'-c Cd-b Nc Cc-d Tad-c Cc-b Tbc-a Ta'd-b Tbc-a Cc-d Nc
dmasl [0,1,14,3,4,5,7,8,15,13,10,6,9,12,11,2]				
7	31-47	24	22	Ta'd-b Cd-c Tbc-a Tc'd-a Ta'b-d Ta'd-c Tc'd-b
11	31	1	-	Tad-b Cd-c Tbc'-a Cd-b Tab-d Cd-b Cb-a Ca-c Cd-a Tad'-c Tcd-b
App2.2 [7,14,9,6,11,0,13,2,5,15,10,12,1,4,3,8]				
9	**38**-59	1067	46	Ca-b Tbc'd-a Ca'-d Tbd-c Ta'd-b Tbc'-d Tad'-b Nc Na
9	**38**	1	-	Ca-b Tbc'd-a Na Ca-d Tbd-c Nc Tad-b Tbc-d Ta'd'-b
App2.11 [7,14,9,6,11,0,13,2,5,15,10,12,1,4,3,8]				
7	32-52	113	1	Tac'-b Tb'c'-d Tbd-c Tad-b Tbd-a Tbc'-d Ca-b
8	**29**	1	-	Ca-b Tb'c'-d Tad'-c Tac'-b Tbd-a Ca-c Tbc'-d Ca-b

Table 7. Summary of the lowest gate count and the lowest quantum cost values for circuits implementing 4-bit reversible benchmark functions built from NCT and m-NCT gates

I	II		III		IV		V		VI	
Benchmark	**Best known (NCT) [5], [9], [20]**		**Best QC (NCT) [20]**		**Optimal GC (NCT) [20]**		**Optimal GC (m-NCT)**		**Best QC (m-NCT)**	
	GC	**QC**	**GC**	**QC**	**GC**	**QC**	**GC**	**QC**	**GC**	**QC**
4b15g_1	15	47	15	39	15	39	11	48	13	41
4b15g_2	15	61	15	31	15	31	11	37	14	34
4b15g_3	15	53	15	33	15	33	11	37	13	34
4b15g_4	15	47	15	35	15	35	11	46	14	38
4b15g_5	15	43	15	31	15	31	10	37	14	34
4_49	12	32	14	28	12	30	9	30	9	30
decode42	10	30	10	28	10	28	6	39	9	29
hwb4	11	21	13	19	11	21	10	22	10	22
imark	7	19	9	17	7	19	6	19	7	19
mperk	9	15	9	13	9	13	8	17	9	17
oc5	11	39	12	34	11	39	9	38	12	36
oc6	12	42	13	37	12	38	9	38	9	38
oc7	13	41	14	34	13	35	10	41	11	40
oc8	12	48	13	35	12	40	9	43	11	39
nth_prime4_inc	15	51	14	26	11	53	8	56	12	32
primes4	10	42	12	22	10	42	8	33	10	26
mini_alu	6	62	8	16	6	30	6	30	8	24
aj_e11	10	30	10	28	10	28	6	39	9	29
mod10_171	10	56	12	32	9	53	5	46	9	37
mod10_176	7	41	9	21	7	35	5	33	8	24
4_49+hwb4	12	30	14	26	12	28	9	34	13	29
msaee	16	72	14	34	12	40	9	43	11	39
gyang	19	103	14	36	10	52	5	50	11	39
dmasl	16	128	10	24	9	25	7	31	11	31
App2.2	18	102	13	35	11	39	9	38	9	38
App2.11	14	82	12	26	9	45	7	32	8	29
average:	**12.50**	**51.42**	**12.46**	**28.46**	**11.08**	**34.69**	**8.23**	**36.81**	**10.53**	**31.85**

7 Conclusions and Future Work

The main contribution of this paper are new results for circuits constructed from mixed-polarity NCT gates obtained from optimal circuit databases for such gates. Our results can be used for comparison with other reversible circuit synthesis algorithms which use mixed-polarity NCT gates. It has also been shown that using m-NCT gates reduces average gate count of the optimal circuits for the known benchmarks by 25.7% in comparison with gate count optimal circuits built from NCT gates. This approach can be extended in a similar manner as described in [21] and applied to resynthesis procedure for some parts of the circuits having more than 4 inputs/outputs.

We would like to find the maximum number of m-NCT gates required to construct the largest 4-bit optimal circuit. In the experiments described in the paper the largest optimal circuits required 11 m-NCT gates, but there might exist optimal circuits with more than 11 m-NCT gates. We plan to perform a computational experiment which would provide a complete distribution of reversible functions over the length of m-NCT optimal circuits to replace extrapolated values from Table 3 with exact results.

Acknowledgements. This work was supported by the Polish Ministry of Science and Higher Education under Grant 4180/B/T02/2010/38.

References

1. Arabzadeh, M., Saeedi, M., Zamani, M.S.: Rule-Based Optimization of Reversible Circuits. In: Proc. Asia-South Pacific Design Automation Conference, pp. 849–854. IEEE (2010)
2. Ardestani, E.K., Zamani, M.S., Sedighi, M.: A Fast Transformation-Based Synthesis Algorithm for Reversible Circuits. In: Proc. EUROMICRO Conference on Digital System Design, pp. 803–806 (2008)
3. Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D., Margolus, N., Shor, P., Sleator, T., Smolin, J., Weinfurter, H.: Elementary Gates for Quantum Computation. Phys. Rev. A 52, 3457–3467 (1995)
4. Golubitsky, O., Falconer, S.M., Maslov, D.: Synthesis of the Optimal 4-bit Reversible Circuits. In: Proc. Design Automation Conference, pp. 653–656. ACM (2010)
5. Golubitsky, O., Maslov, D.: A Study of Optimal 4-bit Reversible Toffoli Circuits and Their Synthesis. IEEE Trans. on Computers 61, 1341–1353 (2012)
6. Grosse, D., Wille, R., Dueck, G.W., Drechsler, R.: Exact Multiple Control Toffoli Network Synthesis with SAT Techniques. IEEE Trans. on CAD 28, 703–715 (2009)
7. Li, M., Zheng, Y., Hsiao, M.S., Huang, C.: Reversible Logic Synthesis Through Ant Colony Optimization. In: Proc. Design and Test in Europe Conference, pp. 307–310 (2010)
8. Markov, I.L., Saeedi, M.: Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation. Quantum Information and Computation 12, 361–394 (2012)
9. Maslov, D.: Reversible Logic Synthesis Benchmarks Page, `http://www.cs.uvic.ca/~dmaslov`
10. Maslov, D., Dueck, G.W.: Improved Quantum Cost for n-bit Toffoli Gates. Electronics Letters 39, 1790–1791 (2003)
11. Miller, D.M., Wille, R., Sasanian, Z.: Elementary Quantum Gate Realizations for Multiple-Control Toffoli Gates. In: Proc. 41st IEEE International Symposium on Multiple-Valued Logic, pp. 288–293. IEEE (2011)
12. Moraga, C.: Mixed Polarity Reed Muller Expressions for Quantum Computing Circuits. In: Proc. 10th Reed-Muller Workshop, pp. 119–125 (2011)
13. Moraga, C.: Hybrid GF(2) – Boolean Expressions for Quantum Computing Circuits. In: De Vos, A., Wille, R. (eds.) RC 2011. LNCS, vol. 7165, pp. 54–63. Springer, Heidelberg (2012)
14. Moraga, C.: Using Negated Control Signals in Quantum Computing Circuits. Facta Universitatis (Niš), Ser.: Elec. Energ. 24, 423–435 (2011)
15. Saeedi, M., Markov, I.L.: Synthesis and Optimization of Reversible Circuits – A Survey. ACM Computing Surveys (accepted 2012), available at arXiv: 1110.2574v1 (2011)
16. Saeedi, M., Zamani, M.S., Sedighi, M.: Moving Forward: A Non-Search Based Synthesis Method toward Efficient CNOT-based Quantum Circuit Synthesis Algorithms. In: Proc. Asia-South Pacific Design Automation Conference, pp. 83–88. IEEE (2008)

17. Saeedi, M., Sedighi, M., Zamani, M.S.: A Novel Synthesis Algorithm for Reversible Circuits. In: Proc. International Conference on Computer Aided Design, pp. 65–68 (2007)
18. Sasanian, Z., Miller, D.M.: Mapping a Multiple-Control Toffoli Gate Cascade to an Elementary Quantum Gate Circuit. Journal of Multiple-Valued Logic and Soft Computing 18, 83–98 (2012)
19. Szyprowski, M., Kerntopf, P.: Estimating the Quality of Complexity Measures in Heuristics for Reversible Logic Synthesis. In: Proc. IEEE Congress on Computational Intelligence, Congress on Evolutionary Computation (CD), 8 p. IEEE (2010)
20. Szyprowski, M., Kerntopf, P.: Reducing Quantum Cost in Reversible Toffoli Circuits. In: Proc. 10th Reed-Muller Workshop, pp. 127–136 (2011), corrected version available at arXiv: 1105.5831v2
21. Szyprowski, M., Kerntopf, P.: An Approach to Quantum Cost Optimization in Reversible Circuits. In: Proc. 11th IEEE Conference on Nanotechnology, pp. 1521–1526. IEEE (2011)
22. Wille, R., Grosse, D., Teuber, L., Dueck, G.W., Drechsler, R.: RevLib: An Online Resourse for Reversible Functions and Reversible Circuits, `http://www.revlib.org`
23. Wille, R., Soeken, M., Przigoda, N., Drechsler, R.: Exact Synthesis of Toffoli Gate Circuits with Negative Control Lines. In: Proc. 42nd IEEE International Symposium on Multiple-Valued Logic, pp. 69–74. IEEE (2012)
24. Zheng, Y., Huang, C.: A Novel Toffoli Network Synthesis Algorithm for Reversible Logic. In: Proc. Asia-South Pacific Design Automation Conference, pp. 739–744. IEEE (2009)
25. Zhu, W., Guan, Z., Hang, Y.: Reversible Logic Synthesis of Networks of Positive/Negative Control Gates. In: Proc. 5th IEEE International Conference on Natural Computation, pp. 538–542. IEEE (2009)

Design of an Online Testable Ternary Circuit from the Truth Table

Noor M. Nayeem and Jacqueline E. Rice

Dept. of Math & Computer Science
University of Lethbridge, Lethbridge, Canada
{noor.nayeem,j.rice}@uleth.ca

Abstract. This paper presents a new approach for converting a ternary reversible circuit implemented from a truth table into an online testable circuit. Our approach adds three extra lines to the given circuit, inserts Feynman gates and M-S gates, and replaces the ternary Toffoli gates (KP-m gates) with TKP-(m+1) gates. Our approach works with only 2×2 gates and 1×1 gates and covers a higher number of detectable faults. Preliminary work shows fault coverage of 84.89% when the approach is applied to a testable ternary half adder.

Keywords: Reversible logic, ternary circuits, online testing.

1 Introduction

Circuits built using traditional logic lose information during computation, which is dissipated as heat [1]. One solution to this loss of information is reversible logic. In particular, Bennett showed that a circuit consisting of only reversible gates dissipates zero energy [2]. There has, however, been little work on testability of reversible circuits, and even less in the area of multiple-valued testable reversible circuits. There is motivation to develop research in multiple-valued reversible logic, as this can provide a stepping stone to up-and-coming quantum technologies since quantum computing is inherently multiple-valued [3]. Previous work in online testing for reversible circuits includes our own in [4] as well as work in the Boolean domain such as [5] and [6].

In this work we introduce a new approach for converting a ternary reversible circuit into an online testable circuit, with fault coverage improvements over previous work reported in [4]. We emphasize that this work is ongoing, and preliminary results are reported here.

2 Background

2.1 Online Testing

Those of us in the field of computing are aware that testing is required to ensure quality and reliability. This applies to reversible logic circuits as well as to traditional logic circuits. According to [7], testing can be performed in the following ways: online; that is, while the circuit is operating normally; offline; or during a period while the circuit is not in use; or using some combination of both online and offline testing. The work proposed here is for an online testing approach, thus we would not require the circuit to be taken out of operation for the fault-detection to take place.

2.2 Fault Models

There are several fault models in reversible logic some of which include the missing, repeated and reduced gate fault models [8]. An additional and technology-independent fault model referred to as the bit fault model is used in various works including [5] and [6]. In this model a fault, possibly in a gate, would change the behavior of the gate's outputs. A single-bit fault is reflected on exactly one output of a gate, changing the correct value of the output to a faulty value. This model is somewhat reminiscent of the stuck-at fault model. We use this single-bit model in this work, although we note that the use of the term "bit" is not entirely accurate for ternary logic. The original concept of this model is still valid, however, as we are identifying the situation when a fault is reflected on exactly one output of a gate.

2.3 Ternary Galois Field Logic

The Ternary Galois Field (TGF) consists of {0, 1, 2} and two operations, addition modulo 3 and multiplication modulo 3. We denote addition modulo 3 by $\oplus$ and multiplication modulo 3 by the absence of any operator. For a ternary variable a, we have $a = a \oplus 3$ and $aaa = a$. According to [9], a ternary variable a has six basic literals: a, $a^{+1} = a \oplus 1$, $a^{+2} = a \oplus 2$, $a^{12} = 2a$, $a^{01} = 2a \oplus 1$, and $a^{02} = 2a \oplus 2$.

2.4 Reversible Ternary Gates

We define here the ternary reversible gates which are required for this paper. A 1-qutrit permutative gate [10] is defined as $\{a\} \rightarrow \{b = a^Z\}$ where $Z \in \{+1, +2, 12, 01, 02\}$ as shown in Fig. 1(a). For example, if $Z = +1$, then $b = a^{+1} = a \oplus 1$.

The Feynman gate has the input vector $[a_1, a_2]$ and output vector $[b_1 = a_1, b_2 = a_1 \oplus a_2]$. The modified Feynman gate discussed in [11] is very similar to the Feynman gate with the exception that $b_2 = 2a_1 \oplus a_2$. The Feynman gate and its modified version are shown in Fig. 1(b) and Fig. 1(c).

A 2-qutrit Muthukrishnan-Stroud (M-S) gate [10] is defined as mapping the input vector $[a_1, a_2]$ to the output vector $[b_1 = a_1, b_2 = r]$ where $r = a_2^Z$ if $a_1 = 2$; otherwise $r = a_2$. Here a_1 is the controlling input and a_2 is the controlled input. An M-S gate is shown in Fig. 1(d).

Khan and Perkowski proposed a ternary Toffoli gate [10]; however its behaviour is somewhat different from the commonly accepted behaviour for a

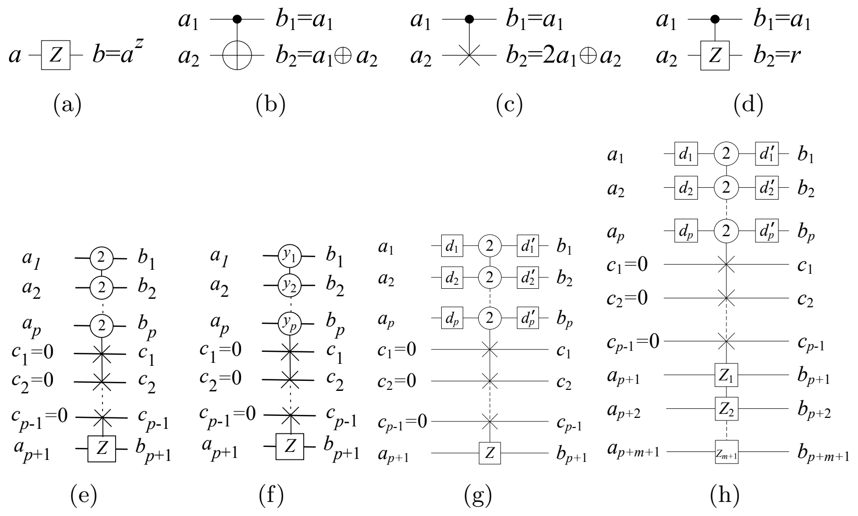


Fig. 1. (a) A permutative gate, (b) Feynman gate, (c) modified Feynman gate, (d) M-S gate, (e) $(p+1)$-qutrit KP gate, (f) $(p+1)$-qutrit generalized KP gate, (g) equivalent representation of a generalized KP gate, and (h) KP-m gate

Toffoli gate and so to avoid confusion we refer to this gate as a KP gate. A $(p+1)$-qutrit KP gate is shown in Fig. 1(e). This gate maps the input vector $[a_1, a_2, \ldots, a_{p+1}]$ to the output vector $[b_1 = a_1, b_2 = a_2, \ldots, b_p = a_p, b_{p+1} = r]$ where $r = a_{p+1}^{Z}$ if $a_1 = a_2 = \ldots = a_p = 2$; otherwise $r = a_{p+1}$. Here $a_1, a_2, \ldots, a_p$ are controlling inputs and a_{p+1} is the controlled input. An implementation of this gate requires p input lines, $p-1$ constant lines, and an output line [10].

Khan and Perkowski also proposed a generalized KP gate [10] as shown in Fig. 1(f), which is very similar to the KP gate. In this gate $r = a_{p+1}^{Z}$ if $a_1 = y_1, a_2 = y_2, \ldots, a_p = y_p$; otherwise $r = a_{p+1}$. An equivalent representation of this gate is shown in Fig. 1(g). This version is built using the KP gate and 1-qutrit permutative gates where $d_j = 2 - y_j$ and $d_j + d'_j = 0$ for $j = 1, 2, \ldots, p$. The permutative gate $+d_j$ is used to change the controlling input of KP gate to 2, and the permutative gate $+d'_j$ restores the controlling value.

We can extend the KP gate and the generalized KP gate for multiple controlled inputs. For example, a $(p+1)$-qutrit generalized KP gate with $m+1$ controlled inputs, denoted as a KP-m gate, is shown in Fig. 1(h). The KP-m gate has the input vector $[a_1, a_2, \ldots, a_p, a_{p+1}, \ldots, k_{p+m+1}]$ and the output vector $[b_1 = a_1, b_2 = a_2, \ldots, b_p = a_p, b_{p+1} = r_1, b_{p+2} = r_2, \ldots, b_{p+m+1} = r_{m+1}]$ where $r_k = a_{p+k}^{Z_k}$ if $a_1 = y_1$, $a_2 = y_2$, …, $a_p{=}y_p$ and $Z_k \in \{+1, +2, 12, 01, 02\}$ for $k = 1, 2, \ldots, m+1$; otherwise $r_k = a_{p+k}$. Here, $a_1, a_2, \ldots, a_p$ are controlling inputs and $a_{p+1}, a_{p+2}, \ldots, a_{p+m+1}$ are controlled inputs. Like the $(p+1)$-qutrit KP gate and generalized KP gate, a $(p+1)$-qutrit KP-m gate also requires $p-1$ constant lines.

In order to design the testable ternary circuit we add the following constraints to the KP-m gate: $Z_k \in \{+1, +2\}$ ($k = 1, 2, \ldots, m$) and $Z_{m+1} = Z_1 \oplus Z_2 \oplus \ldots \oplus Z_m$. To distinguish this gate from the KP-m gate we call this gate a TKP-m gate (testable KP-m). The symbol of a $(p + 1)$-qutrit TKP-m gate is shown in Fig. 2.

2.5 Synthesis of Ternary Reversible Circuits

Several approaches such as [12], [13] and [14] have been proposed for synthesis of ternary reversible circuits. In this section we briefly describe an approach [10] to realize a ternary circuit from the truth table.

Consider a ternary function with p input variables $x_1, x_2, \ldots, x_p$ and q output variables $f_1, f_2, \ldots, f_q$. An empty cascade with p input lines $(I_1, I_2, \ldots, I_p)$, $p-1$ constant lines $(I_{p+1}, I_{p+2}, \ldots, I_{2p-1})$ and q output lines $(I_{2p}, I_{2p+1}, \ldots, I_{2p+q-1})$ is generated. For each input combination $x_1x_2 \ldots x_p$ ($x_i \in \{0, 1, 2\}$, for $i = 1, 2, \ldots, p$) with $m + 1$ outputs ($0 \leq m < q$) having values 1 or 2 in the truth table, a $(p + 1)$-qutrit generalized KP-m gate with $x_1 = y_1, x_1 = y_1, \ldots, x_p = y_p$ is added to the circuit. Each controlling input a_i of the gate is connected to the input line I_i. For each $f_j = 1$ (or 2) $j = 1, 2, \ldots, q$, a controlled input is connected to the output line I_{2p+j-1} with $Z = +1$ (or $+2$). Although we have described this approach using generalized KP-m gates the circuit can also be generated using KP-m gates and permutative gates.

As an example, given the truth table of a ternary function with two input variables (x_1 and x_2) and two output variables (f_1 and f_2) as shown in Table 1, a ternary circuit is implemented as shown in Fig. 3. For the first input combination, a 3-qutrit KP-1 gate is added. The controlling inputs of this gate are connected to lines I_1 and I_2 and the controlled inputs are connected to lines I_4 and I_5. Two 3-qutrit KP gates are added for the second and third input combinations. No more gates are added since both f_1 and f_2 have values 0 for all other input combinations.

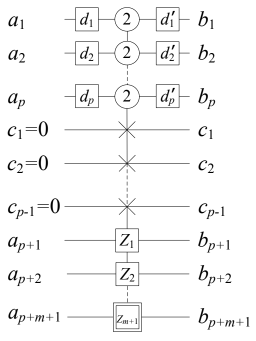


Fig. 2. $(p + 1)$-qutrit TKP-m gate

Table 1. Truth table of a ternary function

x_1	x_2	f_1	f_2
0	0	1	1
0	1	1	0
0	2	0	2
1	0	0	0
1	1	0	0
1	2	0	0
2	0	0	0
2	1	0	0
2	2	0	0

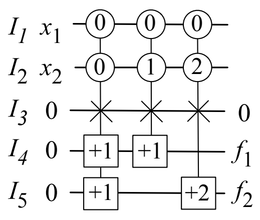


Fig. 3. A ternary circuit

3 Our Approach

3.1 Design

Consider a reversible ternary circuit generated from the truth table as discussed in Section 2.5. A circuit generated in this way consists of only permutative gates, KP-m gates, and generalized KP-m gates. If the circuit has p input lines and q output lines, then the circuit also has $p-1$ constant lines. We refer to the input lines as $I_1, I_2, \ldots, I_p$, the constant lines as $I_{p+1}, I_{p+2}, \ldots, I_{2p-1}$, and the output lines as $I_{2p}, I_{2p+1}, \ldots, I_{2p+q-1}$. If the initial values and final values of any input line (or constant line) are not the same, then the permutative gates and M-S gates are added to restore the initial value at the end of the corresponding line. The following approach converts such circuit into an online testable circuit.

Our proposed approach requires three extra lines, L_1, L_2, and L_3, each of which is initialized with a zero. All permutative gates found in the given circuit are retained. Each KP-m gate is replaced by a TKP-(m+1) gate. The connections of the TKP-($m+1$) gate remain the same as that of KP-m gate with the last controlled input connected to L_1.

For each input line in the given circuit, this approach adds a Feynman gate and a modified Feynman gate at the beginning and at the end of the circuit, respectively. For each such gate, the controlling input is connected to I_u (for $u = 1, 2, \ldots, p$) and controlled input is connected to L_2. At the end of each constant line we first add a permutative ($Z = +2$) gate, then an M-S gate ($Z = +1$) with controlling input connected to I_v (for $v = p+1, p+2, \ldots, 2p-1$) and controlled input connected to L_3. Another permutative gate ($Z = +1$) is also added on the constant line to restore the value. Finally, at the end of each output line a modified Feynman gate is added with controlling input connected to I_w (for $w = 2p, 2p+1, \ldots, 2p+q-1$) and controlled input connected to L_1. This approach requires the addition of p Feynman gates, $p+q$ modified Feynman gates, $p-1$ M-S gates and $2p-2$ permutative gates.

If a single fault occurs in any gate other than the Feynman gate and the modified Feynman gate, then L_1 or L_2 will be non-zero, or L_3 will not be equal to $1 \oplus 2 \oplus \ldots \oplus p-1$. If no fault occurs in the circuit, then L_1 and L_2 will remain 0, and L_3 will be equal to $1 \oplus 2 \oplus \ldots \oplus p-1$. Thus existence of a fault can be detected by examining the values of L_1, L_2 and L_3.

The following example describes this approach. For a ternary circuit as shown in Fig. 3, our proposed approach generates an online testable circuit as shown in Fig. 4. In the testable circuit three extra lines are added and the KP-m gates are replaced by TKP-($m+1$) gates. In addition, two Feynman gates, four modified Feynman gates, one M-S gate and two permutative gates are added.

3.2 Fault Detection

Our proposed approach makes use of the TKP-($m+1$) gates and generalized TKP-($m+1$) gates, which can be decomposed into M-S gates and permutative gates. In this section, we consider a low level design of our testable circuit

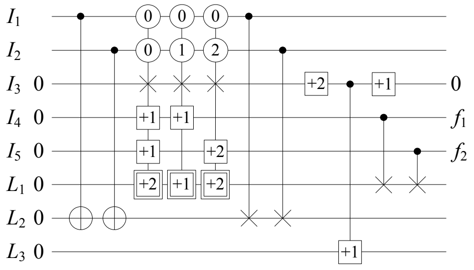


Fig. 4. An online testable ternary circuit

consisting of 1×1 gates and 2×2 gates which are permutative gates, M-S gates, Feynman gates, and the modified Feynman gates.

A single fault on a line can propagate to several lines via M-S gates. This causes multiple faults in the circuit. Consider an M-S gate as shown in Fig. 1(d). A fault in the controlling input a_1 (or controlled input a_2) of an M-S gate affects this gate since it causes b_1 (or b_2) to have the faulty value. It is noted that a fault in a_2 cannot propagate to b_1 since b_1 is independent of a_2. However, if a fault changes the value of a_1 to 2 (or changes the value from 2 to either 0 or 1), then the fault also propagates to b_2 since the value of b_2 depends not only on a_2 but also on a_1.

Our approach ensures detection of a single fault in any M-S gate and permutative gate even though the fault may propagate to multiple lines. Proofs are omitted due to page limitations. However, this testable circuit is unable to detect a fault in Feynman gates and modified Feynman gates, which are added to make the circuit testable. Ongoing work is addressing this.

4 Discussion

We have implemented a non-testable ternary half adder from the truth table using the method described in Section 2.5. The cost of the circuit when built in this way is 50 (based on the cost metrics given in [10]). We have applied our approach to convert this into a testable adder with a final cost of 85. The overhead of adding the testability is in this case 70%. We calculate the fault coverage of our circuit to be 84.89% based on the single-bit fault model from [5].

We point out that our approach adds exactly p Feynman gates, $p+q$ modified Feynman gates, $p-1$ M-S gates, and $2p-2$ permutative gates regardless of the number of gates in the given circuit. This results in a higher overhead cost for a small circuit such as ternary half adder. For circuits with higher numbers of gates the overhead costs will be reduced. In addition a larger circuit will cover a higher number of detectable faults since the number of Feynman gates and modified Feynman gates becomes smaller compared to other gates in a large circuit. Future work includes computation of the overhead cost and fault coverage for large benchmark circuits.

5 Comparison with Related Work

The authors in [15] proposed an approach to design a ternary circuit with online testability. This approach proposes two blocks TR1 and TR2 which are cascaded together to form a testable block (TB) that can perform a single operation such as addition or multiplication. A number of TBs are used to design the circuit. A checker circuit is also required to test the TBs. It is noted that this approach can detect a single fault in a block (TR1 or TR2) if the fault reflects to only one of the outputs of that block. Since the blocks TR1 and TR2 are very large, consisting of many gates, it is very likely that a fault will be reflected to several outputs of a faulty block. In this case, this approach will fail to detect such fault.

Another approach proposed in [4] provides a simple way to convert a non-testable circuit into a testable one. This approach also fails to detect a fault in a large gate if the fault reflects to multiple outputs of that gate.

In contrast the approach described here considers a low level design consisting of only 1×1 gates and 2×2 gates, rather than considering a design with large blocks or gates. Thus the fault coverage of this approach is much higher than that of the previously mentioned approaches.

6 Conclusion

We have introduced a technique that takes a ternary reversible circuit generated as described in [10] and transforms the circuit into an online testable circuit. This is achieved through the addition of three additional lines and p Feynman gates, $p + q$ modified Feynman gates, $p - 1$ M-S gates, and $2p - 2$ permutative gates where p is the number of input lines and q is the number of output lines. Our preliminary work is showing similar overhead to that in [4]; that is, for small circuits the overhead percentage is high, but as gate counts increase this is reduced. The resulting fault coverage seems quite good, although further work on larger benchmarks is continuing. Future work will also include further analysis and comparisons to related work such as [6] and [5].

References

1. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5, 183–191 (1961)
2. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17(6), 525–532 (1973)
3. Nielsen, M., Chuang, I.: Quantum Computation and Quantum Information. Cambridge University Press (2000)
4. Nayeem, N.M., Rice, J.E.: A new approach to online testing of TGFSOP-based ternary Toffoli circuits. In: Proceedings of the 42nd International Symposium on Multiple-Valued Logic (ISMVL), Victoria, BC, Canada, May 14-16 (to appear, 2012)
5. Vasudevan, D.P., Lala, P.K., Jia, D., Parkerson, J.P.: Reversible logic design with online testability. IEEE Transactions on Instrumentation and Measurement 55(2), 406–414 (2006)

6. Mahammad, S.N., Veezhinathan, K.: Constructing online testable circuits using reversible logic. IEEE Transactions on Instrumentation and Measurement 59(1), 101–109 (2010)
7. Wang, L., Wu, C., Wen, X. (eds.): VLSI Test Principles and Architectures: Design for Testability. Morgan Kaufmann (2006)
8. Hayes, J.P., Polian, I., Becker, B.: Testing for missing-gate faults in reversible circuits. In: Proceedings of the 13th Asian Test Symposium, pp. 100–105 (2004)
9. Khan, M.H.A., Perkowski, M., Khan, M.R., Kerntopf, P.: Ternary GFSOP minimization using Kronecker decision diagrams and their synthesis with quantum cascades. Journal of Multiple-Valued Logic and Soft Computing 11(5-6), 567–602 (2005)
10. Khan, M.H.A., Perkowski, M.A.: Quantum ternary parallel adder/subtractor with partially-look-ahead carry. Journal of Systems Architecture 53(7), 453–464 (2007)
11. Rahman, M.R.: Online testing in ternary reversible logic. Master's thesis, University of Lethbridge (2011)
12. Khan, M.H.A., Perkowski, M., Kerntopf, P.: Multi-output Galois field sum of products synthesis with new quantum cascades. In: Proceedings of the 33rd International Symposium on Multiple-Valued Logic, pp. 146–153 (2003)
13. Miller, D.M., Maslov, D., Dueck, G.W.: Synthesis of quantum multiple-valued circuits. Journal of Multiple-Valued Logic and Soft Computing 12(5-6), 431–450 (2006)
14. Khan, M.H.A.: GFSOP-based ternary quantum logic synthesis. In: Proceedings of SPIE 7797 (Optics and Photonics for Information Processing IV), San Diego, California (2010)
15. Rahman, M.R., Rice, J.E.: On designing a ternary reversible circuit for online testability. In: Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), Victoria, B.C., Canada, pp. 119–124 (August 2011)

Garbageless Reversible Implementation of Integer Linear Transformations

Stéphane Burignat[1], Kenneth Vermeirsch[1], Alexis De Vos[1,2], and Michael Kirkedal Thomsen[3]

[1] Department of Electronics and Information Systems
[2] Imec v.z.w., Universiteit Gent, B - 9000 Gent, Belgium
{sburigna,Kenneth.Vermeirsch,alex}@elis.ugent.be
[3] Department of Computer Science,
University of Copenhagen, DK - 2100 Copenhagen, Denmark
michael@kirkedal.dk

Abstract. Discrete linear transformations are important tools in information processing. Many such transforms are injective and therefore prime candidates for a physically reversible implementation into hardware. We present here reversible digital implementations of different integer transformations on four inputs. The resulting reversible circuit is able to perform both the forward transform and the inverse transform. Which of the two computations that actually is performed, simply depends on the orientation of the circuit when it is inserted in a computer board (if one takes care to provide the encapsulation of symmetrical power supplies). Our analysis indicates that the detailed structure of such a reversible design strongly depends on the prime factors of the determinant of the transform: a determinant equal to a power of 2 leads to an efficient garbage-free design.

1 Introduction

Linear transforms are important in analysis and compression of audio signals, images, video, and much more. A common property of most of these transforms is the existence of an inverse transform, meaning that they, in theory, are lossless reversible functions.

An important transform is the Fourier transform. Its fast discrete implementation is widely used in digital signal processing. For the Fourier transform there exists an inverse transform and it has therefore been researched in a reversible context [1]. A problem with the Fourier transform is the use of non-integer and complex values, that in numerical computations result in rounding of fixed-point or floating-point numbers and thus a lossy coding. To avoid complex-number arithmetic, the Fourier transform is often replaced by a discrete cosine transform (DCT), for which real-number arithmetic is sufficient. However, it still suffers from rounding errors due to floating-point arithmetic. Integer transforms, which are invertible integer approximations of a real-valued linear transform such as the DCT, can solve this last problem.

For implementation in reversible computing the ‘lifting scheme’ [2] is a powerful tool. The lifting scheme decomposes an injective computation into a series of ‘reversible updates’ [3]. Designs throughout this paper are based on reversible logic as described by Fredkin and Toffoli [4]. A key element in our designs is the V-shaped reversible binary adder introduced by Vedral et al. [5] and improved by Cuccaro et al. [6]. Detailed descriptions of these can be found in [7, 8]; detailed tests and measurements of adiabatic calculations in a fabricated adder can be found in [9]. The actual physical circuit is implemented using reversible dual-line pass-transistor CMOS technology [8–10].

2 Approximating the Discrete Cosine Transform

The 4×4 discrete cosine transform is defined by

$$C = \begin{pmatrix} a & a & a & a \\ b & c & -c & -b \\ d & -d & -d & d \\ c & -b & b & -c \end{pmatrix} , \tag{1}$$

where

$$\begin{aligned} a &= \tfrac{1}{2}\cos(0) = \tfrac{1}{2} &&= 0.500 \\ b &= \tfrac{1}{\sqrt{2}}\cos(\pi/8) = \tfrac{\sqrt{2+\sqrt{2}}}{2\sqrt{2}} &&\approx 0.653 \\ c &= \tfrac{1}{\sqrt{2}}\cos(3\pi/8) = \tfrac{\sqrt{2-\sqrt{2}}}{2\sqrt{2}} &&\approx 0.271 \\ d &= \tfrac{1}{\sqrt{2}}\cos(\pi/4) = \tfrac{1}{2} &&= 0.500 \; . \end{aligned}$$

Its matrix determinant is

$$\det(C) = 8ad(c^2 + b^2) = 1 \; . \tag{2}$$

An integer transform is obtained by successively multiplying by a constant scalar α and rounding [11]:

$$H_\alpha = \text{round}(\alpha C) \; .$$

We have $\det(H_\alpha) \approx \det(\alpha C) = \alpha^4 \det(C) = \alpha^4$.

Choosing α equal to unity, yields the matrix

$$H_1 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & -1 \\ 1 & -1 & -1 & 1 \\ 0 & -1 & 1 & 0 \end{pmatrix} ,$$

with determinant equal to 8, i.e. a value deviating surprisingly much from $\alpha^4 = 1$. We see how the rounding step strongly influences the value of the determinant.

Choosing α equal to 2, gives the transformation

$$H_2 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & -1 & 1 & -1 \end{pmatrix} ,$$

with determinant equal to 16, i.e. exactly equal to α^4. This is the well-known Walsh–Hadamard matrix, applied in data encryption, signal processing and data compression algorithms (such as MPEG-4) [12].

Given an invertible matrix (a matrix with non-zero determinant), it is possible to automatically generate a lifting scheme using decomposition. One algorithm for this is described in detail by De Vos and De Baerdemacker [13]. However, for transformation matrices with built-in symmetries, like the cosine transform, such standard decomposition is far from optimal. More efficient decompositions into scaling, swapping, and lifting matrices are possible [14]. Figure 1 shows a decomposition of the Walsh–Hadamard matrix, involving only four scalings (all with scaling factor 2), eight liftings (all with lifting factor ± 1) and four swaps. The circuit generates no garbage and has an appealing modular structure: it is built from four 2×2 Hadamard matrices

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} .$$

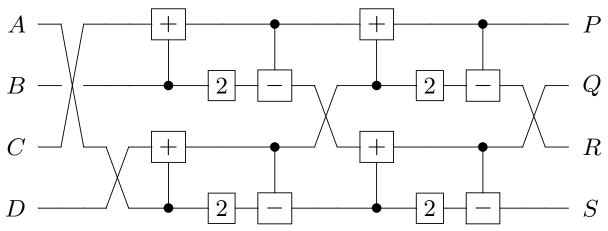


Fig. 1. Optimal lifting scheme for the Walsh–Hadamard 4×4 matrix

The fact that the circuit does not generate garbage numbers is a most valuable property. It means the circuit can be used in reverse direction (i.e. from right to left in Fig. 1) in order to perform the inverse operation:

$$H_2^{-1} = \frac{1}{4}\, H_2 \ .$$

3 The H.264 Transform

For many applications, the Walsh–Hadamard is considered as too 'crude' approximation of the cosine transform. Values of α larger than 2 yield increasingly more precise approximations to the real-valued DCT, but the resulting matrix coefficients will contain more diverse prime factors. Therefore, often the compromise choice $\alpha = 2.5$ is used. After multiplication by 2.5, matrix (1) becomes

$$\begin{pmatrix} 1.25 & 1.25 & 1.25 & 1.25 \\ 1.63 & 0.68 & -0.68 & -1.63 \\ 1.25 & -1.25 & -1.25 & 1.25 \\ 0.68 & -1.63 & 1.63 & -0.68 \end{pmatrix} . \tag{3}$$

Subsequent rounding yields

$$H_{2.5} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{pmatrix} , \tag{4}$$

with determinant 40, i.e. a value close to $\alpha^4 = \left(\frac{5}{2}\right)^4 = \frac{625}{16} \approx 39.063$. The choice of α is motivated by the absence of true multiplications in the resulting transformation. Indeed, conventional implementation of the factors of 2 in (4) is trivial both in hardware and in software, by merely using a bit shift operation.

The H.264 transform [11] is used in the MPEG-4/AVC video format. Figure 2 shows a decomposition with only four scalings, eight liftings and four swaps [14].

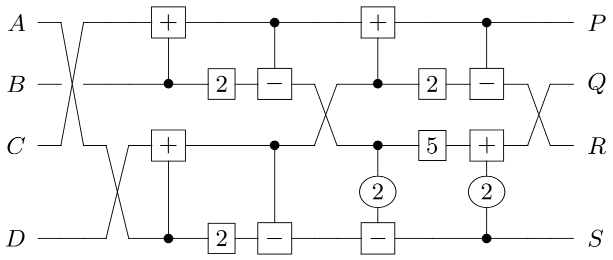
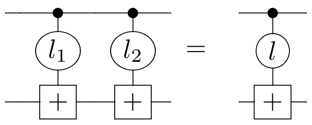


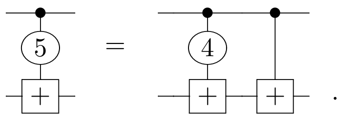
Fig. 2. Optimal lifting scheme for the H.264 discrete cosine transformation

In contrast to the example of Sect. 2, we face here the problem of a scaling factor 5, i.e. a scaling that is different from a power of 2, the so-called ‘perfect’ coefficients [15]. Whereas scaling factors equal to a power of 2 can be implemented as a bit shift, other scaling factors cannot.

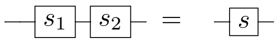
A lifting factor different from a power of 2 is no problem, because lifting factors add up:



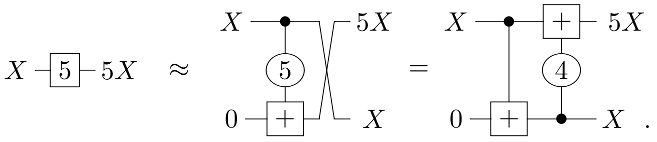
with $l = l_1 + l_2$. For example, the lifting factor 5 can be implemented with two 'perfect' liftings:



In contrast, a scaling factor different from a power of 2 is a problem, because scaling factors multiply:



with $s = s_1 s_2$. If the prime factorization of a scaling factor s contains prime factors different from 2, we need to convert the scaling into a lifting, by introducing a preset input and a garbage output:



The extra (preset) input line and extra (garbage) output line in fact means that we are embedding the 4×4 matrix within a 5×5 matrix. While the former has a determinant with an odd prime factor, the latter has a determinant which is a power of 2. Applying the above scaling-to-lifting transformation is embedding matrix (4) in the matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \mathbf{0} \\ 2 & 1 & -1 & -2 & \mathbf{4} \\ 1 & -1 & -1 & 1 & \mathbf{0} \\ 1 & -2 & 2 & -1 & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{0} \end{pmatrix},$$

with determinant equal to -8, i.e. a 'perfect' value.

Instead of replacing only the scaling factor 5 by a lift, it is advantageous to replace the whole block consisting of the non-perfect scaling factor together with both the preceding lift and the succeeding lift giving the embedding of matrix (4) in

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \mathbf{0} \\ 2 & 1 & -1 & -2 & \mathbf{0} \\ 1 & -1 & -1 & 1 & \mathbf{0} \\ 1 & -2 & 2 & -1 & \mathbf{-2} \\ \mathbf{0} & \mathbf{1} & \mathbf{-1} & \mathbf{0} & \mathbf{1} \end{pmatrix}, \tag{5}$$

with determinant equal to 8 and the lifting scheme shown in Fig. 3. Whereas the former embedding yields intermediate results ranging from -5 to 5 times the input data, the latter embedding restricts all intermediate and final data to the range from -3 to 4 times the input data. Thus reducing the increase in dynamic range of the signals is a valuable improvement.

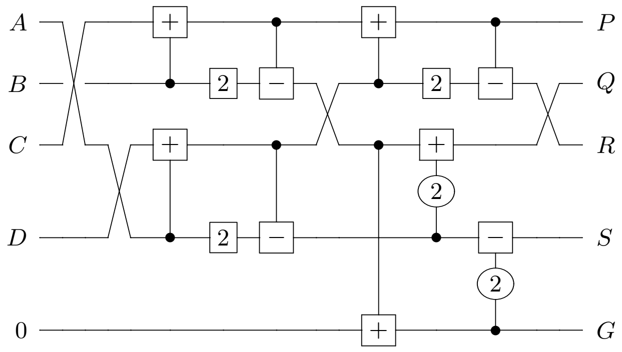


Fig. 3. Lifting scheme for the H.264 transform with simple multipliers but with garbage

We end this section by noting that the appearance of a scaling factor different from 2^k is a direct and inevitable consequence of the fact that the determinant of the transform matrix does not have the form $\pm 2^k$. Indeed, as the determinant of any product of several matrices equals the product of the determinants of the separate matrices, the determinant of the whole circuit equals the product of the determinants of all the subcircuits. Because

– a swap has determinant -1,
– a lifting has determinant 1 (irrespective of its lifting factor l), and
– a scaling has determinant equal to its scaling factor s,

the full circuit has determinant $(-1)^m \prod_j s_j$, where m is the number of swaps in the circuit and s_j are the scaling factors of the circuit. For example, we can immediately see from Fig. 2 that its determinant equals $(-1)^4 \times 2 \times 2 \times 2 \times 5 = 40$, in accordance with the determinant of (4).

4 Implementing H.264

Our H.264 prototype coder is designed using four 3-bit unsigned integer inputs (A, B, C, and D) and four 6-bit signed integer outputs (P, Q, R, and S). The schematic implementation of the linear transform, given in (5) and Fig. 3, is designed in the *Cadence* computer-aided design environment, applying reversible dual-line pass-transistor CMOS style [8].

The design consists of two 4-bit reversible binary adders, two 4-bit reversible binary subtractors, one 5-bit adder, one 5-bit subtractor, one 6-bit adder, one 6-bit subtractor, and ten Feynman gates. Each reversible adder and subtractor being composed of $48w - 32$ transistors (where w is the bit width of the data: either 4, 5, or 6) and each Feynman of 8 transistors; the whole circuit thus contains 1648 transistors.

The prototype silicon chip contains a total of 1704 transistors (852 n-MOS transistors and 852 p-MOS transistors): the 1648 transistors for the actual coder plus 56 transistors for a small diagnostic circuits (that enables probing of intermediate results). It has been designed and fabricated in the standard CMOS technology of the foundry *ON Semiconductor*. The length of the transistors is 0.35 μm; the width is either 0.5 μm (n-MOS) or 1.5 μm (p-MOS). The threshold voltages are 0.6 volt (n-MOS) and -0.6 volt (p-MOS). The complete coder circuit fits into a rectangle of 610 μm $\times$ 300 μm. Figure 4 shows the prototype. One easily recognizes four 4-bit Cuccaro blocks (at the left-hand side), two 5-bit Cuccaro blocks (at the right-bottom part), and two 6-bit Cuccaro blocks (at the right-top part). The complete chip (bonding pads included) measures 2.2 mm $\times$ 2.2 mm = 4.8 mm^2 and is encapsulated in a 68-JLCC package.

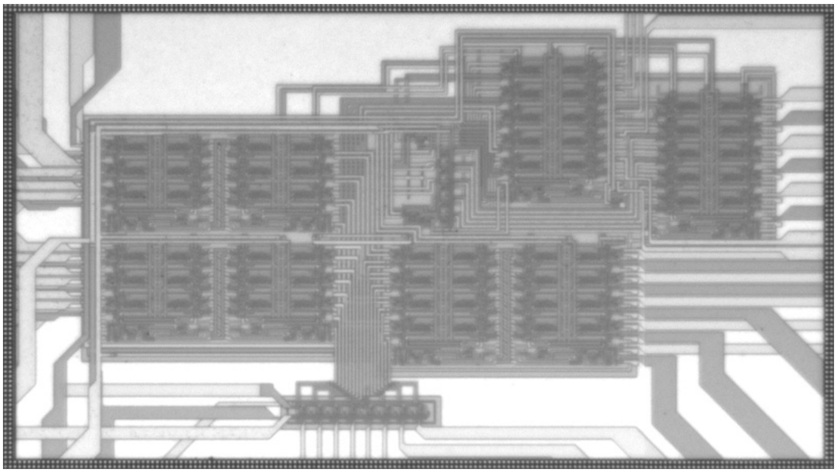


Fig. 4. Microscope view of the integrated circuit (680 μm $\times$ 380 μm)

Figure 5 shows a measurement of the functioning of the circuit. Positive voltages represent a logic 1; negative voltages represent a logic 0. Such so-called adiabatic signals are provided to each one of the inputs. We here see the input bit $A_0 = 1$. The resulting output bit P_4 turns out to be 0. In this example, the input vector $(A, B, C, D) = (111, 011, 000, 011)$ is used. Several other vectors have been tested, yielding positive as well as negative (coded in two's complement) output numbers.

5 Avoiding Garbage

The implementation of the discrete matrix is optimal with respect to the number of lifting steps and transistors. Unfortunately, as mentioned in Sect. 3, reversible implementation of the H.264 transform is hampered by the fact that its determinant equals $40 = 2^3 \times 5$, instead of a power of 2. The prime factor 5 is responsible

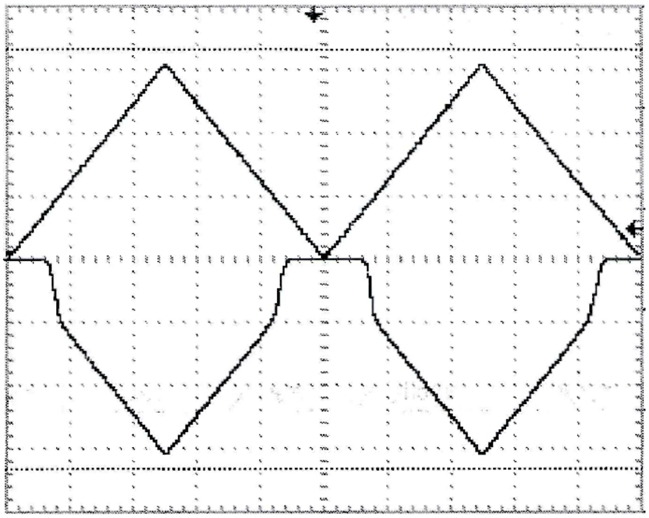

Vertical scale = 500 mV/div.; horizontal scale = 5 ms/div.

Fig. 5. Oscilloscope view of input bit $A_0 = 1$ and output bit $P_4 = 0$, for the input vector $A = 7$, $B = 3$, $C = 0$, and $D = 3$ (resulting in the output vector $P = 13$, $Q = 11$, $R = 7$, and $S = -2$)

for the creation of the garbage output G. This garbage datum makes it difficult to use of a same chip for both coding and decoding. In order to decode, i.e. to apply the inverse coding

$$H_{2.5}^{-1} = \frac{1}{20} \begin{pmatrix} 5 & 4 & 5 & 2 \\ 5 & 2 & -5 & -4 \\ 5 & -2 & -5 & 4 \\ 5 & -4 & 5 & -2 \end{pmatrix} ,$$

knowledge of the numbers P, Q, R, and S is insufficient to recover the values of A, B, C, and D. We need the extra knowledge of the value of G, an information that is not necessarily available. Standard solutions for this problem exist [16], uncomputing garbage, however at the expense of much extra hardware.

Therefore, it is worth investigating whether, especially for reversible hardware, an appropriate choice of the factor α (instead of $\alpha = 5/2$) would be advantageous. According to $\det(H_\alpha) \approx \alpha^4$, a choice $\alpha = 2^n$ looks promising. We therefore try the numbers 4 and 8. They result in the integer transforms

$$H_4 = \begin{pmatrix} 2 & 2 & 2 & 2 \\ 3 & 1 & -1 & -3 \\ 2 & -2 & -2 & 2 \\ 1 & -3 & 3 & -1 \end{pmatrix} \quad \text{and} \quad H_8 = \begin{pmatrix} 4 & 4 & 4 & 4 \\ 5 & 2 & -2 & -5 \\ 4 & -4 & -4 & 4 \\ 2 & -5 & 5 & -2 \end{pmatrix} ,$$

respectively. Unfortunately, because of the rounding of the matrix entries, the former matrix has determinant $320 = 2^6 \times 5$ (instead of $256 = 2^8$), whereas the latter matrix has determinant $3712 = 2^7 \times 29$ (instead of $4096 = 2^{12}$). Thus, straightforward application of $\alpha = 2^n$ is no improvement. According to (2), we need a, d, and $b^2 + c^2$ each to be a power of 2. Well, unfortunately there exists no primitive Pythagorean triple $(b, c, 2^m)$.

A possible solution, is sticking to the choice $\alpha = 2.5$, but replacing the rounding step by a 'rounding up' step for the 1.63 entries of the matrix (3), but a 'rounding down' step for the 0.68 entries of the matrix, resulting in

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & -2 \\ 1 & -1 & -1 & 1 \\ 0 & -2 & 2 & 0 \end{pmatrix}, \tag{6}$$

with determinant $32 = 2^5$. This, of course, has the disadvantage of lacking any international normalization. Nevertheless we give here its optimal reversible implementation in Fig. 6. Because it generates no garbage, the same circuit can be operated in both directions, without any garbage uncomputing circuitry.

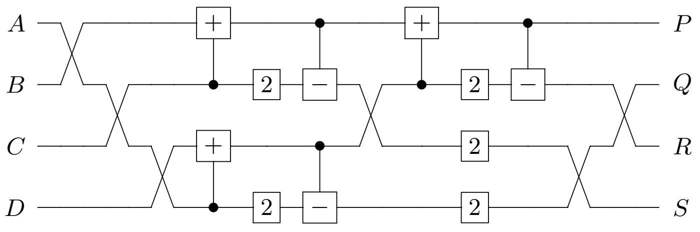


Fig. 6. Optimal lifting scheme for linear transform (6)

Finally, combining $\alpha = 8$ with 'creative rounding' suggests

$$\begin{pmatrix} 4 & 4 & 4 & 4 \\ 5 & 2 & -2 & -5 \\ 4 & -4 & -4 & 4 \\ 1 & -6 & 6 & -1 \end{pmatrix}, \tag{7}$$

with determinant $4096 = 2^{12}$. This solution has the following advantage compared to (6): its four successive rows display the traditional 0, 1, 2, and 3 sign changes. Matrix (7) also is a more accurate approximation of αC, if we use $\sum_{i=1}^{4} \sum_{j=1}^{4} |A_{ij}|$ as norm for a matrix A, according to [17]. Figure 7 shows the corresponding circuit. All scaling factors (i.e. 2, 2, 4, 8, and 32) can be implemented by appropriate bit shifts. All lifting factors (i.e. 5 and 6) can be decomposed into powers of 2 ($5 = 4 + 1$ and $6 = 4 + 2$, respectively) and thus can equally be implemented as bit shifts. From left to right, the circuit performs the transformation (7); from right to left, this same circuit performs the inverse transformation

$$\frac{1}{64}\begin{pmatrix} 4 & 6 & 4 & 2 \\ 4 & 1 & -4 & -5 \\ 4 & -1 & -4 & 5 \\ 4 & -6 & 4 & -2 \end{pmatrix}.$$

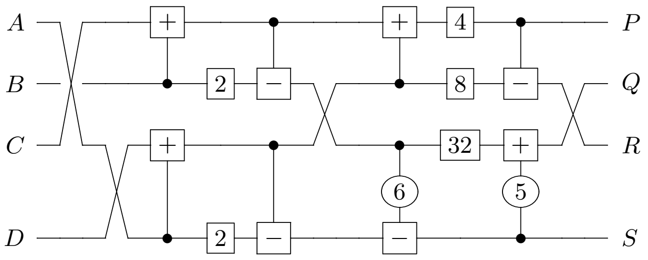


Fig. 7. Optimal lifting scheme for linear transform (7)

6 Conclusion

We have demonstrated that a single digital circuit can both perform a coding and a decoding operation. It suffices that the determinant of the coding matrix is of the form $\pm 2^k$. In analogy with the expression 'perfect scaling coefficient' introduced by Bruekers and van den Enden [15], we call such determinants 'perfect determinants'. They allow the implementation of an integer linear transform without creation of garbage numbers.

Acknowledgment. The authors would like to thank the *Danish Council for Strategic Research* for the support of this work in the framework of the *MicroPower* research project. They are also grateful to the *Invomec* division of *Imec v.z.w.* (Leuven, Belgium) and the *Europractice* consortium for the help with the prototyping of the chip.

References

1. Skoneczny, M., Van Rentergem, Y., De Vos, A.: Reversible Fourier transform chip. In: Proceedings of the 15th International Conference on Mixed Design of Integrated Circuits and Systems, Poznań, pp. 281–286 (2008)
2. Sweldens, W.: The lifting scheme: a custom-design construction of biorthogonal wavelets. Applied and Computational Harmonic Analysis 3, 186–200 (1996)

3. 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)
4. Fredkin, E., Toffoli, T.: Conservative logic. International Journal of Theoretical Physics 21, 219–253 (1982)
5. Vedral, V., Barenco, A., Ekert, A.: Quantum networks for elementary arithmetic operations. Physical Review A 54, 147–153 (1996)
6. Cuccaro, S., Draper, T., Kutin, S., Moulton, D.: A new quantum ripple-carry addition circuit, arXiv:quant-ph/0410184v1
7. Thomsen, M., Axelsen, H.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19, 205–222 (2009)
8. De Vos, A.: Reversible computing. Wiley–VCH, Weinheim (2010)
9. Burignat, S., De Vos, A.: Test of a majority-based reversible (quantum) 4 bit ripple-adder in adiabatic calculation. In: Proceedings of the 18th International Conference on Mixed Design of Integrated Circuits and Systems, Gliwice, pp. 368–373 (2011)
10. De Vos, A.: Reversible computer hardware. Electronic Notes in Theoretical Computer Science 253, 17–22 (2010)
11. Malvar, H., Hallapuro, A., Karczewicz, M., Kerofsky, L.: Low-complexity transform and quantization in H.264/AVC. IEEE Transactions on Circuits and Systems for Video Technology 13, 598–603 (2003)
12. Hadamard transform. Wikipedia (2012)
13. De Vos, A., De Baerdemacker, S.: Decomposition of a linear reversible computer: digital versus analog. International Journal of Unconventional Computing 6, 239–263 (2010)
14. De Vos, A., Burignat, S., Thomsen, M.: Reversible implementation of a discrete linear transformation. International Journal of Multiple-valued Logic and Soft Computing 18, 25–35 (2012)
15. Bruekers, F., van den Enden, A.: New networks for perfect inversion and perfect reconstruction. IEEE Journal on Selected Areas in Communications 10, 129–137 (1992)
16. Yokoyama, T., Axelsen, H., Glück, R.: Optimizing reversible simulation of injective functions. International Journal of Multiple-valued Logic and Soft Computing 18, 5–24 (2012)
17. Baker, A.: Matrix Groups. Springer, London (2002)

Garbage-Free Reversible Integer Multiplication with Constants of the Form $2^k \pm 2^l \pm 1$

Holger Bock Axelsen and Michael Kirkedal Thomsen

DIKU, Department of Computer Science, University of Copenhagen
{funkstar,shapper}@diku.dk

Abstract. Multiplication of integers is non-injective and, thus, requires garbage lines for any reversible logic implementation. However, multiplying with a fixed constant *is* injective, and should therefore be implementable in reversible logic *without* garbage. Despite this, the only reported circuits for constant multiplication without garbage are restricted to powers of 2, *i.e.*, the multiplication is a simple bit-shift.

Here, we show how to generate a garbage-free linear-depth reversible logic circuit for multiplying an input integer with a constant of the form $2^k \pm 1$ or $2^k \pm 2^l \pm 1$, by building on a simple strength reduction to addition. Using several such circuits in sequence allows us to support a greater variety of constants. This enables wider use of constant multiplication in garbage-free reversible circuits than was previously possible.

Keywords: Reversible circuits, logic design, constant multipliers, garbage-free.

1 Introduction

Efficient implementations of binary multiplication is important to perform fast computations, and designs using conventional boolean logic have been widely researched. This importance has also led to significant research into the design of multiplication circuits using reversible logic. As examples, we mention the reversible Karatsuba algorithm using Bennett's method [7], and an implementation based on logic synthesis [9]. These, and other implementations, have the drawback of producing significant amounts of *garbage* lines.

Multiplication of two integer variables is non-injective, and implementations in reversible logic without additional garbage lines are in general not possible. However, for many applications fully general multiplication is not needed and one can instead rely on specialized *constant multiplication* circuits. Examples include *twiddle factor* multiplications in the fast Fourier transform and other loss-less transformations, such as the specific wavelet used for H.264 video encoding [2,5].

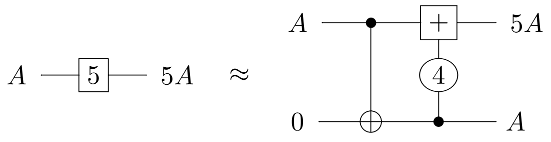
Now, constant multiplication *is* injective. This implies that garbage-free circuits for constant multiplication are possible (with an expanded bit-wise representation of the result) but the known reversible logic circuits for multiplication with constants either leave garbage, or are simple bit-shifts. This prohibits wider use of constant multiplication in reversible circuits aiming for garbage-free implementations of injective functions.

Here, we show the design of garbage-free circuits for multiplying with constants of the form $2^k \pm 1$ (Sec. 2). The proposed family of designs follows the backbone structure of reversible V-shape ripple-carry adders, which are linear depth in the size of the output. The circuits require relatively few ancillae, so it is a 'narrow' design as well. Generalizing this idea to multioperand adders, we show a naïve design for a three operand adder, and specialize it to multiplication with constants of form $2^k \pm 2^l \pm 1$ (Sec. 3), again fully garbage-free. These improvements enable much wider use of constant multiplication in garbage-free reversible circuits than was previously possible: composition of these circuits, their inverses, and the use of bit shifts, enables us to implement many constants that are *not* of the form $2^k \pm 1$ or $2^k \pm 2^l \pm 1$ as well.

2 Garbage-Free Reversible Constant Multiplication

A main reason for the garbage generation of reversible multiplication circuits seems to be that previous designs have not considered how the non-injective problem of multiplication may be embedded in injective functions, and how those might be realized without garbage. (See [1] for a theory of garbage-free reversible computing.) One widely known class of circuit designs has already benefited from looking at the problem in this manner: we refer to binary addition, where the novel V-shaped adder approach by Vedral *et al.* [14] led the way to a garbage-free implementation of the injective embedding $+(A, B) = (A + B, A)$. In this first step towards garbage-free multiplication we work towards multiplication of an n-bit number with a $(k + 1)$-bit *constant*, giving an $(n + k + 1)$-bit result.

Constant multiplication is simpler than general multiplication, but is still useful in many applications. Our particular motivation for this work is a reversible version of the H.264 wavelet [2,5]. This was implemented with a *lifting scheme* [4], which leads to a scaling factor of 5 at a point in the circuit. In [5] this multiplication with 5 was implemented as a strength reduction, $5A = 4A+A = A \ll 2+A$, leaving a copy of the multiplicand A as garbage (figure adapted from [5]):



That is, the scaling factor is converted to a lifting. Thus, even though the overall algorithm is injective (the wavelet matrix is invertible), the scaling factors complicated the design sufficiently such that garbage lines were deemed necessary.

2.1 The Idea

We initially restrict our scope of multiplication constants to those in range of the strength reduction outlined above: the input A is copied, one copy shifted some number of bits (multiplied by a power of 2) and added to the other copy. We thus start by considering the case of multiplying A by $2^k + 1$. (As we shall discuss in Sec. 2.4, this class of constants is more versatile than it may initially appear.)

The addition on which the strength reduction is based has the following form.

$$\begin{array}{rccc} & 0_k \dots 0_1 & \boxed{A_{n-1} \dots A_1 A_0} \\ + & \boxed{A_{n-1} \dots A_1 A_0} & 0_k \dots 0_1 \\ \hline = & M_{n+k} \quad \dots\dots\dots\dots & M_1 M_0 \end{array}$$

From this we can identify a number of dependencies between the bitwise representation of the product M and the multiplicand A. Concretely,

- the k least significant bits of M and A are identical,
- the $k+1$ most significant bits of M are the k most significant bits of A, incremented with the nth carry bit (C_n),
- the remaining $n-2k$ middle bits of M are given by $M_i = C_i \oplus A_i \oplus A_{i-k}$.

The underlying idea of our reversible circuit design is to leverage these dependencies to clear the extraneous copy of A left over by the reversible addition.

2.2 The Implementation

We base the implementation on the V-shaped ripple-carry adder introduced by Vedral *et al.* [14], optimized first by Cuccaro *et al.* [3] (CDKM) and later by Van Rentergem and De Vos [13]. We use the latter design, but our approach can also be adapted to the CDKM adder. In the following we shall assume some familiarity with these circuit designs. (See *e.g.* [11] for an in-depth exposition.)

The method we shall use is to clear the extraneous multiplicand A "on-the-fly" while computing the sum M. This is mainly done by extending the unmajority-and-sum (*ums*) part of the upwards ripple in the circuit, see Fig. 1.

The extension is based on the fact that the majority (*maj*) downwards ripple generates $A_i \oplus A_{i-k}$ as an intermediate product. That is, for the ith bit-slice ($n > i \geq k$) we have that

$$maj(C_i, A_{i-k}, A_i) = (T_i, C_{i+1}, A_i \oplus A_{i-k})$$

and

$$ums(T_i, C_{i+1}, A_i \oplus A_{i-k}) = (C_i, A_{i-k}, M_i)$$

giving the result M_i, but also leaving the value A_{i-k} that we need to clear.[1] Now, rather than leaving this as garbage, we instead *propagate* the A_{i-k} upwards in the circuit to the $(i-k)$-th bit-slice, to be zero-cleared during the sum computation. Thus, in the ith bit-slice ($n-k > i \geq k$) we will have an extra line with the value A_i propagated from the $(i+k)$-th bit-slice below, which is zero-cleared as follows.

[1] The exact form of the intermediate product T_i is unimportant, and can be ignored.

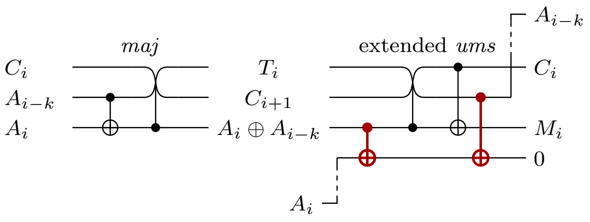


Fig. 1. The ith bit-slice of the ripple-down *maj* circuit and ripple-up extended *ums* circuit used in the implementation. A_i is propagated from the $(i+k)$-th bit-slice below, with the added gates to clear it marked with **bold**. A_{i-k} is propagated to the $(i-k)$-th bit-slice above.

First, before we apply the *ums* circuit, we exclusive-or the intermediate product $A_i \oplus A_{i-k}$ onto the propagated A_i line using a Feynman gate.[2]

$$\mathsf{Feyn}_{3,4}(T_i, C_{i+1}, A_i \oplus A_{i-k}, A_i) = (T_i, C_{i+1}, A_i \oplus A_{i-k}, A_{i-k}) \ .$$

This leaves A_{i-k} on the propagated line. We then apply the *ums* circuit to get

$$ums_{1,2,3}(T_i, C_{i+1}, A_i \oplus A_{i-k}, A_{i-k}) = (C_i, A_{i-k}, M_i, A_{i-k}) \ ,$$

whereupon we can clear one A_{i-k} line with a second Feyman gate,

$$\mathsf{Feyn}_{2,4}(C_i, A_{i-k}, M_i, A_{i-k}) = (C_i, A_{i-k}, M_i, 0) \ .$$

The total effect is thus to *clear* the line containing A_i that was propagated from the $(i+k)$-th bit-slice, using just two Feynman gates. Similarly, the leftover copy of A_{i-k} is propagated to the $(i-k)$-th bit-slice where it will be cleared using the same procedure. The resulting extended *ums* circuit is shown in Fig. 1.

Using this extended *ums* circuit we can now implement a complete garbage-free constant multiplier, exploiting the dependencies identified in Sec. 2.1. The general structure of the $2^k + 1$ multiplier is shown in Fig. 2. First, we copy the $n-k$ most significant bits of A and shift them and the k least significant bits of A (the uncopied part) k bits downward. Second, we perform an $(n-k)$-bit downwards ripple of majority circuits (see Fig. 1), calculating the intermediate exclusive-or products and the carry C_n. For the remaining $k+1$ most significant bits of the result we need not calculate a sum, but instead have only to increment the input if the carry C_n is set. After this conditional incrementation, we perform an upwards ripple to calculate the sum using our extended unmajority-and-sum circuit. Now, for the first k bits of the upwards ripple there are no propagated values to clear and it suffices to use the standard *ums* circuit. Only for the next $(n-2k)$ bit-slices do we have to use the extended circuit. This leaves k propagated lines that are not cleared, but these are exactly the k least significant bits of the result. The total effect is to calculate the $(n+k+1)$-bit product without *any* garbage.

[2] Here, the subscripts refer to the entries on which the gates are applied.

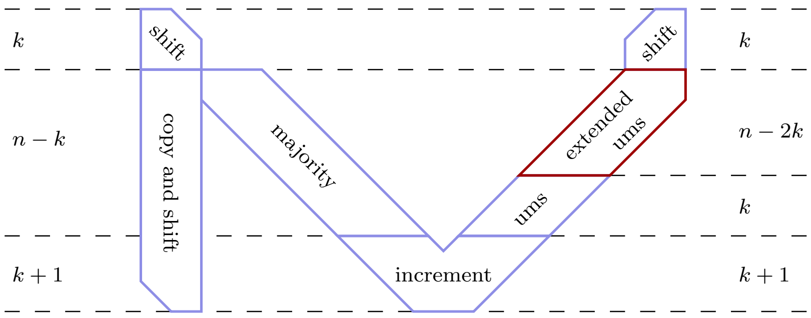


Fig. 2. Overview of the $(n + k + 1)$-bit multiplication circuit

As a full example, Fig. 8 shows the complete circuit for multiplying a 10-bit value with the constant 9 $(= 2^3 + 1)$ resulting in a 14-bit product. The circuit uses 7 $(= n - k + 1)$ ancilla bits.

2.3 Extending to Multiplication by $2^k - 1$

We can extend the class of multiplication constants to include those of form $2^k - 1$. As before, multiplication with such constants can be implemented by a strength reduction $(2^k - 1) \cdot A = 2^k \cdot A - A = A \ll k - A$, with the sole difference being that we now have to perform a subtraction rather than an addition.

Fortunately, for two's complement numbers we can implement subtraction by addition. This allows us to use the $2^k + 1$ multiplier design above almost directly to implement multiplication by $2^k - 1$. Concretely, we make use of the fact that $B - A = \overline{\overline{B} + A}$ (where $\overline{B}$ denotes the bitwise negation of B), as previously used in the design of subtraction in a reversible ALU [12]. For our $2^k - 1$ multiplier circuit $B = 2^k \cdot A$.

How must we modify the multiplier circuit to support this? First, we note that B, the bit-shifted copy of A, is bitwise negated. In particular, this means that the k least significant bits of $\overline{B}$ are no longer 0, but 1. Therefore, we must explicitly perform the addition on the k least significant bit-slices. Second, for the intermediate $(n - k)$ bit-slices, the only major changes are that the input line of the bit-shifted copy is negated, and that the final sum must be negated as well. Third, the incrementation circuit for the most significant $k + 1$ bits does not need modification, save for negating the result.

For the intermediate bits, the ith bit-slice is shown in Fig. 3 . Note that the change compared to the $2^k + 1$ multiplier (Fig. 1) is limited to just two negations, and that these can be performed in parallel for all bit-slices.

For the k least significant bit-slices, the $2^k - 1$ multiplier circuit looks somewhat different to its $2^k + 1$ counterpart. Instead of a simple shift, we here have a ripple down and up, implementing addition. However, given that the k least bits of $\overline{B}$ are all 1, we can specialize the *maj* and *ums* circuits to this particularly simple addition. Furthermore, one of the intermediate products is exactly $\overline{A_i}$,

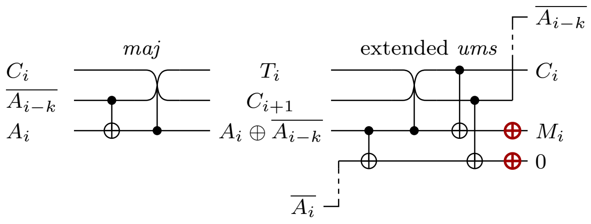


Fig. 3. The ith bit-slice of the $2^k - 1$ multiplier, for an intermediate bit-slice. Just two NOT-gates are added (**bold**) compared to the $2^k + 1$ multiplier.

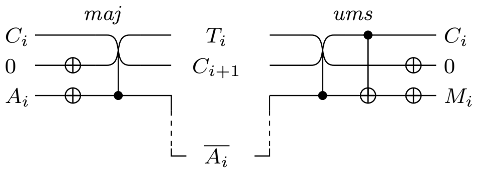


Fig. 4. The ith bit-slice of the $2^k - 1$ multiplier for the k least significant bits. The $\overline{A_i}$ line is propagated to the $(i + k)$-th bit-slice below and back.

which we then can propagate down to the $(i + k)$-th bit-slice for use as $\overline{B_{i+k}}$. Finally, in the upwards ripple this line is propagated back to the i-th bit-slice, to be used for the final sum calculation. Fig. 4 shows the simple circuit that implements this.

Thus, with only very limited changes to the design, we can support multiplication with constants of form $2^k \pm 1$.

2.4 Cascades and Limitations

This class of multiplication constants is more versatile than it may initially seem. By using shifts (multiplications by 2) and repeated strength reductions, many other constants can be implemented: multiplication by 21 (which is *not* of the form $2^k \pm 1$) can obviously be implemented by multiplication by 3 followed by multiplication by 7, because scaling factors multiply.

Perhaps more surprisingly, multiplication by 11 can be implemented by multiplication by 33 followed by the *inverse* circuit of multiplication by 3. The inverse circuit implements division by 3 for numbers exactly divisible by 3, such as $33 \cdot A$. This is possible only because the multiplication circuit is garbage-free, and would *not* be possible if the circuit produced garbage. Thus, both division and multiplication of constants of form $2^k \pm 1$ are available for cascading.

Unfortunately, this is still not sufficient to support multiplication with *all* integer constants, as proven recently by Rotenberg [10]. As examples, multiplication by 23 or 67 is not in range with this class of multiplication constants.

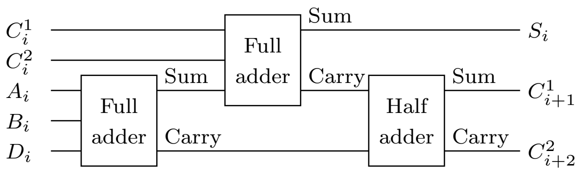


Fig. 5. Bit-slice of a conventional 5:3 compressor

3 Multioperand Adders and Multiplication

As it was possible to show that the proposed multiplier design is *not* yet general enough to implement multiplication with all constants, one might be tempted to believe that our design approach is not strong enough. However, the idea of using strength-reduction and clearing of the copied operand is fundamentally sound. The problem is rather that our 2^k+1 multiplier is based on a two-operand adder, which means that it can only be used to multiply with constants whose binary representation has exactly two bits set.

For conventional (irreversible) multipliers this does not pose a problem, in that a cascade of two-operand adders can be used to implement any (constant) multiplication. However, the same idea is not directly applicable in a garbage-free reversible circuit, as it appears difficult to remove all the extraneous copies of the multiplicand.

All is not lost: instead, our strategy shall be to design a (garbage-free) reversible *multi-operand* V-shape adder and use the same method to clear the extra copies of the multiplicand as in the simple 2^k+1 multiplier. In this section we shall first describe the design of a binary three-operand adder. Afterwards, we show how the adder can be extended to 2^k+2^l+1 multiplication, and finally how to obtain all constants of form $2^k \pm 2^l \pm 1$.

3.1 A Three-Operand V-Shape Adder

Each bit-slice of the three-operand adder will take three independent inputs (A_i, B_i, and D_i[3]) and we therefore need two bits to represent the carry; three input bits and (at least) one carry yields (at least) a sum of four bits, for which we need two bits to represent the carry-outs. This means that each bit-slice has two carry-ins as well. To ease the design, we shall use binary representation for the two carry-outs, which means that one carry will have a weight of 2 (we call this C_i^1) as a normal carry and the other (called C_i^2) will have a weight of 4. This idea is not novel, and if we also include the calculation of the sum, it is simply the definition of a 5:3 (lossy) compressor: encode the sum of 5 bits as a 3-bit output. (Similarly, a full-adder is a 3:2 compressor.)

[3] We use D to denote the third operand as C is already used for the carries.

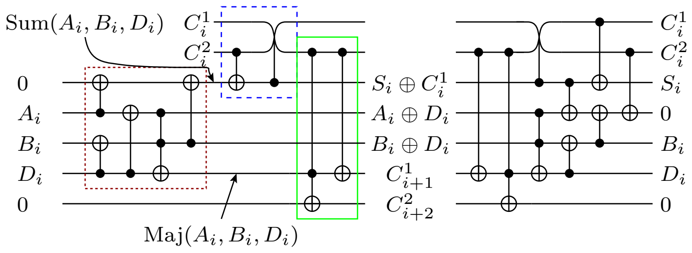


Fig. 6. The ith bit-slice of the three-operand V-shape adder design

The conventional logic design of a 5:3 compressor consists of two full-adders and one half-adder, connected as shown in Fig. 5. We can define the functionality of this as follows:

$$
\begin{aligned}
M_{abd} &= \mathrm{Maj}(A_i, B_i, D_i)\\
S_{abd} &= A_i \oplus B_i \oplus D_i\\
S_i &= S_{abd} \oplus C_i^1 \oplus C_i^2\\
C_{i+1}^1 &= M_{abd} \oplus \mathrm{Maj}(S_{abd}, C_i^1, C_i^2)\\
C_{i+2}^2 &= M_{abd} \wedge \mathrm{Maj}(S_{abd}, C_i^1, C_i^2)
\end{aligned}
$$

where Maj() is the majority function. Obviously, the conventional compressor design is not reversible. Therefore, we redesign the adder structure to follow the design of the V-shape adder. Functionally, the adder must compute $+(A, B, D) \mapsto (A + B + D, B, D)$, and must do so without generating garbage.

We first identify the number of lines we need for each bit-slice. With two carry-ins and three operands there are many possibilities for there to be 2 or 3 bits set in the input: $\binom{5}{2} + \binom{5}{3} = 20$ lines out of the $5^2 = 32$ in the truthtable for the circuit. This means that the carry-out sequence "01" occurs 20 times as well, but the remaining three lines in the circuit only allows for $2^3 = 8$ possibilities if the circuit is to be 5×5 and reversible. Thus, we cannot compute the carry-outs without additional lines; in this case we need two extra.

The great fundamental insight of the V-shape adder design [14] is that the sum and the carry should not be calculated simultaneously. We can use exactly the same approach for our three-operand adder. First, we have a downwards ripple that only calculates the carries and then we have an upwards ripple which uncomputes these carries and calculates the final sum. By using the V-shape adder design, the extra lines will not be problematic (*i.e.*, they will not end up with garbage values) because *by design* they will always be uncomputed as well. Thus, the additional lines are ancillae, rather than garbage.

A bit-slice in a straightforward design of the adder is shown in Fig. 6. Starting from the left, the first (red, dotted) box implements the first full-adder from Fig. 5 using one of the ancillae lines. This sub-circuit can be performed in parallel in all bit-slices as it only involves the three operands and not the carries.

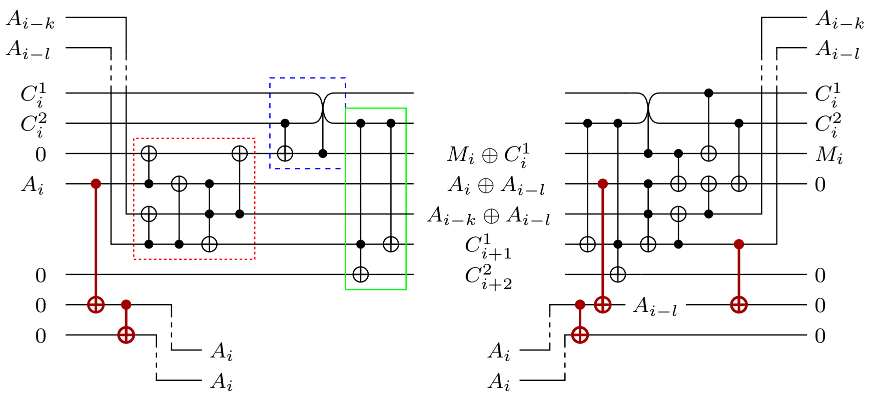


Fig. 7. The ith intermediate bit-slice of the $2^k + 2^l + 1$ multiplier. In the downwards ripple, one of the two copied A_i lines is propagated to the $(i + l)$-th bit-slice, and the other to the $(i + k)$-th bit-slice, and return in the upwards ripple. Likewise, A_{i-k} and A_{i-l} are propagated to this bit-slice from the $(i - k)$-th and $(i - l)$-th bit-slice, and return there in the upwards ripple. New gates are shown in **bold red**.

The second (blue, dashed) box implements a majority circuit. The reason that this is not a second full-adder (as in Fig. 5) is that we are not yet interested in calculating the final sum and, thus, only a majority circuit is needed. The third (green, lined) box implements the half-adder using the second ancilla bit. This results in the two carry-bits where C_{i+1}^1 is rippled down to the next bit-slice, while C_{i+2}^2 is rippled down two bit-slices. The second part of Fig. 6 shows the uncomputation-and-sum circuit. This circuit has been optimized slightly for a more compact representation, but is admittedly naïve. We will leave circuit optimization for future work – possibly for one of the many synthesis algorithms for reversible logic, *cf.* [15].[4]

We remark that this three-operand adder circuit has worse characteristics than two sequential two-operand adders, both in terms of delay and gates count. While this does not come as a surprise, as the conventional 5:3 compressor is also larger than two full-adders, it is not in itself a particularly satisfying implementation of three-operand addition. However, unlike sequential two-operand adders, the design allows for easy specialization to constant multipliers, as we shall now see.

3.2 Specializing to Multiplication by $2^k + 2^l + 1$

Now that we have the three-operand adder, we can specialize it to a design that can multiply with constants of the form $2^k + 2^l + 1$, where $k > l > 0$. The method used to extend the adder is the exact same as we used for the $(2^k + 1)$-multiplier, and again, very little extra logic is needed.

[4] Mogensen [8] has already proposed an improved bit-slice design.

Figure 7 shows a bit-slice of the resulting circuit. We first create *two* copies of A_i. The first of these is propagated l bits downwards, while the second is propagated k bits. In the upwards ripple these return, and must be zero-cleared. One copy of A_i is easily removed by the other copy. Then, we use the intermediate product $A_i \oplus A_{i-l}$ to transform A_i to A_{i-l}. This A_{i-l} line can be cleared using the copy of A_{i-l} that was earlier propagated to this bit-slice as an input operand. In total, the only extension to the clearing method from the $2^k + 1$ multiplier circuit is to add two Feynman gates that, respectively, create and remove an extra copy of A_i.

This also means that we can now multiply with all constants of form 2^k+2^l+1. These include $67 = 2^6 + 2^1 + 1$, which was not possible with any combination of $2^k \pm 1$ multipliers.

3.3 Extending to Multiplication by $2^k \pm 2^l \pm 1$

Finally, as with the first multiplier, we shall now extend the design such that it can multiply with any constant of the form $2^k \pm 2^l \pm 1$. Again, we can employ essentially the same method. However, in this case we now have three operands, which gives more possibilities for the signs of the operands. The three remaining possibilities are captured with the following formulas (where we use the fact that for two's complement numbers $-A = \overline{A} + 1$):

$$A + B - D = A + B + \overline{D} + 1$$
$$A - B + D = A + \overline{B} + D + 1$$
$$A - B - D = \overline{\overline{A} + B + D}$$

For the two first equations we can exploit that we have a carry-in line that we can set to 1 at will. The third equation is very similar to the formula we had for multiplication with $2^k - 1$, and can be similarly implemented. We shall not show the resulting circuits, but rather note that this extension makes it possible to multiply by $23 = 2^5 - 2^3 - 1$, yet another constant which was not previously in range.

4 Conclusion and Future Work

In this paper we have shown a garbage-free linear-depth reversible circuit family for multiplying an n-bit number with any constant of the form $2^k \pm 1$ or $2^k \pm 2^l \pm 1$, yielding an $(n + k + 1)$-bit product.

This allows garbage-free multiplication with many more constants than have previously been reported. As a direct application, the optimal lifting scheme for the H.264 discrete cosine transform (previously complicated by a scaling factor of 5) can now be implemented directly in reversible logic, without having to resort to garbage or having to approximate the wavelet [2].

The $2^k \pm 1$ design in particular compares favorably to the backbone V-shape ripple-carry adder on which it is based: it uses roughly $3n - 5k$ Feynman gates more, but is generally just 1 gate slower. However, the $2^k + 2^l + 1$ design is

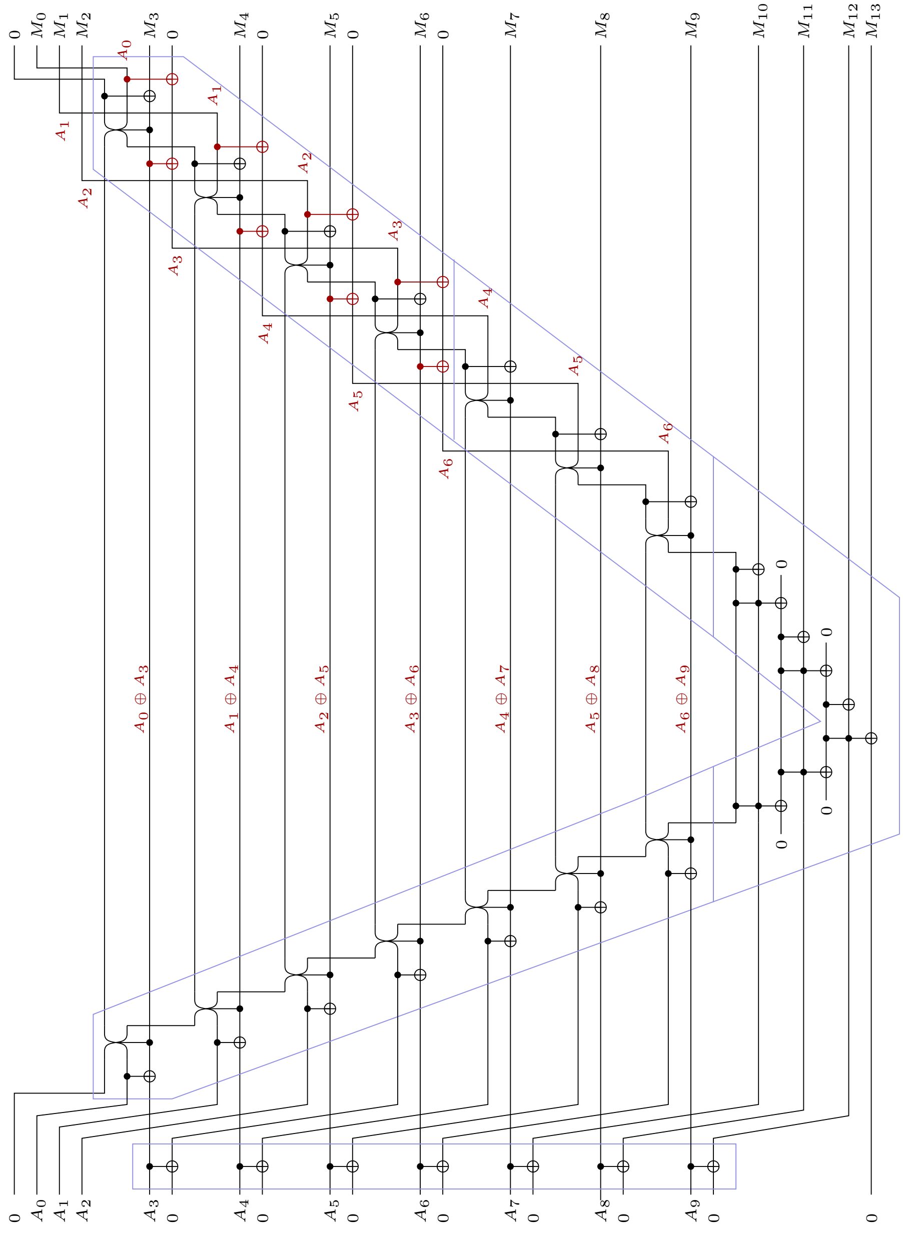


Fig. 8. Multiplying the 10-bit input A by the constant 9, yielding 14-bit product M

less efficient, and although the generalization to multi-operand adders suggests a fairly straightforward route to multiplication with arbitrary constants, the resulting circuits will likely not be a particularly elegant solution to this problem.

Instead, general constant multiplications may be possible using a *nested* V-shape approach as in [12]. It is also open whether strength reduction can work for sub-linear depth adders [6,11]. Finally, it would be interesting to examine if the approach can somehow be extended to fully general reversible multiplication.

References

1. Axelsen, H.B., Glück, R.: What Do Reversible Programs Compute? In: Hofmann, M. (ed.) FOSSACS 2011. LNCS, vol. 6604, pp. 42–56. Springer, Heidelberg (2011)
2. Burignat, S., Vermeirsch, K., De Vos, A., Thomsen, M.K.: Garbageless Reversible Implementation of Integer Linear Transformations. In: Glück, R., Yokoyama, T. (eds.) RC 2012. LNCS, vol. 7581, pp. 160–170. Springer, Heidelberg (2013)
3. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit. arXiv:quant-ph/0410184v1 (2005)
4. Daubechies, I., Sweldens, W.: Factoring wavelet transforms into lifting steps. Journal of Fourier Analysis and Applications 4(3), 247–269 (1998)
5. De Vos, A., Burignat, S., Thomsen, M.K.: Reversible implementation of a discrete integer linear transform. J. Mult.-Val. Log. S. 18(1), 25–35 (2012)
6. Draper, T.G., Kutin, S.A., Rains, E.M., Svore, K.M.: A logarithmic-depth quantum carry-lookahead adder. arXiv:quant-ph/0406142v1 (2004)
7. Kowada, L.A.B., Portugal, R., de Figueiredo, C.M.H.: Reversible Karatsuba's algorithm. J. Universal Computer Science 12(5), 499–511 (2006)
8. Mogensen, T.Æ.: Private communication (2012)
9. Offermann, S., Wille, R., Dueck, G.W., Drechsler, R.: Synthesizing multiplier in reversible logic. In: 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pp. 335–340. IEEE (2010)
10. Rotenberg, E.: Mersennary numbers, University of Copenhagen (report in preparation, 2012)
11. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters 19(1), 205–222 (2009)
12. Thomsen, M.K., Glück, R., Axelsen, H.B.: Reversible arithmetic logic unit for quantum arithmetic. J. Phys. A: Math. Theor. 43(38), 382002 (2010)
13. Van Rentergem, Y., De Vos, A.: Optimal design of a reversible full adder. International Journal of Unconventional Computing 1(4), 339–355 (2005)
14. Vedral, V., Barenco, A., Ekert, A.: Quantum networks for elementary arithmetic operations. Physical Review A 54(1), 147–153 (1996)
15. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer Science (2010)

Property Checking of Quantum Circuits Using Quantum Multiple-Valued Decision Diagrams

Julia Seiter[1], Mathias Soeken[1,2], Robert Wille[1], and Rolf Drechsler[1,2]

[1] Institute of Computer Science, University of Bremen
Group of Computer Architecture, D-28359 Bremen, Germany
`{jseiter,msoeken,rwille,drechsle}@informatik.uni-bremen.de`
[2] Cyber-Physical Systems
DFKI GmbH, D-28359 Bremen, Germany
`rolf.drechsler@dfki.de`

Abstract. For the validation and verification of quantum circuits mainly techniques based on simulation are applied. Although lots of effort has been put into the improvement of these techniques, ensuring the correctness still requires an exhaustive consideration of all input vectors. As a result, these techniques are particularly insufficient to prove a circuit to be error free.

As an alternative, we present a symbolic formal verification method that is based on Quantum Multiple-Valued Decision Diagrams (QMDDs), a data-structure allowing for a compact representation of quantum circuits. As a result, using QMDDs it is possible to check the correctness of a circuit without exhaustively considering all input patterns.

1 Introduction

Quantum computation [1] has received significant attention in recent years. Using quantum circuits many important problems such as factorization or database search can be solved quadratically or even exponentially faster in comparison to conventional technologies. As a result, much effort has been spent in the past on how to design the corresponding circuit structures. In particular synthesis received much attention (see e.g. [2–6]). But besides realizing quantum circuits for a given problem, verification and validation is an essential step that ensures whether obtained designs realize the desired functionality or not.

Considering conventional circuit design, verification has become one of the most important steps in the design flow. As a result, very powerful approaches have been developed in this domain, ranging from simulative verification (see e.g. [7–10]) to formal equivalence checking (see e.g. [11, 12]) and property checking (see e.g. [13, 14]).

For quantum computation, verification is still at the beginning. Even if first approaches in this area exist (a brief outline is provided in Sect. 3.2), they are mainly based on simulation, i.e. ensuring the correctness still requires an exhaustive consideration of the input vectors. As a result, these techniques are particularly insufficient to prove a circuit to be error-free.

In this work, we present an alternative solution to the property checking problem of quantum circuits which makes use of symbolic formal verification. More precisely, for this task we consider *Quantum Multiple-Valued Decision Diagrams* (QMDDs, [15]), which is a data-structure that allows for a compact representation of quantum circuits. Using QMDDs both a given quantum circuit and the property to be verified can be efficiently represented. Then, checking whether the circuit satisfies the considered property or not can be conducted on this data-structure. The properties themselves are thereby provided by means of a combinatorial and LTL-like language.

Experiments show that, in comparison to state-of-the-art simulation methods, the proposed approach is more robust. While simulation-based approaches work faster for failing properties (where the simulation can immediately be terminated once a counter-example is obtained), QMDDs clearly outperform simulation for holding properties where all possible input patterns need to be traversed exhaustively.

The remainder of this paper is structured as follows. The next section briefly reviews the basics on quantum circuits and the QMDD data-structure. Section 3 formally defines the considered problem, discusses related work, and introduces the general idea of the proposed solution. Afterwards, implementation details are described in Sect. 4. A summary of the experimental evaluation and conclusions are provided by means of Sect. 5 and Sect. 6, respectively.

2 Preliminaries

In order to keep the paper self-contained, this section reviews the basics on quantum circuits and the QMDD data-structure applied in this work. Due to page limitations the following descriptions are kept brief. We refer the reader to [1] and [15] for a more detailed treatment of quantum circuits and QMDDs, respectively.

2.1 Quantum Circuits

In quantum computation [1], *qubits* are the elementary information elements. A qubit is usually denoted by its state $|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$ which is a *superposition* of the basis states $|0\rangle$ and $|1\rangle$ with α and β being *amplitudes* such that $|a|^2 + |b|^2 = 1$. Qubits can be composed in terms of quantum registers $|\varphi_1 \dots \varphi_n\rangle = \sum_{i=0}^{2^n-1} \alpha_i|i\rangle$, where $|i\rangle$ denotes the conventional state on n bits. Quantum registers are elements in the 2^n-dimensional complex Hilbert space $\mathcal{H}$. Quantum computation can be performed using unitary matrices that are closed under $\mathcal{H}$.

Established quantum operations include e.g. $\sigma_x = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$ (also known as Pauli-X or NOT operation) and $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ (also known as Hadamard operation). The first operation interchanges the amplitudes of a quantum state whereas the latter operation can be used to bring a conventional state into a superposition, e.g. $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. While these operations act on single

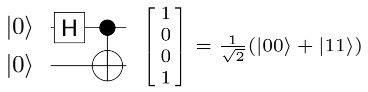

$|0\rangle$
$|0\rangle$
H
$\begin{bmatrix}1\\0\\0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$

Fig. 1. Circuit that entangles two qubits

qubits only, they can be extended to also act on quantum registers. This can either be accomplished by the parallel composition using the Kronecker product ($\otimes$) or by controlling the single-qubit operation by one further qubit. As an example, a Pauli-X operation controlled by another qubit can be represented by $\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$. Such an operation is called *controlled NOT* (CNOT).

Quantum circuits allow for a graphical representation of the composition of several quantum operations. To this end, a circuit line is drawn horizontally for each qubit. The quantum gates that represent quantum operations are drawn as a cascade from left to right. Control lines are denoted using ● while a quantum operation U is drawn using a rectangle labeled U. An exception is the Pauli-X operation that is denoted using $\bigoplus$.

Example 1. Figure 1 shows a circuit that entangles two qubits. The first gate is a Hadamard operation on the first qubit while the second operation is a CNOT. The overall quantum operation of the circuit is CNOT $\cdot (H \otimes I)$ where $I = \begin{bmatrix}1&0\\0&1\end{bmatrix}$ is the 2×2 identity matrix.

2.2 Quantum Multiple-Valued Decision Diagrams

A *Quantum Multiple-valued Decision Diagram* (QMDD, [15]) is a canonical representation of a unitary matrix M and, thus, also of a quantum circuit. It is a directed acyclic graph with one root node and two terminal nodes, labeled $\boxed{0}$ and $\boxed{1}$.

Each node v in a QMDD represents a submatrix of M. In order to build a QMDD, M is divided into four submatrices. The matrix M itself is represented by the root node v_1. Each submatrix is represented by a child of v_1 so that each node in the QMDD has four child nodes. This is repeated recursively for each submatrix until they are reduced to size 1×1. Submatrices of this size are represented accordingly by one of the terminal nodes [15].

Figure 2(a) illustrates this principle by decomposing an arbitrary matrix M. On top of the matrix M, the input patterns are denoted. The corresponding output patterns are denoted to the left-hand side of M. As can be seen, each of the submatrices of M represents an input/output assignment of the very first variable x_1. For example, the submatrix M_1 represents the mapping of x_1 from 0 to 0. Figure 2(b) shows the respective submatrices represented in terms of successors of the root node. The submatrices of M are assigned to the child nodes as follows.

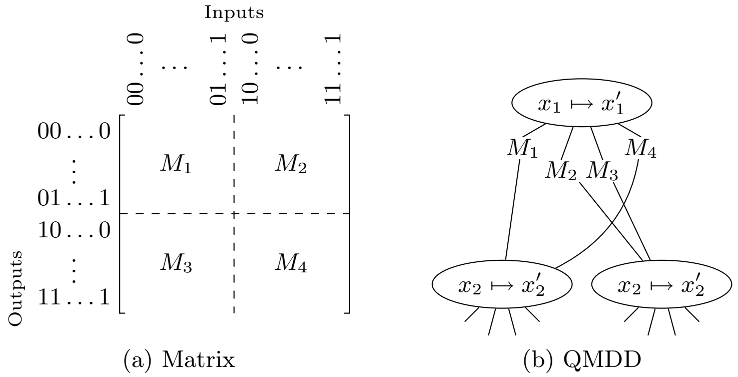


Fig. 2. Matrix with input output mappings and QMDD

– The upper left submatrix M_1 is the first child and represents the mapping $0 \mapsto 0$ of the input x_1 to the output x_1'.
– The upper right submatrix M_2 is the second child and represents the mapping $1 \mapsto 0$ of the input x_1 to the output x_1'.
– The lower left submatrix M_3 is the third child and represents the mapping $0 \mapsto 1$ of the input x_1 to the output x_1'.
– The lower right submatrix M_4 is the fourth child and represents the mapping $1 \mapsto 1$ of the input x_1 to the output x_1'.

In a QMDD, each edge is labeled with a weight. If a submatrix consists of 0 or 1 entries, the respective edge is annotated with the weight 1. Submatrices consisting only of 0-entries are represented by edges leading directly to $\boxed{0}$. In order to keep the QMDD readable, these edges are drawn by a 0-stub. Also, the weight 1 is often omitted as 1 defines the default weight. In case of a submatrix containing complex-valued entries, the respective edge is annotated with a complex-valued weight. More precisely, if all entries have the value w or are a multiple of w, then the edge is labeled with the weight w. The final value of an entry in the matrix is then computed by multiplying all the weights from the root node to the terminal.

These concepts are illustrated in the following example:

Example 2. Figure 3 shows a QMDD representing the circuit from Fig. 1. The corresponding matrix M is

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \end{bmatrix}.$$

By dividing M into four submatrices, the child nodes of the root node can be determined. The first and second submatrix contain the same entries and, thus, are represented by a single shared node. The third and fourth submatrix contain entries of the same value but with different signs. Consequently, they

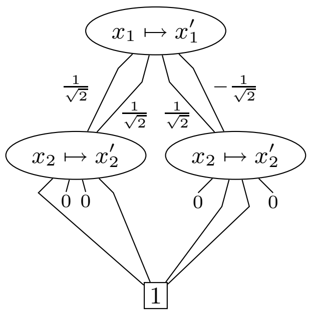


Fig. 3. QMDD representing the circuit from Fig. 1

are represented by the same node, too, but the edges leading to this node are annotated with different weights. The edges leading from the child nodes to the terminals can be determined easily by division of the submatrices. As described above, the edge weight 1 is omitted as well as the terminal $\boxed{0}$ and the edges leading to it. Instead, they lead to a 0-stub.

3 Property Checking of Quantum Circuits

This section formally defines the problem considered in this work and discusses previously introduced approaches which address this issue. Afterwards, the general idea of our solution is proposed.

3.1 Problem Formulation

In this work, we consider property checking of quantum circuits. Property checking is applied during the hardware design phase in order to check whether a designed circuit in fact satisfies the specification or not, i.e. whether the circuit has been designed correctly or not. Since a specification may be very complex, usually several properties are defined which the circuit has to satisfy. Then, these properties are individually checked. The properties describe the intended relation between the inputs and the outputs of the design or they describe requirements which have to be satisfied. If the complete specification is covered by means of properties and, additionally, a realized circuit satisfies all of them, then it has been proven that the circuit was designed correctly. Otherwise, the circuit contains design errors and has to be revised.

Formally a circuit G is considered which realizes the function $f_G : I\!B^n \rightarrow I\!B^n$ with inputs $x_1, \ldots, x_n$ and outputs $x_1', \ldots, x_n'$. The function $f_P : (f_G, X) \mapsto r$ (where X is an input pattern of G and $r \in I\!B$ is the result) evaluates the property P, i.e. f_P maps to 1 if, and only if, the circuit G with the input pattern X satisfies P. Given that, the *property checking* problem is defined by proving that $\forall X. f_P(f_G, X) = 1$ holds for a given circuit G and a given property P.

If f_P maps to 1 for all possible input patterns X of G, then the circuit G satisfies the considered property. However, the evaluation of this formula would require 2^n computations of f_G. Instead, it is often easier to determine a single counter-example X_{CEX} for which the property is not satisfied. This can be expressed as $\exists X_{\text{CEX}}.f_P(f_G, X_{\text{CEX}}) = 0$. This requires the evaluation of all 2^n input patterns only in the worst case.

As a result, property checking can be considered as the problem of determining an input pattern X_{CEX} of G so that G does not satisfy P or to prove that no such pattern exists.

3.2 Quantum Circuit Verification

The verification of conventional hardware is a well-considered field (see e.g. [13]). The approaches developed in the last decades are highly optimized and have been implemented in efficient tools. These accomplishments can be exploited when reversible circuits are considered exclusively. Since here all operations only act on Boolean values, it is sufficient to simply map these circuits to conventional gate libraries and afterwards apply the existing methods such as bounded model checking [16] or equivalence checking [17].

In contrast, quantum circuits contain non-Boolean values and often have non-Boolean outputs. As a consequence, the approaches developed so far for Boolean circuits are not applicable. Yet only few formal verification approaches for quantum circuits exist. In [18], the quantum model checker QMC has been introduced. This model checker has specifically been developed to check properties of quantum protocols. However, it is uncertain whether this approach is applicable to other more generic quantum algorithms or how the model checker behaves when used for the verification of larger systems. To the best of our knowledge, no studies in these fields have been published so far.

Instead, the majority of verification approaches for quantum circuits applies simulation techniques [19–21]. In order to simulate a circuit, several stimuli, i.e. input patterns, are generated and the respective output patterns are produced. Each of these output patterns needs to be checked against the specification in order to determine the correctness of the design. On the one hand, simulation is a very fast and inexpensive method, especially for determining a circuit's behavior when only a few cases are concerned. As a result, it can be used to ensure the correctness for several critical or common input patterns. However, a circuit with n variable inputs has 2^n possible input patterns and, thus, coverage of the entire behavior through simulation is intractable as all 2^n patterns need to be simulated resulting in an impracticably large run-time. Additionally, the operations performed by quantum circuits are usually described by complex-valued matrices. As many simulation approaches apply matrix multiplication, they have high memory requirements as well.

In order to especially address the latter problem, the simulator QuIDDPro has been introduced in [21]. QuIDDPro employs a particular data-structure, so-called *Quantum Information Decision Diagrams* (QuIDDs), to simulate quantum circuits. QuIDDs have been explicitly developed in such a way that they allow

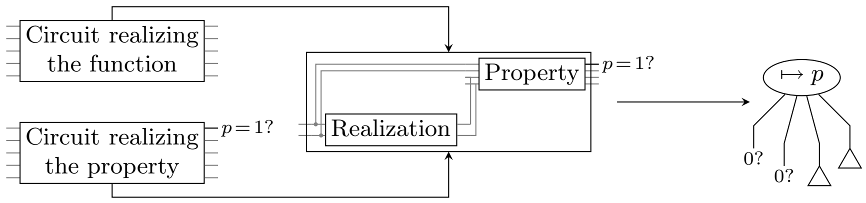

Circuit realizing the function
Circuit realizing the property
$p = 1?$
Property
Realization
$p = 1?$
$\mapsto p$
0?
0?

Fig. 4. Proposed verification flow

efficient quantum circuit simulation. Existing evaluations show that QuIDDPro outperforms all other known simulation approaches when only one simulation run is considered. However, QuIDDPro is naturally bounded to the number of qubits that correspond to the inputs which again leads to an exponentially growing run-time for a complete simulation.

Since simulation of conventional circuits leads to similar problems, conventional hardware is usually either entirely verified by means of formal methods or just validated by means of partial simulation for certain crucial cases. In this work, we aim for a verification of quantum circuits. That is, we introduce a formal verification approach for quantum circuits in order to provide an alternative to the existing simulation-based approaches.

3.3 General Idea

Figure 4 outlines the underlying idea of the verification flow proposed in this paper. The input to the flow are the circuit under verification and the property to be checked. The property is represented by means of the function f_P realized as a circuit.

Having this, a naïve property check could be done as follows. Each possible input combination is assigned to the inputs of the original circuit which is used to evaluate the corresponding output assignment. Then, both assignments are given as input to the circuit realizing the property (see center of Fig. 4). If the output signal f_P is 1 for each input assignment, the property holds. Although this process can be simplified by combining both circuits and connecting the inputs and outputs of the original function to the property circuit, this will not change the complexity of the verification procedure.

However, with certain modifications which are outlined in detail in the next section, both circuits can be combined and represented by a QMDD. When constructing the corresponding QMDD in a way such that the property signal f_P is located at the top-most line, possible input/output mappings of f_P will be represented by the root node of the QMDD. This easily allows to check whether there exists an input assignment such that f_P evaluates to 0, i.e. such that the property is not satisfied. In fact, it just has to be checked whether the first two successors from the root node lead to path to $\boxed{1}$ (see right-hand-side of Fig. 4). In this case a mapping to $f_P = 0$ exists, i.e. there is a input assignment which violates the property. Hence, checking whether the property holds can be done

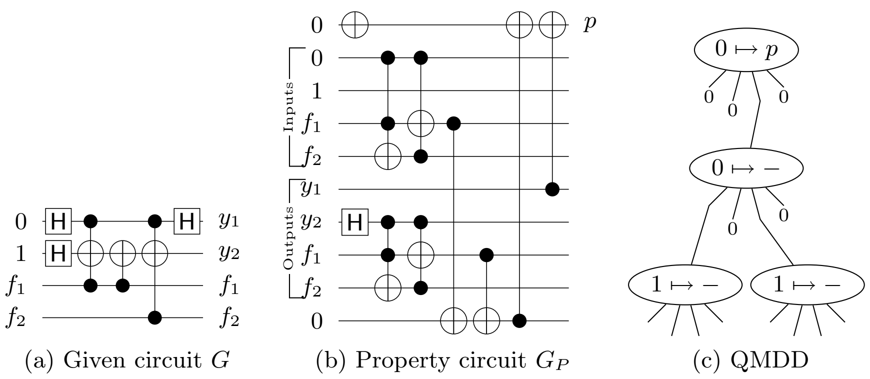


Fig. 5. Application of the proposed verification flow

with a constant number of look-ups after the QMDD has been determined. This general idea is illustrated by the following example.

Example 3. Figure 5(a) shows a generalization of the quantum circuit realizing the Deutsch algorithm [22]. Usually the oracle is the input to the Deutsch problem and the circuit solely consists of constant inputs. The generalization allows for *configuring* all possible four oracles using two additional circuit lines f_1 and f_2 which combinations lead to all truth tables 00, 01, 10, and 11 representing constant 0, identity, negation, and constant 1, respectively.

The property represented by the circuit in Fig. 5(b) will explicitly evaluate the possible functions and afterwards compare the result gathered from the original circuit on signal y_1. A (partial) QMDD representing the combined circuit is depicted in Fig. 5(c)[1]. Since the first two successors of the root node point to $\boxed{0}$, no input assignment exists which maps to $f_P = 0$. Hence, the property has been proven to be true, i.e. the circuit indeed realizes the Deutsch algorithm.

4 Implementation

This section describes the algorithm implementing the idea proposed above and illustrates the resulting verification flow by means of an example. After a brief overview of the main steps, these steps are discussed in detail.

Given are a quantum circuit G and a property P to be verified. The property P is evaluated by f_P (see Sect. 3.1) which is represented by a circuit G_P over the inputs and outputs of G. The property is satisfied if f_P always evaluates to 1, i.e. f_P is tautologous. The aim of the procedure is to obtain a QMDD which represents the result f_P. Consequently, G and G_P have to be altered in such a way that a combined circuit of the two, in the following denoted by G_C, can

[1] Note that this QMDD has been modified to handle constant inputs. This is described in detail in the next section.

be used as a basis for the QMDD Q. Since QMDDs can only represent unitary functions, this particularly includes a transformation of G_C into a unitary function realization. To this end, additional circuit lines (assuming constant inputs) need to be added. Those lines have to be explicitly handled when obtaining the result of the property check (i.e. f_P from the QMDD).

Overall, the respective algorithm executes the following steps.

1. Combine G and G_P to G_C and make G_C unitary
2. Build a QMDD Q for G_C
3. Modify the QMDD Q such that additional circuit lines (assuming constant inputs) are handled
4. Obtain the result of the property check from the root node of the QMDD Q

In the following sections, these steps are described in detail.

4.1 Combine the Circuits and Ensure Unitary

The goal of the first step is to alter both circuits G and G_P in such a way that they can be combined into one circuit G_C by concatenating G and G_P. Based on G_C, a QMDD is built in the next step.

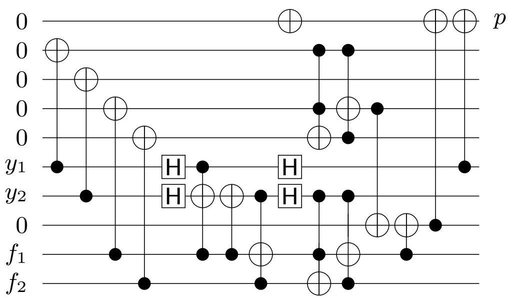
First, G and G_P have to be defined over the same set of variables and the variables have to be in the same order, because they cannot be combined into one circuit otherwise. Consequently, both circuits are checked for variables which occur exclusively in the respective circuit. These variables are then added to the other circuit in such a way that the variable ordering is the same for G and G_P.

Since from the combined circuit G_C a QMDD is supposed to be created, G_C has to represent a unitary circuit which therefore also applies for G and G_P. While it already holds for G, G_P may be non-unitary and, thus, has to be extended accordingly. Here, existing approaches for *embedding* (introduced e.g. in [23]) can be exploited. Furthermore, in order to ensure that also the combined circuit G_C is unitary, the fan-outs have to be removed which are caused by the fact that both G and G_P use the same inputs. This is done by adding additional circuit lines and gates that keep copies of the inputs. The following example illustrates this step.

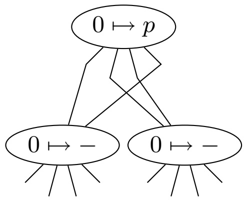
Example 4. Consider again the circuits in Figs. 5(a) and 5(b) representing a given circuit G and a property circuit G_P. As can be seen, three additional lines (assuming constant values) are necessary in order to properly embed G_P. Afterwards, both circuits are combined to G_C as shown in Fig. 6(a). For this purpose, four gates are added replacing the non-unitary fan-outs, i.e. copying the values of the inputs y_1, y_2, f_1, and f_2.

4.2 Build a QMDD from the Combined Circuit

Before building a QMDD from the combined circuit, the lines are reordered in such a way that all lines assuming a constant input value are placed at the top of



(a) Combined circuit



(b) QMDD representing the complete function

Fig. 6. Intermediate steps of the proposed verification flow

the circuit. Such a structure simplifies the succeeding steps. Afterwards, a QMDD is built representing G_C, i.e. the combined circuit of G and G_P. However, the resulting QMDD cannot be used to determine the result of the property check as it does not explicitly handle constant inputs.

Example 5. Figure 6(b) shows the QMDD representing the combined circuit G_C from Fig. 6(a). As can be seen, constant input values have not been considered yet. In fact, the root node has four outgoing edges leading to non-terminal nodes. That is, all possible input/output mappings of the first line (including output f_P) are considered. The underlying circuit however assumes a constant input 0 at this line, i.e. only the first and the third outgoing edge should be considered. Hence, the QMDD needs to be modified as described in the following step.

4.3 Modify the QMDD

Obtaining a QMDD Q which also takes constant input values into consideration requires altering the QMDD in order to avoid an incorrect result. As a consequence, each node representing a line with a constant input value has to be checked. If such a node has outgoing lines representing an input/output assignment which does not occur in the circuit, then the respective edge is replaced by an edge leading to $\boxed{0}$.

As a result from eliminating edges in the QMDD, it can occur that nodes remain whose edges all point to $\boxed{0}$. Since these nodes never appear in paths from the root node to $\boxed{1}$ and thus do not contribute to valid input/output mappings, they can be removed from the QMDD. The resulting QMDD represents the combined circuit G_C and additionally considers constant input values.

Example 6. Figure 5(c) shows the QMDD after edges and nodes have been eliminated as described above. Now, the root node has only one outgoing edge leading to a non-terminal node which corresponds to the defined constant input value.

The second and fourth edge have been eliminated through edge elimination, while the first edge has been eliminated in the course of node elimination.

4.4 Determine the Result

After the steps described above, the resulting QMDD represents all input/output mappings considering the given circuit G, the given property P, and the possibly assumed constant inputs. Of particular interest is whether there exists a mapping to $f_P = 0$. If that is the case, it has been shown that the resulting circuit does not hold the property. Since f_P is the top-most variable and, thus, represented by the root node of Q, the existence of such a mapping can be obtained by evaluating the root node and its outgoing edges. An existing mapping is indicated by the existence of the respective edge.

If the property check does not hold, then the first line maps from 0 to 0 and a counter-example can be derived from the QMDD. Such a counter-example is an input pattern for which the property is not satisfied. It can be obtained by traversing a path from the top-most node to $\boxed{1}$ which starts with the first outgoing edge of the top-most node.

Example 7. The root node of the QMDD in Fig. 5(c) indicates the result of the property check. Only the third edge leads to a non-terminal node. This edge represents the mapping $0 \mapsto 1$ for the top-most line in the combined circuit. Hence, f_P always maps to 1 and, thus, the property always holds.

5 Experimental Results

The verification flow introduced in the previous sections has been implemented and evaluated by verifying several instances of the Grover search and the Deutsch algorithm. The realizations for both algorithms have been generalized as described in Sect. 3.3. The corresponding properties were automatically synthesized. All operations concerning QMDDs, including the construction, were provided as a C-library [15]. The experiments have been conducted on a 2.3 GHz Intel Core i5 with 2GB main memory running Linux as a virtual machine.

The results of the experiments are shown in Table 1. They were compared to those obtained by the simulator QuIDDPro [21] which was applied to the same circuits as the proposed verification procedure. In addition to that, we distinguished between holding and failing property checks. For satisfying properties, the circuits have been completely simulated using QuIDDPro. For failing properties the simulation was only performed until a counter-example was determined.

The table is structured as follows. In the first column, the name of the circuit which was verified is given. In the second and third column, the number of lines and gates is denoted, split into the number of the combined and the original circuits to be checked. The remaining columns show the run-times of QuIDDPro and the QMDD-based verification flow for holding and failing properties.

In case of satisfying properties, the QMDD-based verification approach is faster than QuIDDPro except for one test case. For the smaller circuits, the

Table 1. Results for passing property checks

			Holding properties		Failing properties	
Circuit	Lines	Gates	QuIDDPro	QMDD	QuIDDPro	QMDD
Grover2	6 (5)	26 (19)	0.07	**0.01**	**0.03**	1.65
Grover3	8 (7)	111 (83)	0.72	**0.03**	**0.11**	1.67
Grover4	10 (9)	117 (107)	**2.08**	2.83	**0.12**	1.72
Grover5	12 (11)	144 (132)	5.60	**4.48**	**0.16**	1.94
Grover6	14 (13)	165 (151)	10.65	**1.63**	**0.24**	1.68
Grover7	16 (15)	263 (247)	33.27	**1.70**	**0.36**	1.66
Grover8	18 (17)	229 (211)	79.98	**3.03**	**0.28**	8.00
Grover9	20 (19)	182 (162)	timeout	**16.98**	**0.16**	18.00
Grover10	22 (21)	201 (179)	timeout	**230.48**	**0.19**	244.69
Deutsch	12 (6)	21 (8)	0.04	**0.01**	**0.02**	1.66
Deutsch-Josza	23 (22)	75 (22)	0.14	**0.03**	**0.07**	1.67

run-times are similar. However, as the number of lines and gates increases, the QMDD-based approach becomes much faster than QuIDDPro. In particular, the QMDD-based approach could still be applied for the circuits *Grover9* and *Grover10*, while the simulation with QuIDDPro does not finish. This occurs due to the complete simulation which has to be performed by QuIDDPro.

For failing property checks, the results are reversed. Whereas QuIDDPro performs very good, the run-time of the QMDD-based approach is nearly identical to that of the holding property checks. Since QuIDDPro does not need to completely simulate the circuits, but terminates the simulation as soon as a counter-example is determined, the run-time can be reduced significantly. In contrast, the QMDD is built completely for all possible input/output mappings instead of particular simulation patterns.

Although the simulation performs much faster for failing property checks, the proposed verification flow is more robust. As a result, the run-time of the verification does not depend on the outcome of the verification whereas simulation can lead to a timeout.

6 Conclusions

In this work, a new verification approach for quantum circuits has been presented. We described how QMDDs can be applied in order to prove correctness of a design and evaluated our approach by verifying realizations of Grover's algorithm and Deutsch's algorithm. In contrast to QuIDDPro, one of the best known quantum circuit simulators, the run-time of our approach is not dependent on the result of the property check and can prove correctness of a design much faster than the simulator.

Acknowledgments. We would like to sincerely thank D. Michael Miller for providing us with an implementation of the QMDD package introduced in [15] and for many helpful discussions. This work was supported by the German Research Foundation (DFG) (DR 287/20-1) and the German Academic Exchange Service (DAAD).

References

1. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, New York (2000)
2. Hung, W.N.N., Song, X., Yang, G., Yang, J., Perkowski, M.A.: Quantum logic synthesis by symbolic reachability analysis. In: Malik, S., Fix, L., Kahng, A.B. (eds.) Design Automation Conference, pp. 838–841. ACM (June 2004)
3. Shende, V.V., Bullock, S.S., Markov, I.L.: Synthesis of quantum logic circuits. In: Tang, T. (ed.) Asia and South Pacific Design Automation Conference, pp. 272–275. ACM Press (January 2005)
4. Große, D., Wille, R., Dueck, G.W., Drechsler, R.: Exact Synthesis of Elementary Quantum Gate Circuits for Reversible Functions with Don't Cares. In: Int'l Symp. on Multiple-Valued Logic, pp. 214–219 (May 2008)
5. Maslov, D., Dueck, G.W., Miller, D.M., Negrevergne, C.: Quantum Circuit Simplification and Level Compaction. IEEE Trans. on CAD 27(3), 436–444 (2008)
6. Soeken, M., Wille, R., Dueck, G.W., Drechsler, R.: Window optimization of reversible and quantum circuits. In: Int'l Symp. on Design and Diagnostics of Electronic Circuits and Systems, pp. 341–345 (April 2010)
7. Yuan, J., Shultz, K., Pixley, C., Miller, H., Aziz, A.: Modeling design constraints and biasing in simulation using BDDs. In: Int'l Conf. on Computer-Aided Design, pp. 584–590 (November 1999)
8. Bergeron, J.: Writing Testbenches Using SystemVerilog. Springer (2006)
9. Yuan, J., Pixley, C., Aziz, A.: Constraint-Based Verification. Springer (January 2006)
10. Wille, R., Große, D., Haedicke, F., Drechsler, R.: SMT-based Stimuli Generation in the SystemC Verification Library. In: Forum on Specification & Design Languages (September 2009)
11. Brand, D.: Verification of large synthesized designs. In: Lightner, M.R., Jess, J.A.G. (eds.) Int'l Conf. on Computer-Aided Design, pp. 534–537. IEEE Computer Society (1993)
12. Disch, S., Scholl, C.: Combinational Equivalence Checking Using Incremental SAT Solving, Output Ordering, and Resets. In: Asia and South Pacific Design Automation Conference, pp. 938–943 (2007)
13. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
14. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
15. Miller, D.M., Thornton, M.A.: QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits. In: Int'l Symp. on Multiple-Valued Logic, p. 30. IEEE Computer Society (May 2006)
16. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
17. Wille, R., Große, D., Miller, D.M., Drechsler, R.: Equivalence Checking of Reversible Circuits. In: Int'l Symp. on Multiple-Valued Logic, pp. 324–330. IEEE Computer Society (May 2009)
18. Gay, S.J., Nagarajan, R., Papanikolaou, N.: QMC: A Model Checker for Quantum Systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 543–547. Springer, Heidelberg (2008)

19. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. Phys. Rev. A 70, 052328 (2004)
20. Vidal, G.: Efficient Classical Simulation of Slightly Entangled Quantum Computations. Phys. Rev. Letters 91, 147902 (2003)
21. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Quantum Circuit Simulation. Springer, Heidelberg (2009)
22. Deutsch, D.: Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer. Royal Society London A 400(1818) (July 1985)
23. Miller, D.M., Wille, R., Dueck, G.: Synthesizing Reversible Circuits for Irreversible Functions. In: EUROMICRO Symp. on Digital System Design, pp. 749–756 (2009)

Using πDDs in the Design of Reversible Circuits (Work-In-Progress)

Mathias Soeken[1,3], Robert Wille[1], Shin-ichi Minato[2], and Rolf Drechsler[1,3]

[1] Institute of Computer Science, University of Bremen
Group of Computer Architecture, D-28359 Bremen, Germany
{msoeken,rwille,drechsle}@informatik.uni-bremen.de
[2] Hokkaido University
Sapporo 060-0814, Japan
minato@ist.hokudai.ac.jp
[3] Cyber-Physical Systems, DFKI GmbH
D-28359 Bremen, Germany
rolf.drechsler@dfki.de

Abstract. With πDDs a data structure has recently been introduced that offers a compact representation for sets of permutations. Since reversible functions constitute permutations on the input assignments, they can naturally be expressed using this data structure. However, its potential has not been exploited so far. In this work-in-progress report, we present and discuss possible applications of πDDs within the design of reversible circuits including techniques for synthesis, debugging, and an efficient determination of the number of minimal circuits. We observed that πDDs inhibit the same space complexities as truth tables and, hence, do not superior existing design methods in many cases. However, they are advantageous when dealing with several functions or gates at once.

1 Introduction

Decision diagrams offer a compact representation of Boolean functions and matrices and, thus, have been widely applied in the design of reversible circuits. As examples, *Binary Decision Diagrams* (BDDs) have been applied for exact, heuristic, and hierarchical synthesis of both reversible and irreversible functions [1–3]. As an alternative to BDDs, the application of Kronecker functional decision diagrams, an extension of BDDs, has lead to further improvements [4]. *Quantum Multiple-valued Decision Diagrams* (QMDDs) [5], enabling a compact representation for complex matrices, have been used for both equivalence checking [6] and synthesis of large reversible functions ensuring a minimal number of lines [7]. Similar data-structures have efficiently been applied for the simulation and verification of quantum circuits [8, 9]. In fact, decision diagrams have been the key methodology for breakthroughs in the design of reversible circuits. For the first time, BDDs allowed synthesis of minimal circuits for a significant amount of functions [1] and they enabled the synthesis of large Boolean functions with more than 100 variables [2]. For the latter case, the main problem of the algorithm is the huge amount of extraneous lines which impedes the practical applicability of that approach. However, the problem of additional lines in the synthesis of large functions has been solved again with decision diagrams, in particular using QMDDs [7].

However, while BDDs and QMDDs offer a compact representation for functions and matrices, the recently introduced πDDs [10] allow for a compact representation for permutations. Hence, they are an interesting extension to the set of considered decision diagrams in the design of reversible circuits. Since reversible functions constitute permutations on the input assignments, they can naturally be expressed using this data structure. In fact, πDDs do not only allow a compact representation for single permutations, but for a set of permutations. Therefore, they can particularly be applied for many problems where the above mentioned data structures are not advantageous.

In this work-in-progress report, we briefly review the underlying data-structure and afterwards discuss possible applications in different areas of the design of reversible circuits. These applications include synthesis, debugging, and an efficient determination of the number of minimal circuits. Our observations show that, πDDs inhibit the same space complexities as truth tables (i.e. they are exponential in space) and, hence, in many cases do not superior existing methods. However, advantages can be gained when dealing with several functions or gates at once.

2 Preliminaries

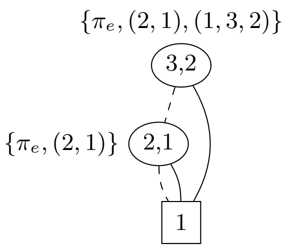
2.1 Reversible Functions and Circuits

A function $f : \mathbb{B}^n \to \mathbb{B}^n$ is called reversible if it represents a bijection, i.e. each input maps uniquely to an output pattern. As a result, reversible functions represent permutations on the set $\{0, \dots, 2^n - 1\}$. Reversible functions can be realized using reversible circuits. The process of determining a reversible circuit for a given function is called *synthesis*. The circuits are usually composed as a cascade of reversible gates where the Toffoli gate [11] constitutes their most prominent representative. Given a set of variables $X = \{x_1, \dots, x_n\}$ a Toffoli gate is a tuple (C, t) with $C \subset \bigcup_{x \in X} \{x, \overline{x}\}$ such that $\forall x \in X : \{x, \overline{x}\} \not\subset C$ being the set of *control lines* and $t \in X$ with $\{t, \overline{t}\} \cap C = \emptyset$ being the *target line* of the gate. A Toffoli gate inverts the target line if, and only if, all control lines x_i ($\overline{x}_i$) are set to 1 (0). Positive (negative) literals in C are called positive (negative) control lines.

2.2 The πDD Data-Structure

The πDDs [10] allow for a compact representation of sets of permutations and work similar to ZDDs [12] which allow for a compact representation of sets of variables. πDDs exploit that permutations can be decomposed into elementary transpositions swapping two elements, i.e. that a permutation can be seen as a set of its transpositions. As an example the permutation $(3, 5, 2, 1, 4)$ can be represented by a sequence of transpositions $\tau_{(2,1)}\tau_{(3,2)}\tau_{(4,1)}\tau_{(5,4)}$, i.e. first the items 5 and 4 are interchanged, then 4 with 1 and so on until the identity permutation $\pi_e = (1, 2, 3, 4, 5)$ results. When always swapping the elements with the highest absolute value first, sequences of transpositions are canonical.

According to this principle, the vertices in πDDs are labeled using the respective transposition (in comparison, in ZDDs the vertices are labeled using the set element). The terminal vertices 1 and 0 represent the set containing the identity permutation $\{\pi_e\}$ and the empty set $\emptyset$, respectively. As an example, on the left-hand side, a πDD is illustrated representing the permutations $\{\pi_e, (2, 1), (1, 3, 2)\}$. Traversing this πDD from the top to the bottom leads to the transpositions to be applied so that eventually the identity permutation results.



Several operations can be carried out efficiently on πDDs, e.g. counting the number of permutations which is equivalent to counting the number of 1-paths in BDDs or ZDDs. Furthermore, calculating the Cartesian product $P*Q = \{\alpha\beta|\alpha \in P, \beta \in Q\}$ is efficient, which is the set of all possible composite permutations chosen from P and Q. Due to page limitations, the reader is referred to [10] for a comprehensive discussion on πDDs.

3 Applications in the Design of Reversible Circuits

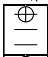
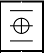
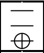
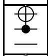
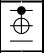
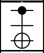
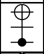
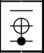
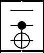
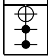
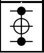
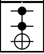
3.1 Determination of the Number of Minimal Circuits

As discussed in the previous section, πDDs allow for a compact representation of sets of permutations as well as efficient operations such as counting the permutations and the Cartesian product. If gates in a circuit are considered as permutations, the latter naturally expresses the gate composition in reversible circuits. Combining both operations allows for an efficient determination of the number of minimal circuits.

This is achieved by creating a set of all elementary gates that may occur in a circuit. Afterwards, the Cartesian product on these gates is iteratively constructed. As a result, all reversible functions are enumerated and contained in the resulting πDD. By extracting the transpositions from the paths in the πDD, it can easily be obtained how many minimal circuits composed of a certain number of gates exist. This πDD represents a set of possible functions but does not explicitly represent the structure of gates for each function. However, we can extract the actual gates by using πDD-based operations, as shown in the next section. Nevertheless, the information on the number of minimal circuits already is interesting for statistical purposes.

Table 1 lists all permutations for positively controlled Toffoli gates acting on 3 lines. For the sake of an improved readability transpositions $\tau_{(x,y)}$ are written as $\binom{x}{y}$. Each permutation is denoted $T_{t,\mu}$ where t is the target line of the gate and μ is an index denoting the set of control lines. Given that, the

Table 1. Permutations for all positively controlled Toffoli gates on 3 lines

μ	$T_{0,\mu}$		$T_{1,\mu}$		$T_{2,\mu}$	
0		$\binom{000}{001}$ $\binom{010}{011}$ $\binom{100}{101}$ $\binom{110}{111}$		$\binom{000}{010}$ $\binom{001}{011}$ $\binom{100}{110}$ $\binom{101}{111}$		$\binom{000}{100}$ $\binom{001}{101}$ $\binom{010}{110}$ $\binom{011}{111}$
1		$\binom{010}{011}$ $\binom{110}{111}$		$\binom{001}{011}$ $\binom{101}{111}$		$\binom{001}{101}$ $\binom{011}{111}$
2		$\binom{100}{101}$ $\binom{110}{111}$		$\binom{100}{110}$ $\binom{101}{111}$		$\binom{010}{110}$ $\binom{011}{111}$
3		$\binom{110}{111}$		$\binom{101}{111}$		$\binom{011}{111}$

set of all positively controlled Toffoli gates acting on n lines can be written as $T_n = \bigcup_{t=0}^{n-1} \bigcup_{\mu=0}^{2^{n-1}-1} T_{t,\mu}$. Since reversible functions can also be represented by permutations, we can count all functions realized by minimal circuits using F_k where k denotes the number of gates with

$$F_0 = \{\pi_e\}, \quad F_1 = F_0 \cup T_n, \quad \text{and} \quad F_k = F_{k-1} * T_n \text{ for } k > 1 \ . \tag{1}$$

The approach can easily be adapted to support other gate libraries such as Toffoli gates containing also negative control lines (denoted by $T_n^{\pm}$) as well as libraries that only consist of Toffoli gates that are fully controlled, i.e. $|C| = n-1$ for each gate (denoted by $\hat{T}_n$ and $\hat{T}_n^{\pm}$). Table 2 shows the number of circuits determined for k ranging from 0 to 12 for all these four gate libraries. In fact, only the number of newly found functions is listed, i.e. $|F_k| - |F_{k-1}|$. As can be seen, all gate libraries except for $\hat{T}_3$ are universal, since the number of all reversible functions over 3 variables is $2^3! = 40320$. In fact, only 24 functions can be represented when using gates exclusively from library $\hat{T}_3$. Furthermore, the numbers for the gate library $\hat{T}_3^{\pm}$ are very interesting as they are more balanced than the other ones. Also, this gate library should be more efficient when used with πDDs since all elementary gates can be represented by permutations that are composed of only one transposition. The largest minimal function consists of 12 gates when using this library and is described by the permutation $(7, 6, 4, 5, 1, 0, 2, 3)$.

Table 2. Size of $|F_k| - |F_{k-1}|$ for four gate libraries

k	T_3	$T_3^{\pm}$	$\hat{T}_3$	$\hat{T}_3^{\pm}$
0	1	1	1	1
1	12	27	3	12
2	102	369	6	90
3	625	2925	9	476
4	2780	13282	5	1903
5	8921	20480	0	5472
6	17049	3236	0	10388
7	10253	0	0	11756
8	577	0	0	7347
9	0	0	0	2408
10	0	0	0	430
11	0	0	0	36
12	0	0	0	1
$\sum$	40320	40320	24	40320

All this information can easily be extracted using the operations supported on πDDs within less than a second for all gate libraries. However, when performing the same experiments for $n = 4$, the approach is not scalable anymore. Note that increasing n by 1 doubles the number of elements in the respective permutations. Further, the number of vertices in the πDD is the number of elements squared, i.e. 2^{2n}.

3.2 Synthesis with Minimal Number of Gates

Based on the results and procedures of the previous section, a synthesis algorithm realizing a function f with a minimal number of gates can be formulated. For this purpose, the function f to be synthesized is represented as permutation π_f. Then, all functions are enumerated as in Eq. (1). After each step k, it is checked whether π_f is contained in all functions F_k. In that case, the minimal number of gates k is already determined. However, the actual circuit has not been obtained yet. For this purpose, the algorithm moves backwards going to F_0 by applying gates from T_n. More precisely, an algorithm for the synthesis of reversible functions ensuring minimal number of gates can be formulated as follows.

Algorithm E (*Exact Synthesis*). The reversible function f to be synthesized is given as permutation π_f.

E1. [Initialize.] Set $F_0 \leftarrow \{\pi_e\}$, $F_1 \leftarrow F_0 \cup T_n$, and $k \leftarrow 1$.

E2. [Found minimum?] If $F_k \cap \{\pi_f\} \neq \emptyset$, i.e. $\pi_f \in F_k$, go to step E4.

E3. [Increase k.] Set $k \leftarrow k + 1$, $F_k \leftarrow F_{k-1} * F_1$ and return to step E2.

E4. [Extract gate.] Select $\pi \in T_n$ such that $\pi_f \pi \in F_{k-1}$.

E5. [Next gate?] Set $k \leftarrow k - 1$ and $\pi_f \leftarrow \pi_f \pi$. If $k = 1$, terminate, otherwise return to step E4. ▮

This algorithm faces similar problems as discussed in the end of Sect. 3.1, i.e. the complexity raises significantly with increasing size of the function. One possibility to address that is to use a different gate library such as $\hat{T}_n^{\pm}$ which contains smaller permutations. However, as for any other existing exact synthesis approach, the size of all elements contained is still exponential, i.e. $|T_n| = |\hat{T}_n^{\pm}| = n \cdot 2^{n-1}$. This will always cause scalability problems in step E4 in which all gates need to be traversed in the worst case.

3.3 Heuristic Synthesis

Unlike the exact synthesis approach, where all functions are enumerated first in order to check whether the function f to be synthesized is contained, the starting point of the heuristic synthesis approach is the function f itself. Similar to the QMDD-based synthesis procedure [7] gates (or permutations) should be applied according to the structure of the πDD for the function in its current state. However, the πDD-based approach allows for applying several gates at once instead of only one at a time. The result can then be checked for the best current solution and then proceed from there. The goal is to transform the πDD representing π_f by means of gate operations such that eventually the identity function, i.e. π_e is reached. Since the corresponding πDD consists only of one terminal vertex, the aim during synthesis is to constantly reduce the number of non-terminal vertices in the πDD.

3.4 Debugging

As discovered in the previous section, the πDDs for reversible functions grow exponentially with respect to the number of lines. As a result, it is likely that the above mentioned techniques are not applicable to circuits of a larger scale. However, πDDs are advantageous when considering multiple functions at once which should be illustrated in this section. Consider a simple debugging problem to be solved where a faulty circuit should be checked for a *missing-gate* defect. Given a circuit $C = g_1 \dots g_d$ consisting of n lines where each gate g_i can be described by its permutation π_{g_i} and a function f represented by π_f, this debugging problem can be solved using πDDs by checking if $\pi_f \in F$, where

$$F = \bigcup_{i=0}^{d} \left(\{\pi_{g_1} \dots \pi_{g_i}\} * T_n * \{\pi_{g_i+1} \dots \pi_{g_d}\} \right) \quad .$$

All operations can be carried out efficiently on the πDD data structure.

4 Conclusions and Future Work

The πDD for one function can be exponential in size, in fact the permutation for a NOT gate (Toffoli gate without control lines) in a circuit with n lines consists of 2^{n-1} transpositions which is equal to the number of non-terminal vertices in the πDD. As a result, πDDs are not suitable for the synthesis of large functions and thus probably not suitable for synthesis in general. Since determining the minimal number of circuits for 4 circuit lines is already inefficient, it is not to expect that the exact synthesis algorithm based on πDDs can keep up with the results achieved using the exact synthesis approaches based on Boolean satisfiability [13]. However, conceptual algorithms that have been discovered allow an efficient counting and enumerating of reversible functions of small sizes. Interesting statistics comparing different gate libraries have been observed.

Further, the πDDs are advantageous when several functions at once should be considered at the same time, whereas no other graphical data-structure that has been used in the design for reversible circuits so far possesses this property. As a result, the efficient storing of multiple permutations can be exploited. Hence, πDDs should be used in the context of algorithms for reversible functions and circuits that inhibit these properties, e.g. within debugging where πDDs allow to consider several solutions at once.

Algorithms of such kind should be considered in future work. Furthermore, the performance of the πDD implementation should be enhanced such that the determination of the minimal number of circuits can be performed for a larger number of circuit lines. The size of the permutations that are represented through reversible functions grows exponentially as the number of circuit lines grows linearly. This affects the performance of the πDDs in a bad manner, since they allow permutations of all sizes and not only the special cases represented by reversible functions. As a result, also the decomposition technique that serves as the base for current πDDs should be inspected for improvement in the special case of reversible functions, e.g. by explicitly targeting reversible gates as atomic unit instead of transpositions.

Nevertheless, we are convinced that πDDs fit well in the current zoo of graphical data-structures. Although they do not allow an improvement of current techniques they will serve as an efficient data-structure for problems that explicitly require the use of several gate operations or the consideration of several functions in general.

Acknowledgments. This work was supported by the German Research Foundation (DFG) (DR 287/20-1).

References

1. Wille, R., Le, H.M., Dueck, G.W., Große, D.: Quantified Synthesis of Reversible Logic. In: Design, Automation and Test in Europe, pp. 1015–1020. IEEE (March 2008)
2. Wille, R., Drechsler, R.: BDD-based synthesis of reversible logic for large functions. In: Design Automation Conference, pp. 270–275. ACM (July 2009)
3. Kerntopf, P.: A New Heuristic Algorithm for Reversible Logic Synthesis. In: Design Automation Conference, pp. 834–837 (June 2004)
4. Soeken, M., Wille, R., Drechsler, R.: Hierarchical synthesis of reversible circuits using positive and negative Davio decomposition. In: Int'l Design and Test Workshop, pp. 143–148 (December 2010)
5. Miller, D.M., Thornton, M.A.: QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits. In: Int'l Symp. on Multiple-Valued Logic, p. 30. IEEE Computer Society (May 2006)
6. Wille, R., Große, D., Miller, D.M., Drechsler, R.: Equivalence Checking of Reversible Circuits. In: Int'l Symp. on Multiple-Valued Logic, pp. 324–330. IEEE Computer Society (May 2009)
7. 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 (January 2012)
8. Viamontes, G.F., Markov, I.L., Hayes, J.P.: Quantum Circuit Simulation. Springer, Heidelberg (2009)
9. Wang, S.A., Lu, C.Y., Tsai, I.M., Kuo, S.Y.: An XQDD-based verification method for quantum circuits. IEICE Transactions 91-A(2), 584–594 (2008)
10. Minato, S.-I.: πDD: A New Decision Diagram for Efficient Problem Solving in Permutation Space. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 90–104. Springer, Heidelberg (2011)
11. Toffoli, T.: Reversible Computing. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 632–644. Springer, Heidelberg (1980)
12. Minato, S.: Zero-Supressed BDDs for Set Manipulation in Combinational Problems. In: Design Automation Conference, pp. 272–277 (June 1993)
13. Große, D., Wille, R., Dueck, G.W., Drechsler, R.: Exact Multiple-Control Toffoli Network Synthesis With SAT Techniques. IEEE Trans. on CAD 28(5), 703–715 (2009)

A Verification Technique for Reversible Process Algebra

Jean Krivine[*]

Univ. Paris Diderot, Sorbonne Paris Cité,
Laboratoire PPS, UMR 7126, F-75205 Paris, France

Abstract. A verification method for distributed systems based on decoupling forward and backward behaviour is proposed. This method uses an event structure based algorithm that, given a CCS process, constructs its causal compression relative to a choice of observable actions. Verifying the original process equipped with distributed backtracking on non-observable actions, is equivalent to verifying its relative compression which in general is much smaller. The method compares well with direct bisimulation based methods. Benchmarks for the classic dining philosophers problem show that causal compression is rather efficient both time- and space-wise. State of the art verification tools can successfully handle more than 15 agents, whereas they can handle no more than 5 following the traditional direct method; an altogether spectacular improvement, since in this example the specification size is exponential in the number of agents.

1 Introduction

Backtracking is commonplace in transactional systems where different components, such as processes accessing a distributed database, need to acquire a resource simultaneously. To ensure unconditional correctness of the overall execution of the transaction, one usually provides a code that incorporates explicit escapes from those cases where a global consensus cannot be met. Such an upfront method generates a large and unstructured state space, which often means verification based on proving that the code is bisimilar to a reference specification becomes unfeasible. Based on earlier work, we propose here an indirect verification method, and show on an example that it can handle larger specifications. The idea is to break down the distributed implementation of a given reference specification in two steps. First, one writes down a code which is only required to meet a weaker condition of causal or forward correctness relative to the specification. This condition is parameterized by a choice of observable actions corresponding to the actions of the specification. Second, the obtained code is equipped with a generic form of distributed backtracking on non-observable actions. A general theorem reduces the correctness of the latter partially reversible

[*] This work has been partially supported by the French National Research Agency (ANR), project REVER n° ANR 11 INSE 007.

code to the causal correctness of the former [DK05]. In many transactional examples, this structured programming method works well, and obtains codes which are smaller, and simpler to understand [DKT07]. It also seems interesting from a correctness perspective, since one never has to deal with the full state space, and it is enough to consider the much smaller state space of the forward code causal compression relative to observable actions. Thus it obtains codes which are also easier to prove correct. It is only natural then to ask whether and to which extent such indirect correctness proofs can be automated. This is the question we address in this paper. Specifically we propose an algorithm, which, under certain rather mild assumptions about the system of interest, will compute its causal compression relative to a choice of observables. The true concurrency semantics tradition of using event structures as an intrinsic process representation comes to the rescue here. Besides event structures are uniquely suited to the handling of causal relationships between various events triggered by a process [Win82]. For these reasons our procedure includes a translation of the process as a recursive flow event structure, and computes the relative causal compression on this intermediate representation. Benchmarks given for the classical example of the dining philosophers show a significant state compression, and a relatively low cost incurred by compression. Direct programming generates a state space that is already too big for being constructed by bisimulation verifiers for 6 agents, whereas our method can go well beyond 15. The language we use to formalize concurrent systems is the Calculus of Communicating Systems (CCS) [Mil89]. This is a slightly more expressive language than basic models of communicating automata, in that processes can dynamically fork. On the other hand, this communication model includes no name-passing, which is a severe limitation in some applications.

Section 2 starts with a quick recall of CCS [Mil89]. Section 3 develops its reversible variant RCCS, together with the central notion of causal correctness, and the fundamental result connecting causal correctness of a CCS process and full correctness of its lifting as a partially reversible process in RCCS [DK05]. The relative causal compression algorithm, and the accompanying verification method are explained in Section 4. Section 5 compares this method with the traditional direct method, using the dining philosphers problem as a benchmark. The conclusion discusses related work and further directions.

2 CCS

2.1 Syntax

CCS processes interact through binary communications on named channels: an output on channel x is written $\bar{x}$, an input on the same channel is simply written x.

$$\text{Processes}\, p, q ::= a.p \mid (p \mid q) \mid p + q \mid D(\tilde{x}) := p \mid (x)p \mid 0$$

We write P for the set of processes, A for the set of actions, and A^* for the free monoid of action words. Restriction $(x)p$ binds x in p and the set of free names of p is defined accordingly. In a recursive definition $D(\tilde{x}) := p$ free names of p have to be $\tilde{x}$.

2.2 Operational Semantics

A *labelled transition system* (LTS) is a tuple $\langle S, s, L, \rightarrow\rangle$ where S is called the state space, s the initial state, L the set of labels, and $\rightarrow\ \subseteq S \times L \times S$ the transition relation. One uses the common notation $s \rightarrow_a t$, and for $m = a_1 \dots a_n \in A^*$, $s \rightarrow_m^* t$ means $s \rightarrow_{a_1} s_1, \dots, s_{n-1} \rightarrow_{a_n} t$ for some states $s_1, \dots, s_{n-1}$. The operational semantics of a CCS term p is given by means of such an LTS $(P, p, A, \rightarrow)$, written $\mathsf{TS}(p)$, where $\rightarrow$ is given inductively by the rules:

$$\frac{}{a.p + q \rightarrow_a p}(\mathsf{act}) \qquad \frac{p \rightarrow_a p' \qquad q \rightarrow_{\bar{a}} q'}{p \mid q \rightarrow_\tau p' \mid q'}(\mathsf{synch}) \qquad \frac{p \rightarrow_a p'}{p \mid q \rightarrow_a p' \mid q}(\mathsf{par})$$

$$\frac{p \rightarrow_a p' \quad a \notin \{x, \bar{x}\}}{(x)p \rightarrow_a (x)p'}(\mathsf{res}) \qquad \frac{p \equiv p' \rightarrow_a q' \equiv q}{p \rightarrow_a q}(\mathsf{equiv})$$

The equivalence relation $\equiv$ is the classical structural congruence for choice and parallel composition, together with the recursion unfolding rule $D(\tilde{y}) \equiv p\{\tilde{y}/\tilde{x}\}$ if $D(\tilde{x}) := p$.

2.3 Process Equivalence

Given a set of observable actions, a basic requirement is to decide whether two processes have an equivalent interaction capacity with their environment. Several variants of observational equivalence for CCS processes have been considered. We use here a variant of weak bisimulation based on the choice of a countable distinguished subset K of the set of actions A, which we fix here once and for all. Actions in K are called *observable* actions. The complement $A \setminus K$ of non-observable actions is denoted by K^c and also taken to be countable.

Let $\mathcal{S}_1 = (S_1, s_1, A, \rightarrow)$ and $\mathcal{S}_2 = (S_2, s_2, A, \rightarrow)$ be LTSs both with labels in A, a relation $\mathcal{R}$ over $S_1 \times S_2$ is said to be a *weak simulation* between $\mathcal{S}_1$, $\mathcal{S}_2$, if $s_1\ \mathcal{R}\ s_2$ and whenever $p_1\ \mathcal{R}\ p_2$:

— if $p_1 \rightarrow_a q_1$, $a \in K^c$, then $p_2 \rightarrow_m^* q_2$ with $m \in (K^c)^*$, and $q_1\ \mathcal{R}\ q_2$;
— if $p_1 \rightarrow_a q_1$, $a \in K$, then $p_2 \rightarrow_m^* q_2$ with $m \in (K^c)^* a (K^c)^*$, and $q_1\ \mathcal{R}\ q_2$.

The idea is that $\mathcal{S}_2$ has to simulate the behaviour of $\mathcal{S}_1$ regarding observable actions, but is free to use any sequence of non observable ones in so doing. Such a relation $\mathcal{R}$ is said to be a *weak bisimulation* if both $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are weak simulations. When there is such a relation, $\mathcal{S}_1$ and $\mathcal{S}_2$ are said to be *bisimilar*, and one writes $\mathcal{S}_1 \sim \mathcal{S}_2$. A CCS process p is said to be a *correct implementation* of a specification LTS $\mathcal{S}$, if $\mathsf{TS}(p) \sim \mathcal{S}$. When the specification is clear from the context, we may simply say p is correct. One thing to keep in mind is that all these definitions are relative to a choice of K. Usually, K is taken to be $A \setminus \{\tau\}$, but this more flexible definition will prove convenient.

3 Reversible CCS

We turn now to a quick intuitive introduction to RCCS. Consider the following CCS process:

$$(x)\big(x \mid x \mid \bar{x}.\bar{x}.a.p \mid \bar{x}.\bar{x}.b.q\big) \tag{1}$$

Both subprocesses $a.p$ and $b.q$ require two communications on x to execute, so the whole process may reach a deadlocked state $(x)\big(\bar{x}.a.p \mid \bar{x}.b.q\big)$ where neither a nor b may be triggered. If the intention is that the system implements the mutual exclusion process $a.p+b.q$, a possible fix is to give both subprocesses the possibility to release x:

$$(x)\big(x \mid x \mid R_p(x,a) \mid R_q(x,a)\big) \tag{2}$$

with $R_p(x,a) := \bar{x}.\big(\tau.(R_p(x,a) \mid x) + \bar{x}.(\tau.(R_p(x,a) \mid x \mid x) + a.p)\big)$.

This example helps in realising two key things: first the original code (1) although not correct, is partially correct in the sense that any successful action a or b leads to a correct state p or q; second the proposed fix can be made an instance of a generic distributed backtracking mechanism. The idea of RCCS is to provide such a mechanism, in a way that partial or causal correctness (yet to be defined formally) in CCS, can be proved to be equivalent to full correctness of the same process once lifted to RCCS [DK04].

3.1 Syntax

RCCS *forward actions* are the same actions as CCS, namely A. Recall these are split into K and its complement K^c. In the RCCS context actions in K are also called irreversible, or sometimes commit actions (following the transaction terminology); actions in K^c are also called reversible, since these are the ones one wants to backtrack. RCCS therefore also has *backward actions* written a^-, with $a \in K^c$.

RCCS processes are composed of *threads* of the form $m \triangleright p$, where m is a *memory*, and p is a plain CCS process:

$$r ::= m \triangleright p \mid (r \mid r) \mid (x)r$$

Memories are stacks used to record past interactions:

$$m ::= \langle \theta, a, p\rangle \cdot m \mid \langle\langle \theta \rangle\rangle \cdot m \mid \langle \uparrow \rangle \cdot m \mid \langle\rangle$$

where θ is a thread identifier drawn from a countable set. Open memory elements $\langle \theta, a, p\rangle$ are used for reversible actions and contain a thread identifier θ, the action last taken, and the alternative process that was left over by a choice if any. Closed memory elements $\langle\langle \theta \rangle\rangle$ are used for irreversible actions, and only contain an identifier. Eventually the memory element $\langle \uparrow \rangle$ keeps track of the forking structure of processes (see congruence rules below)[1]. The prefix relation on memories is defined as $m \sqsubseteq m'$ if there is an m'' such that $m'' \cdot m = m'$.

[1] The convention used here for keeping track of forking processes differs slightly from Ref. [DK04].

Processes are considered up to the usual congruence for parallel composition together with the following specific rules:

$$\begin{array}{lll} m \triangleright D(\tilde{y}) & \equiv m \triangleright p\{\tilde{y}/\tilde{x}\} & \text{if } D(\tilde{x}) := p \\ m \triangleright (p \mid q) & \equiv (\langle\uparrow\rangle \cdot m \triangleright p) \mid (\langle\uparrow\rangle \cdot m \triangleright q) & \\ m \triangleright (x)p & \equiv (x)(m \triangleright p) & \text{if } x \notin m \end{array}$$

Any CCS process p can be lifted to RCCS with an empty memory $\ell(p) := \langle\rangle \triangleright p$, and conversely, there is a natural forgetful map φ erasing memories and mapping back RCCS to CCS. Clearly $\varphi(\ell(p)) = p$. When we want to insist that the lift operation is parameterised by the set K, we write $\ell_K(p)$.

3.2 Operational Semantics

The operational semantics of RCCS is also given as an LTS with transitions given inductively by the rules:

$$\frac{a \in K^c \quad \theta \notin m}{m \triangleright a.p + q \rightarrow_a^{\theta} \langle\theta, a, q\rangle \cdot m \triangleright p}(\mathsf{act}) \qquad \frac{a \in K^c}{\langle\theta, a, q\rangle \cdot m \triangleright p \rightarrow_a^{\theta^-} m \triangleright a.p + q}(\mathsf{act}^*)$$

$$\frac{k \in K \quad \theta \notin m}{m \triangleright k.p + q \rightarrow_k^{\theta} \langle\langle\theta\rangle\rangle \cdot m \triangleright p}(\mathsf{commit})$$

$$\frac{r \rightarrow_a^{\Theta} r' \quad \Theta \notin s}{r \mid s \rightarrow_a^{\Theta} r' \mid s}(\mathsf{par}) \qquad \frac{r \rightarrow_a^{\Theta} r' \quad s \rightarrow_{\bar{a}}^{\Theta} s'}{r \mid s \rightarrow_{\tau}^{\Theta} r' \mid s'}(\mathsf{synch})$$

$$\frac{r \rightarrow_a^{\Theta} r' \quad a \neq x, \bar{x}}{(x)r \rightarrow_a^{\Theta} r'}(\mathsf{res}) \qquad \frac{r \equiv r' \rightarrow_a^{\Theta} s' \equiv s}{r \rightarrow_a^{\Theta} s}(\mathsf{equiv})$$

In the contextual rules Θ stands either for θ or θ^-. The freshness of the thread identifier θ is guaranteed by the side conditions $\theta \notin m$ in the (act) and (commit) rules, and $\theta \notin s$ in the (par) rule. The use of such identifiers corresponds to the notation introduced in Ref. [PU06] and equivalent to the one introduced originally for RCCS [DK05], as shown in Ref. [Kri06]. Note that backtracking as defined in the operational semantics is a binary communication mechanism of exactly the same nature as usual forward communication. However, since threads are required to backtrack with the exact same thread with which they communicated earlier, backtrack can be shown to be confluent, at least for those processes that are reachable from the lifting of a CCS process.

The (commit) rule uses a closed memory element $\langle\langle\theta\rangle\rangle \cdot m$ indicating that the information contained in m is no longer needed, since by definition actions in K are not backtrackable. Supposing r is a process where any recursive process definition is guarded by a commit, an assumption to which we will return later on, this bounds the total size of open memory elements in any process reachable from r.

3.3 The Fundamental Property

The question is now to determine what are the possible (definitive) interactions of $\ell_K(p)$ with the context. A first approach would be to find, for each p, a specification that would be bisimilar to the LTS engendered by $\ell_K(p)$ (in which we would not observe θ on transitions). But then RCCS would be mere progress over CCS with explicit backtracking since one would need to consider every transitions of $\ell_K(p)$ to check for bisimulation.

The question is now to see whether it is possible to obtain a characterisation of the behaviour of a lifted process $\ell_K(p)$ solely in terms of p. Intuitively, $\ell_K(p)$ being p enriched with a mechanism for escaping computations not leading to any observable actions, one might think that $\ell_K(p)$ is bisimilar to the transition system generated by those traces of p which lead to an observable action. This is almost true.

To give a precise statement, we need first a few notations and definitions. An RCCS transition as defined above is fully described by a tuple $t = \langle r, a, \Theta, r' \rangle$ where r is the *source* of t, r' its *target*, a its label and Θ its identifier. If $a \in K$ we say that t is a *commit* transition, otherwise it is a *reversible* transition. If $\Theta = \theta$ ($\Theta = \theta^-$) we say t is *forward* (*backward*). A *trace* is a sequence of composable transitions, and we write $r \rightarrow_\sigma^* s$ ($p \rightarrow_\sigma^* q$) whenever σ is an RCCS (CCS) trace with source r (p) and target s (q). A trace is said to be forward if it contains only forward transitions.

A final and key ingredient is the notion of causality between transitions in a given forward trace. For CCS this is usually defined using the so-called proof terms [BC89], but one can also use RCCS memories.

The set of memories involved in a forward transition $t = \langle r, a, \theta, r' \rangle$ is defined as $\mu(t) := \{m \in r \mid \exists a, q : \langle \theta, a, q \rangle . m \in r'\}$; this is either a singleton, if no communication happened, or a two elements set, if some did.

Definition 1 (Causality). *Let $\sigma : t_1; \ldots; t_n$ be a forward RCCS trace:*
— *t_i and t_j with $i < j$, are in* direct causality relation*, written $t_i <_1 t_j$ if there is $m \in \mu(t_i)$, $m' \in \mu(t_j)$ such that $m \sqsubset m'$; one says that t_i* causes *t_j, written $t_i < t_j$, if $t_i <_1^* t_j$.*
— *σ is said to be* causal *if for all transitions t_i with $i < n$, $t_i < t_n$; it is said to be k-*causal *if it is causal, its last transition t_n is labelled with $k \in K$, and all preceding transitions are labelled in K^c.*

One extends this terminology to CCS traces by saying a CCS trace $p \rightarrow_\sigma^* p'$ is causal, if it lifts to a *causal* trace $\ell_K(p) \rightarrow_{\sigma'}^* r'$ with $\varphi(r') = p'$.

We are now ready to state the property that explicits the effects of a commit transitions. Say a process r is *initial* if there is no possible backward transition with source r.

Proposition 1 (Committed trace). *Let σ be a RCCS trace with an initial source. Then the target of σ is also initial if and only if σ is in k-causal form.*

This property says that whenever a commit is taken, then any other process can no longer cancel an action that played a role in the transaction; this expresses the durability of the transaction.

With the notion of causal trace in place, we can define the causal compression of a process p relative to K.

Definition 2 (Relative causal compression). *Let p be a CCS process, its* causal compression relative to K*, written $\mathsf{CTS}_K(p)$, is the LTS $\langle P, p, K, \twoheadrightarrow\rangle$ where $\twoheadrightarrow_k$ is defined as $q \twoheadrightarrow_k q'$ if $q \rightarrow^*_\sigma q'$ for some k-causal trace σ.*

We are now ready to state the theorem that characterizes the behaviour of $\ell_K(p)$ in terms of the simpler process p.

Theorem 1 ([DK05]). *Let $\mathsf{TS}_K(p) := \langle R, \ell_K(p), A, \rightarrow\rangle$ be the LTS associated to the lift $\ell_K(p)$, $\mathsf{TS}_K(p) \sim \mathsf{CTS}_K(p)$.*

As said above, it is not true that $\mathsf{TS}_K(p)$ is bisimilar to the transition system of traces of p leading to observable actions, one has to be careful to restrict to causal traces. A trivial but useful rephrasing of this result is:

Corollary 1. *Let p be a CCS process, and $\mathcal{S}$ be its specification, if $\mathsf{CTS}_K(p) \sim \mathcal{S}$ then $\ell_K(p) \sim \mathcal{S}$.*

In words, this says that to check the correctness of $\ell_K(p)$ with respect to $\mathcal{S}$, it is enough to check the correctness of $\mathsf{CTS}_K(p)$.

If one goes back to the example at the beginning of this section, this says that $\ell_{\{a,b\}}((x)\big(x \mid x \mid \bar{x}.\bar{x}.a.p \mid \bar{x}.\bar{x}.b.q\big))$ is equivalent to $a.p + b.q$, as long as the causal compression of $p = (x)\big(x \mid x \mid \bar{x}.\bar{x}.a.p \mid \bar{x}.\bar{x}.b.q\big)$ relative to $\{a, b\}$ is. This is easily seen in this example, and in fact, as often in practice, $\mathsf{CTS}_K(p)$ and $\mathcal{S}$ turn out to be equal.

The interest of this fundamental property lies in the fact that the causal compression relative to K, $\mathsf{CTS}_K(p)$, is significantly smaller than the partially reversible process $\ell_K(p)$. A natural question is therefore, given a process p, to compute $\mathsf{CTS}_K(p)$. By finding an efficient way to do this, one would obtain an efficient verification procedure. This is the object of the next section.

4 Causal Compression

A first idea to extract the causal transition system of a process p is to use the LTS generated by $\ell(p)$ and screen off non causal traces. One cannot know however whether a trace can be extended into a k-causal form until a commit is effectively taken, and such an approach would likely lead to both superfluous (because lots of traces will not be causal) and redundant (because of trace equivalence) computations. A more astute approach is to look only at traces that will eventually be in a k-causal form. This requires a bottom up view of traces where one starts from commits inside a term, and then reconstructs causal traces triggering this commit by consuming its predecessors in every possible way.

However, there is no need to work directly in the syntax, and event structures [Win82] provide exactly what is needed here: a truly concurrent semantics that abstracts from the interleaving of concurrent transitions, and more importantly an explicit notion of causality. Among the various types of event structures the most often considered are prime ones, because consistent runs can be simply characterized. Yet they lead to quite large data structures.[2] Our algorithm uses instead *flow event structures (FES)* [BC89, Bou90, vGG03]. On the one hand, there is a simple inductive translation of CCS terms into FESs that incurs no computational cost; on the other hand, FES are algorithmically convenient compact forms of event structures.

We first explain how to extract the causal compression $\mathsf{CTS}_K(p)$ from the translation of p into an FES. Then we discuss computational issues such as how to make this an algorithm, and how some of the apparent computational costs can be circumvented at the level of the implementation.

4.1 Flow Event Structures

A (labelled) flow event structure is a tuple $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ where

— E is a set of *events*,
— $\prec \subseteq E \times E$ is the *flow relation* which has to be irreflexive,
— $\# \subseteq E \times E$ is the *conflict relation* which is symmetric,
— and $\lambda : E \to A$ a labelling function.

The idea is that the flow relation gives all immediate possible causes of an event, while the conflict relation indicates a conflicting choice between two events.

Definition 3. *Let $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle$ be an FES, a set $X \subseteq E$ is a* configuration *of $\mathcal{E}$, written $X \in \mathcal{C}(\mathcal{E})$, if it is:*

— conflict free*: $\# \cap (X \times X) = \emptyset$,*
— cycle free*: $\prec^* /X$ is a partial order,*
— *and* left-closed up to conflicts*: if $e \in X$ and there is $d \in E$ such that $d \prec e$ then either $d \in X$ or there exists $f \in X$ such that $f \prec e$ and $f \# d$.*

The last two conditions are the price to pay for working with FESs, and are not needed for prime ones. The first one will require some optimised structuring of the conflict relation, we'll return to this point soon.

A configuration X in $\mathcal{E}$ with $e \in X$ is *e-minimal* if $\forall e' \in X : e' \prec^* e$. The set of e-minimal configurations is denoted by $\mathcal{C}\langle \mathcal{E}, e \rangle$.

There is an easy inductive translation u unfolding any CCS process into a FES, where events correspond to communications, and configurations are those subsets of events that a trace can trigger. We recall now this translation, defined by induction on the structure of the CCS process, from [BC89].

— **(Prefix)** Let $u(p) \overset{def}{=} \langle E, \prec, \#, \lambda \rangle$ be the FES corresponding to P and let $e \notin E$, then for any prefix action α we have $u(\alpha.p) \overset{def}{=} \langle E \uplus \{e\}, \prec', \#, \lambda' \rangle$ where:

[2] Specifically in prime event structure causes of an event must be uniquely determined, and this forces duplication of the future of an event each time it is engaged in a synchronization.

- $\prec' / E \times E \overset{def}{=} \prec$ and $\forall e' \in E,\ e \prec' e'$
- $\lambda' / E \times E \overset{def}{=} \lambda$ and $\lambda(e) \overset{def}{=} \alpha$

— **(Choice)** Let $u(p) \overset{def}{=} \langle E_p, \prec_p, \#_p, \lambda_p \rangle$ and $u(q) \overset{def}{=} \langle E_q, \prec_q, \#_q, \lambda_q \rangle$ with $E_p \cap E_q = \emptyset$, we have $u(p+q) \overset{def}{=} \langle E_p \uplus E_q, \prec, \#, \lambda \rangle$ where:

- $e\#e'$ if $(e,e') \in E_p \times E_q$ or if either $e\#_p e'$ or $e\#_q e'$
- $\prec / E_p \times E_p \overset{def}{=} \prec_p$ and $\prec / E_q \times E_q \overset{def}{=} \prec_q$

— **(Parallel product)** with use the notation $e_{(e_0,e_1)}$ to denote the event resulting from the synchronization of events e_0 and e_1. We use the traditional projection $\pi_0(e_{(e_0,e_1)}) \overset{def}{=} e_0$ and $\pi_1(e_{(e_0,e_1)}) \overset{def}{=} e_1$. In addition for every event e that is not a synchronization we consider $\pi_0(e) \overset{def}{=} \pi_1(e) \overset{def}{=} e$. With these conventions, we define the unfolding $u(p \mid q) \overset{def}{=} \langle E_q \uplus E_q \uplus E, \prec, \#, \lambda \rangle$ where:

- $E \overset{def}{=} \left\{ e_{(e',e'')} \mid (e',e'') \in E_p \times E_q \ \&\ \lambda_p(e) = \overline{\lambda_q(e')} \right\}$
- $\forall e \in E_p \uplus E_q \uplus E$, $\lambda(e) \overset{def}{=} \lambda_p(e)$ if $e \in E_p$, $\lambda(e) \overset{def}{=} \lambda_q(e)$ if $e \in E_q$ and $\lambda(e) \overset{def}{=} \tau$ else.

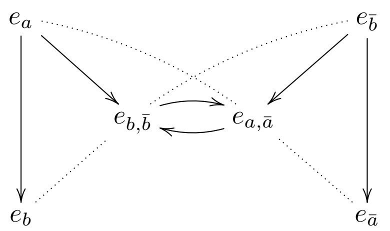
- $\forall (e,e') \in (E_p \uplus E_q \uplus E)^2$, we have $e \prec e'$ if either:

  - $\pi_0(e) \prec_p \pi_0(e')$
  - $\pi_1(e) \prec_q \pi_1(e')$

- $e\#e'$ if:

  - $\pi_i(e) = \pi_i(e')$ for some $i \in \{1,2\}$
  - $\pi_0(e)\#_p\pi_0(e')$
  - $\pi_1(e)\#_q\pi_1(e')$

Consider for instance the process $p = a.b.0 \mid \bar{b}.\bar{a}.0$. Using the above unfolding function one obtains the following FES, where arrows represent causality and dotted edges represent conflict:



$$\begin{tikzcd}
e_a \arrow[d] \arrow[dr] & & e_{\bar b} \arrow[d] \arrow[dl] \\
& e_{b,\bar b} \arrow[r, bend left] \arrow[dl, dash, dotted] & e_{a,\bar a} \arrow[l, bend left] \arrow[dr, dash, dotted] & \\
e_b & & e_{\bar a}
\end{tikzcd}$$

With the convention that $\lambda(e_\alpha) = \alpha$ and $\lambda(e_{\alpha,\bar{\alpha}}) = \tau$, for all $\alpha \in \{a, \bar{a}, b, \bar{b}\}$. It follows from Definition 3, that maximal configurations[3] of $u(p)$ are:

$$X_0 \stackrel{def}{=} \{e_a, e_b, e_{\bar{b}}, e_{\bar{a}}\} \qquad X_1 \stackrel{def}{=} \{e_a, e_{b,\bar{b}}, e_{\bar{a}}\} \qquad X_2 \stackrel{def}{=} \{e_{\bar{b}}, e_{a,\bar{a}}, e_b\}$$

The correctness of the unfolding function u is given by the following representation theorem:

Theorem 2 ([Bou90]). *Let p be a CCS process, and $\mathcal{T}_{\simeq}(p)$ stand for the traces of p quotiented by trace equivalence, then $(\mathcal{T}_{\simeq}(p), \leq)$ and $(\mathcal{C}(u(p)), \subseteq)$ are isomorphic.*

One can define a transition system out of an FES. To do this, we define $\mathcal{E}|X$, the *residual* of $\mathcal{E}$ by a configuration X in $\mathcal{C}(E)$.

Definition 4 (Residual). *Let $\mathcal{E} = \langle E, \prec, \#, \lambda\rangle$ be an FES, X be a configuration of $\mathcal{E}$, and define $X_\# := \{e \in E \mid \exists e' \in X : e'\#e\}$. The residual of E by X is $\mathcal{E}|X := \langle E', \prec', \#'\rangle$ where:*

$$E' := E \setminus (X \cup X_\#) \qquad \prec' := \prec \cap\, (E' \times E') \qquad \#' := \# \cap\, (E' \times E')$$

The LTS associated to $\mathcal{E} = \langle E, \prec, \#, \lambda\rangle$ has initial state $\mathcal{E}$, and transition relation given by $\mathcal{E}' \rightarrow_X \mathcal{E}''$ if $X \in \mathcal{C}(\mathcal{E}')$ and $\mathcal{E}'' = \mathcal{E}'|X$.

It is here that our reframing of the compression question in terms of event structures pays off, since to obtain the causal compression of the transition system above, all one has to do is to restrict labels to e-minimal configurations such that $\lambda(e) \in K$. The *causal LTS* associated to $\mathcal{E}$, written $\mathsf{CTS}_K(\mathcal{E})$, has initial state $\mathcal{E}$, and transition relation given by $\mathcal{E}' \twoheadrightarrow_k \mathcal{E}''$ if there is an event $e \in E'$ such that $\mathcal{E}' \rightarrow_X \mathcal{E}''$ with $X \in \mathcal{C}\langle \mathcal{E}', e\rangle$ and $\lambda(e) \in K$. As a consequence of the representation theorem one gets:

Lemma 1. *Let p be a CCS process, then $\mathsf{CTS}_K(p)$ and $\mathsf{CTS}_K(u(p))$ are isomorphic.*

At this point, we have an equivalent definition of $\mathsf{CTS}_K(p)$ in terms of the FES $u(p)$, and it remains to see how one can turn this definition into an algorithm. This is what we discuss now.

4.2 Algorithmic Discussion

First, the unfolding $u(p)$ is in general an *infinite* object even if we restrict to finite state processes. To keep with finite internal data structures, we require each recursive process definition to be guarded by a commit action. This seems a reasonable constraint, in that there is a priori no reason to model a transactional mechanism with a process that allows infinite forward inconclusive traces.

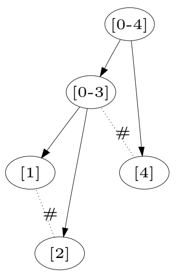
[3] A configuration X is maximal if there is no additional event e such that $e \uplus X$ is a valid configuration.

To compute $\mathsf{CTS}_K(u(p))$, we use instead of u, a *partial unfolding* u^{fin} that coincides with u except it does not unfold any recursive definition. The constraint above ensures that every commit k that is reachable by a single causal transition can be seen by this partial unfolding. Only after triggering the event corresponding to k, are the recursive calls guarded by k (if any) unfolded, and their translations by u^{fin} added to the residual of the obtained event structure. One then checks whether the obtained residual event structure is isomorphic with some obtained previously, and adds it to the state space if not. Given a process p, the algorithm to compute $\mathsf{CTS}_K(u(p))$ proceeds as follows:

0. $\mathcal{E} = \langle E, \prec, \#, \lambda \rangle := u^{fin}(p)$
1. For all $e \in E$ such that $\lambda(e) \in K$, compute the e-minimal configurations $X_e \in \mathcal{C}\langle \mathcal{E}, e \rangle$.
2. For each such X_e build the residual $\mathcal{E}|X_e$, with recursive definitions guarded by e unfolded using u^{fin}.
3. Add the transitions $\mathcal{E} \twoheadrightarrow_k \mathcal{E}|X_e$, where $k = \lambda(e)$, to the CTS under construction.
4. For each residual $\mathcal{E}|X_e$ not isomorphic to any previous one, set $\mathcal{E} := \mathcal{E}|X_e$ and goto step 1.

By the representation theorem, this algorithm will terminate as soon as $\mathsf{CTS}_K(p)$ is finite. In practice most of the isomorphism tests can be avoided by using a quite discriminative equality test between FES signatures which is linear in the number of events. Another efficiency problem one has to deal with is the internal representation of the conflict relation (which is involved in step 1 because of the conflict-free condition on configurations). In prime event structures conflict is inherited by causality, that is to say if $e\#e'$ and $e' \prec e''$, then $e\#e''$. Hence a rather compact way to represent conflict is to keep only $(e, e') \in \#$ and deduce when needed that $e\#e''$ by heredity.

We have found that a similar compact structure, which we call a conflict tree can be used for FESs. Conflict trees are built during process partial unfoldings, and result in a typically logarithmically compact representation of conflict, for a low computational cost. An example of a conflict tree is given on the right: conflicts are predicated of intervals, and $[n - m]\#[n' - m']$ means that any pair of events indexed within $\{n, \dots, m\} \times \{n', \dots, m'\}$ is in conflict.



5 Benchmark

The relative compression algorithm was implemented as a prototype in Ocaml in order to get a sense of how well our verification technique performs compared with a straight bisimulation based verification. To do so we ran several tests[4]

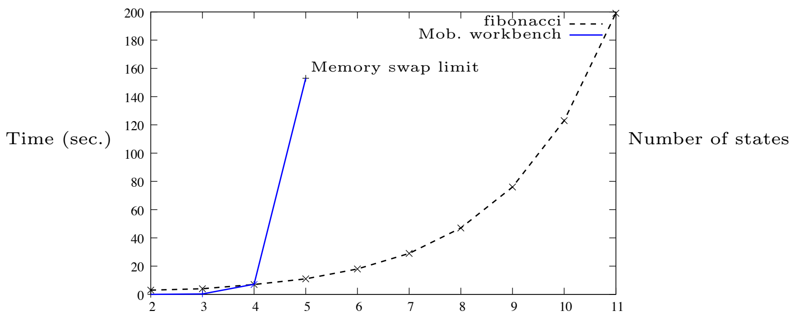
[4] Tests were made with an Intel Pentium 4 CPU 3.20GHz with 1GB of RAM.

using encodings of the dining philosophers problem. This timeless example of distributed consensus involves n philosophers eating together around a table. Each of them needs two chopsticks to start eating, and has to share them with his neighbours. When a philosopher has eaten, he releases his chopsticks after a while and goes back to the initial state. In the partial implementation, say p_{part}, once a philosopher takes a chopstick he never puts it back unless he has successfully eaten. In the fully correct one, say p_{full}, he may release chopsticks at any time (thus avoiding deadlocks). The CCS processes p_{part} and p_{full} for $n = 2$ correspond roughly to the earlier examples (1) and (2). (See [DK05] for a general definition and detailed study.)

There are two main reasons for taking the dining philosophers example. First it is a paradigmatic example of distributed consensus, so the way to solve it without access to the scheduler (by adding additional semaphores for instance) has to involve backtracking. Second, it turns out that the number of possible states of the specification is given by a Fibonacci sequence.

$$S(1) = 1 \quad S(2) = 3 \quad S(n+1) = S(n) + S(n-1)$$

Direct bisimulation test for p_{full}:



Relative causal compression using our algorithm:

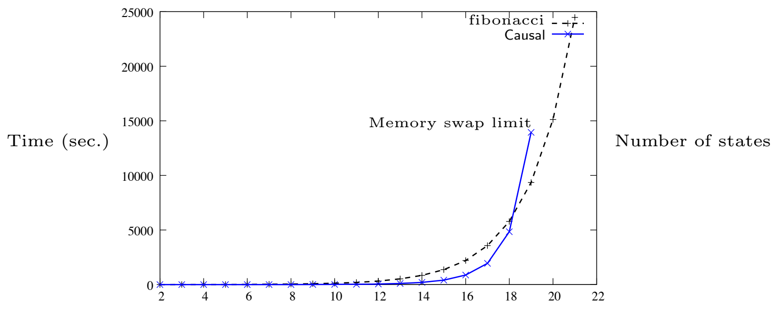


Fig. 1. Benchmark results for the dining philosophers

This is convenient in that it gives a simple means to compare the time of computation with the size of the specification state space. Verifying correctness of p_{full} using the Mobility Workbench (MWB) [VM94] (see top curve, Fig. 1) proved to be impossible beyond 5 philosophers (around 160 specification states) because of memory limitations. By using first the our prototype (see bottom curve, Fig. 1) to extract the causal transition system of p_{part}, we could verify up to 19 philosophers (around $15,000$ specification states) within a time which stayed roughly proportional to the number of states. Since $\mathsf{CTS}(p_{part})$ is in this case equal to the specification, the remaining part of the correctness proof takes negligible time (MWB needs $0.4s$ for 10 philosophers).

6 Conclusion

We have proposed a method for the verification of distributed systems which uses an algorithm of relative causal compression. The method does not always apply: the process one wants to verify must use a generic backtracking mechanism. This may seem a limitation, but it often obtains a much simpler code, and many examples of distributed transactions lend themselves naturally to this constraint. When the method does apply, however, it proves very effective as we have shown in the dining philosophers example.

State space explosion in automated bisimulation proofs is a well known phenomenon, and trace compression techniques have been proposed to avoid the redundancy created by the interleaving of transitions [BC89, GW91], and used in model-checking applications [BCDP95, AQR+04]. These compressions preserve bisimilarity, whereas our does not, and is of a completely different nature. Besides, and because our algorithm uses event structures, we also benefit from this classical kind of compression.

There is no reason why this verification method should be limited to CCS. Other concurrent models can be equipped with backtracking, and forward and backward aspects of correctness can be split there as well. Recent work extends the concept of partially reversible computations to various process algebras [PU06, PU07, LMS10, LMSS11], and it is possible to define an analogue of RCCS for the π-calculus. New advances in event structure semantics for π-calculus [VY10, CVY12] might allow to extend the causal compression algorithm, so as to cover the important case of name-passing calculi.

References

[AQR$^+$04] Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: A Model Checker for Concurrent Software. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 484–487. Springer, Heidelberg (2004)

[BC89] Boudol, G., Castellani, I.: Permutation of Transitions: An Event Structure Semantics for CCS and SCCS. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 411–427. Springer, Heidelberg (1989)

[BCDP95] Bianchi, A., Coluccini, S., Degano, P., Priami, C.: An Efficient Verifier of Truly Concurrent Properties. In: Malyshkin, V.E. (ed.) PaCT 1995. LNCS, vol. 964, pp. 36–50. Springer, Heidelberg (1995)

[Bou90] Boudol, G.: Flow Event Structures and Flow Nets. In: Guessarian, I. (ed.) LITP 1990. LNCS, vol. 469, pp. 62–95. Springer, Heidelberg (1990)

[CVY12] Crafa, S., Varacca, D., Yoshida, N.: Event Structure Semantics of Parallel Extrusion in the Pi-Calculus. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 225–239. Springer, Heidelberg (2012)

[DK04] Danos, V., Krivine, J.: Reversible Communicating Systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)

[DK05] 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)

[DKT07] Danos, V., Krivine, J., Tarissan, F.: Self-assembling trees. Electr. Notes Theor. Comput. Sci. 175(1), 19–32 (2007)

[GW91] Godefroid, P., Wolper, P.: Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. In: Larsen, K.G., Skou, A. (eds.) CAV 1991. LNCS, vol. 575, pp. 332–342. Springer, Heidelberg (1992)

[Kri06] Krivine, J.: Algèbres de Processus Réversibles. PhD thesis, Université Paris 6 & INRIA-Rocquencourt (2006)

[LMS10] 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)

[LMSS11] 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)

[Mil89] Milner, R.: Communication and Concurrency. International Series on Computer Science. Prentice Hall (1989)

[PU06] 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)

[PU07] Phillips, I., Ulidowski, I.: Reversibility and models for concurrency. Electr. Notes Theor. Comput. Sci. 192(1), 93–108 (2007)

[vGG03] van Glabeek, R., Goltz, U.: Well-behaved flow event structures for parallel composition and action refinement. Theoretical Computer Science 311 (1-3), 463–478 (2003)

[VM94] Victor, B., Moller, F.: The Mobility Workbench — a Tool for the π-Calculus. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 428–440. Springer, Heidelberg (1994)

[VY10] Varacca, D., Yoshida, N.: Typed event structures and the linear pi-calculus. Theor. Comput. Sci. 411(19), 1949–1973 (2010)

[Win82] Winskel, G.: Event Structure Semantics for CCS and Related Languages. In: Nielsen, M., Schmidt, E.M. (eds.) ICALP 1982. LNCS, vol. 140, pp. 561–576. Springer, Heidelberg (1982)

A Reversible Process Calculus and the Modelling of the ERK Signalling Pathway

Iain Phillips[1], Irek Ulidowski[2], and Shoji Yuen[3]

[1] Department of Computing, Imperial College London, England
[2] Department of Computer Science, University of Leicester, England
[3] Graduate School of Information Science, Nagoya University, Japan

Abstract. We introduce a reversible process calculus with a new feature of execution control that allows us to change the direction and pattern of computation. This feature allows us to model a variety of modes of reverse computation, ranging from strict backtracking to reversing which respects causal ordering of events, and even reversing which violates causal ordering. The SOS rules that define the operators of the new calculus employ communication keys to handle communication correctly and key identifiers to control execution.

As an application of our calculus, we model the ERK signalling pathway which delivers mitogenic and differentiation signals from the membrane of a cell to its nucleus. The proteins participating in the pathway are represented by reversible processes in such a way that the pathway's bio-chemical reactions are simply interactions between the processes.

1 Introduction

Reversing computation of a concurrent system poses a number of conceptual and technical questions. How is the forward and reverse computation performed and controlled? When reversing, in what order are computation steps undone? We answer the last question first. Consider a computation where the event a causes the event b, written $a < b$, and the event c occurs at another location independently of a and b. The three traces of this computation that preserve causality are abc, acb and cab: note that a always precedes b. There are several conceptually different ways of undoing these events. *Backtracking* is undoing in precisely the reverse order in which they happened. So, undo b undo c undo a is a backtrack of acb.

Reversing is a more general form of undoing: here events can be undone in any order as long as causality is preserved, meaning that causes cannot be undone before effects. For example, undo c undo b undo a is a reversal of acb for a, b and c as defined above. However, and quite surprisingly, there are situations where events happen, or are undone, *out of causal order*. The creation and breaking of molecular bonds between the proteins involved in the ERK signalling pathway described in Section 3 is a good example. Simplifying, let us assume that the creation of molecular bonds is represented by events a, b, c where, as above, $a < b$ and c is independent of a and b. In the ERK pathway, the molecular bonds are broken in the following order: undo a undo b undo c, which seems to undo the cause a before the effect b. Similarly, an execution of multi-threaded programs under weak memory models or under the out-of-order regime may result in traces which contradict program (thread) order; this is a result of well-known hardware or compiler optimisations. In this paper we propose a reversible process calculus in which we can model reversibility and out-of-order computation. To the best of our knowledge this is the first such calculus.

We return to the question of how to control the direction of computation. Reversible process calculi RCCS [7] and CCS with communication Keys (CCSK) [15,16] use a memory with a history of past computation and communication keys, respectively, to reverse computation so that causality is preserved. Reversible systems modelled in RCCS and CCSK choose the direction of computation spontaneously. When an execution of a fault-tolerant system encounters an error, the system recovers by undoing execution to a state where the error can be eliminated. The instruction when and how far to reverse is a part of the system's software and such reversibility control mechanism can be modelled by the rollback construct of the higher-order π-calculus [10]. In this paper we propose a different and more expressive mechanism for controlling reversibility. Its operational formulation and usefulness compares favourably with that of the rollback construct, and it allows us to model additionally out-of-order forward and reverse computation which we believe has not been done before in the process calculi setting.

Our calculus is an extension of CCSK [15,16], a reversible process calculus based on Milner's CCS [13], with prefixing by multisets of actions and with an execution control mechanism (controllers). The generalised prefixing gives us the ability to represent a loose relationship between events of out-of-order computation and, more specifically, it allows us to model more faithfully the structure and reactions of bio-chemical molecules. This form of prefixing was previously employed in [9]. Controllers permit us to manage the pattern and the direction of computation and together with the multiset prefixing they are able to model out-of-order computation (note that weaker forms of prefixing are not sufficient). It is a different form of the rollback construct of the higher-order π calculus [10].

Processes and controllers are quite strongly contrasted: processes (without controllers) can compute freely either forwards or in reverse, whereas controllers can only compute forwards (even when the process under control is reversing). We envisage a wide variety of uses for controllers, ranging from handling error recovery to providing the main focus of the computation, as in the bio-chemical example we present later.

We give SOS rules for the operators of our calculus in Section 2. The rules for reversing computation are simply symmetric versions of the forward rules. In order to manage correctly both communication and the reversing of communication we employ communication keys [15,16]. The new notion of key identifiers is introduced to mark the actions of processes that are to be performed or undone, thus giving us the ability not only to reverse specific past actions, as achieved by the rollback [10], but also to specify which forward action to compute and when to compute them. In this way, we achieve a more general mechanism for controlling computation. To illustrate this, consider a process that can perform actions a and b in parallel. We can define a controller that forces b to execute always after a, effectively setting a as the cause of b. In the standard setting, this means that reversing a must be proceeded by undoing b. However, our control mechanism gives the ability to reverse a and b 'out of order': first a and then b. Such patterns of computation, seemingly breaking the causal relationships between actions, are common in the bio-chemical setting as can be seen in our model of the ERK signalling pathway.

The usefulness of the execution control mechanism in exhibited in several examples. In Section 2.2 we consider the modelling of long-running transactions with compensations and we re-work the example from [10] of a system with complex causal dependencies between executing and reversing communications. The first example shows the need for the new key identifiers, whereas in the second example communication keys alone suffice. The second part of the paper (Section 3) is devoted solely to the modelling of the ERK signalling pathway [5,20], which delivers mitogenic and differentiation signals from the membrane of a cell to its nucleus, and how it is regulated by RKIP proteins. There, the execution control mechanism and prefixing with multisets of actions play a vital rôle.

The research on reversing process calculi can be traced back perhaps to the work by Berry and Boudol on the Chemical Abstract Machine [1]. We were inspired to look at reversible computation by, among others, the paper of Danos and Krivine on reversing CCS [9] and the subsequent [7,8]. We then proposed an alternative, more algebraic method for reversing CCS in [15,16], and recently provided both bisimulation and modal logic semantics for reversible concurrency [17,18]. Lanese, Mezzina, Schmitt and Stefani proposed a reversible version of a higher-order π calculus and equipped it with a rollback construct [11,10]. They also studied other forms of reversibility for defining programming abstractions for dependable distributed systems, and discussed the need for compensations [12]. Finally, reversible structures that compute forwards and backwards in an asynchronous manner were proposed by Cardelli and Laneve [4].

2 A Reversible Process Calculus with Execution Control

In this section we extend CCSK with an execution control mechanism which allows us to control the direction and the pattern of computation. The extended calculus is given an operational semantics and its usefulness is illustrated in several examples including long-running transactions with compensations.

2.1 CCSK

We define the (forward) actions of CCS as usual: let $\mathcal{A}$ be a set of actions a, let $\overline{a}$ be the *complement* of a, and let $\overline{\mathcal{A}} = \{\overline{a} : a \in \mathcal{A}\}$. Also, let $\overline{\overline{a}} = a$ for $a \in \mathcal{A}$. We assume that α, β range over $\mathcal{A} \cup \overline{\mathcal{A}}$, and μ, ν range over all actions,

$$\frac{\mathsf{std}(X)}{\alpha[v].X \xrightarrow{\alpha[n,v]} \alpha[n,v].X} \qquad \frac{X \xrightarrow{\mu[n,v]} X'}{\alpha[m,u].X \xrightarrow{\mu[n,v]} \alpha[m,u].X'}\, m \neq n$$

$$\frac{X \xrightarrow{\mu[n,v]} X' \quad \mathsf{fsh}[n](Y)}{X \,|\, Y \xrightarrow{\mu[n,v]} X' \,|\, Y} \qquad \frac{X \xrightarrow{\alpha[n,v]} X' \quad Y \xrightarrow{\overline{\alpha}[n,u]} Y'}{X \,|\, Y \xrightarrow{\tau[n]} X' \,|\, Y'}$$

$$\frac{X \xrightarrow{\mu[n,v]} X' \quad \mathsf{std}(Y)}{X + Y \xrightarrow{\mu[n,v]} X' + Y} \qquad \frac{X \xrightarrow{\mu[n,v]} X'}{X \backslash A \xrightarrow{\mu[n,v]} X' \backslash A}\, \mu, \overline{\mu} \notin A \qquad \frac{X \xrightarrow{\mu[n,v]} X'}{X[f] \xrightarrow{f(\mu)[n,v]} X'[f]}$$

$$\frac{\mathsf{std}(X)}{\alpha[n,v].X \overset{\alpha[n,v]}{\rightsquigarrow} \alpha[v].X} \qquad \frac{X \overset{\mu[n,v]}{\rightsquigarrow} X'}{\alpha[m,u].X \overset{\mu[n,v]}{\rightsquigarrow} \alpha[m,u].X'}\, m \neq n$$

$$\frac{X \overset{\mu[n,v]}{\rightsquigarrow} X' \quad \mathsf{fsh}[n](Y)}{X \,|\, Y \overset{\mu[n,v]}{\rightsquigarrow} X' \,|\, Y} \qquad \frac{X \overset{\alpha[n,v]}{\rightsquigarrow} X' \quad Y \overset{\overline{\alpha}[n,u]}{\rightsquigarrow} Y'}{X, Y \overset{\tau[n]}{\rightsquigarrow} X' \,|\, Y'}$$

$$\frac{X \overset{\mu[n,v]}{\rightsquigarrow} X' \quad \mathsf{std}(Y)}{X + Y \overset{\mu[n,v]}{\rightsquigarrow} X' + Y} \qquad \frac{X \overset{\mu[n,v]}{\rightsquigarrow} X'}{X \backslash A \overset{\mu[n,v]}{\rightsquigarrow} X' \backslash A}\, \mu, \overline{\mu} \notin A \qquad \frac{X \overset{\mu[n,v]}{\rightsquigarrow} X'}{X[f] \overset{f(\mu)[n,v]}{\rightsquigarrow} X'[f]}$$

Fig. 1. Forward and reverse SOS rules

namely $\mathsf{Act} = \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$, where $\tau \notin \mathcal{A}$ is the silent action and $\overline{\overline{\tau}} = \tau$. Let $\mathcal{K}$ be an infinite set of *communication keys* (or just *keys* for short), ranged over by k, m, n. And, let $\mathcal{I}$ be an infinite set of key identifiers, ranged over by v, u, w. We also have a set of process identifiers $\mathcal{PI}$, with typical elements S, T, and a set of variables, ranged over by X, Y. $\mathcal{PI}$ contains the deadlocked process $\mathbf{0}$.

The syntax of CCSK is given below, where $A \subseteq \mathsf{Act} \setminus \{\tau\}$ and $f : \mathsf{Act} \to \mathsf{Act}$ with $f(\tau) = \tau$. The set of CCSK closed terms is $\mathcal{P}$, and we shall refer to closed terms as processes. We let P, Q to range over processes.

$$P ::= X \;\mid\; S \;\mid\; \alpha[v].P \;\mid\; \alpha[n,v].P \;\mid\; P + Q \;\mid\; P|Q \;\mid\; P\backslash A \;\mid\; P[f]$$

The prefixing with forward actions operator is $\alpha[v].X$ where v is a key identifier and is optional. Each $\{\alpha, \overline{\alpha}\}$ (and $\{\alpha, \overline{\alpha}, \underline{\alpha}, \underline{\overline{\alpha}}\}$ in Section 2.2) has a set of key identifiers associated with it, and we assume that all such sets are disjoint. Prefixing with past actions has the form $\alpha[n,v].X$ where n is the specific key for performing this α and v is drawn from the set of key identifiers for α, and may be omitted if it plays no rôle (but the key n must occur). There is no prefixing with τ. We often omit trailing $\mathbf{0}$s so, for example, $a.\mathbf{0}$ is written as a.

$X \mid Y$ represents two systems X and Y that can perform actions or reverse actions on their own or they can interact with each other on complementary actions, for example a and $\overline{a}$. The choice, restriction and relabelling operators, namely '$\cdot + \cdot$', '$\cdot \setminus A$' and '$\cdot[f]$', are as in CCS except that $+$ is now static in process terms. Each process identifier S has a defining equation $S \overset{\mathrm{df}}{=} P$.

The SOS forward and reverse SOS rules for CCSK are given in Figure 1. Note that the reverse rules are simply the reversals of the forward rules. We associate with each term X the set of its keys, written as $\mathsf{keys}(X)$. A term X is standard, written $\mathsf{std}(X)$, if it contains no prefixing with past actions. A key n is fresh in Y, written $\mathsf{fsh}[n](Y)$, if n is not used in Y.

Structural congruence $\equiv$ on terms is defined by $X \,|\, Y \equiv Y \,|\, X$, $X \,|\, (Y \,|\, Z) \equiv (X \,|\, Y) \,|\, Z$ and $X \,|\, \mathbf{0} \equiv X$. Also, $X + Y \equiv Y + X$, $X + (Y + Z) \equiv X + (Y + Z)$ and $X + \mathbf{0} = X$. And $S \equiv P$ for all S and P such that $S \overset{\mathrm{df}}{=} P$. We also have the Structural Congruence Rule:

$$\frac{X \equiv Y \quad Y \overset{\mu[n,v]}{\rightarrow} Y' \quad Y' \equiv X'}{X \overset{\mu[n,v]}{\rightarrow} X'}$$

Example 1. In CCSK we keep track of the identities of actions that communicate so that when we reverse we undo the correct past actions. Consider $P \overset{\mathrm{df}}{=} (a \,|\, a.c \,|\, \overline{a} \,|\, \overline{a}.e) \backslash a$. Here the restriction of a prevents a and $\overline{a}$ being performed except as part of a communication. Suppose that a communicates with $\overline{a}$ and then $a.c$ with $\overline{a}.e$. In CCSK we write this as follows:

$$P \equiv \overset{\tau[m]}{\rightarrow} (a[m] \,|\, a.c \,|\, \overline{a}[m] \,|\, \overline{a}.e) \backslash a \overset{\tau[n]}{\rightarrow} (a[m] \,|\, a[n].c \,|\, \overline{a}[m] \,|\, \overline{a}[n].e) \backslash a$$

Note that the process $a[m] \,|\, a.c \,|\, \overline{a}[m] \,|\, \overline{a}.e$ cannot regress by reversing $a[m]$ alone because key m is not fresh in $a.c \,|\, \overline{a}[m] \,|\, \overline{a}.e$. The fact that m appears in $a.c \,|\, \overline{a}[m] \,|\, \overline{a}.e$ which is in parallel with $a[m]$ proves that the processes communicated with a and $\overline{a}$ rather than performed them independently.

Our notation does not allow us to backtrack by undoing a different pair of actions, but clearly we can change the order of reversing $\tau[m]$ and $\tau[n]$:

$$(a[m] \,|\, a[n].c \,|\, \overline{a}[m] \,|\, \overline{a}[n].e) \backslash a \overset{\tau[m]}{\rightsquigarrow} (a \,|\, a[n].c \,|\, \overline{a} \,|\, \overline{a}[n].e) \backslash a \overset{\tau[n]}{\rightsquigarrow} \equiv P$$

CCSK processes are fully reversible because the reverse SOS rules in Figure 1 are obtained by simply reversing the forward SOS rules in Figure 1 [15,16]. We have $P \overset{\mu[n,v]}{\rightarrow} Q$ iff $Q \overset{\mu[n,v]}{\rightsquigarrow} P$ for all processes P, Q and all $\mu \in \mathsf{Act}, n \in \mathcal{K}, v \in \mathcal{I}$. Moreover, CCSK is a conservative extension of CCS [15,16].

2.2 Execution Control Operator

We add a new operator '$\cdot\langle\cdot\rangle$' to CCSK for controlling the execution of processes. We shall need new actions that control the reversing of the forward actions: $\underline{a}$ and $\underline{\overline{a}}$ prompt reversing of the past versions of a and $\overline{a}$ respectively. Thus, we have two further sets $\underline{\mathcal{A}}$ and $\underline{\overline{\mathcal{A}}}$. We let $\underline{\alpha}, \underline{\beta}$ range over $\underline{\mathcal{A}} \cup \underline{\overline{\mathcal{A}}}$, κ range over $\mathcal{A} \cup \overline{\mathcal{A}} \cup \underline{\mathcal{A}} \cup \underline{\overline{\mathcal{A}}}$ and, from now on, we let μ, ν range over all actions, namely $\mathsf{Act} = \mathcal{A} \cup \overline{\mathcal{A}} \cup \underline{\mathcal{A}} \cup \underline{\overline{\mathcal{A}}} \cup \{\tau\}$.

$X\langle Y\rangle$ is the process X controlled by Y. The behaviour of $X\langle Y\rangle$ is a subset of the behaviour of X as prescribed by Y according to the rules in Figure 2 which are in the Ordered SOS format [19,14]. Before we explain these rules and how the control operator works, we define control terms and update the definition of processes. The syntax for control terms is given below. Closed control terms, or simply control terms or controllers, are ranged over by C, D.

$$C ::= X \;\mid\; c \;\mid\; \kappa[v].C \;\mid\; \kappa[n,v].C \;\mid\; C + D \;\mid\; C|D$$

Terms c are typical elements of a set of control identifiers. By abuse of notation we shall often use C, D for control identifiers. Every control identifier has a defining equation $c \overset{\mathrm{df}}{=} C$; we extend the definition of $\equiv$ by $c \equiv C$ for all c, C such that $c \overset{\mathrm{df}}{=} C$. The SOS rules for the operators of control terms are the standard SOS rules for CCS, except that we have prefixing with new actions and prefixing carries keys or key identifiers. Note that the prefixing and $+$ operators are dynamic operators as in CCS. Thus, controllers compute forwards only so, for example, $\kappa[v].C \overset{\kappa[k,v]}{\rightarrow} C$, for some k, and $\kappa[v].C + D \overset{\kappa[k,v]}{\rightarrow} C$.

The class of processes is extended to include terms $P\langle C\rangle$ for all P and C.

$$\text{(cf1)}\ \frac{X \overset{\alpha[n,v]}{\rightarrow} X' \quad Y \overset{\alpha[n,v]}{\rightarrow} Y'}{X\langle Y\rangle \overset{\alpha[n,v]}{\rightarrow} X'\langle Y'\{n,v/v\}\rangle} \quad > \quad \text{(cf2)}\ \frac{X \overset{\beta[m,u]}{\rightarrow} X' \quad Y \overset{\alpha'[k,w]}{\rightarrow} Y'}{X\langle Y\rangle \overset{\beta[m,u]}{\rightarrow} X'\langle Y\{m,u/u\}\rangle}$$

$$\text{(cr1)}\ \frac{X \overset{\alpha[n,v]}{\rightsquigarrow} X' \quad Y \overset{\underline{\alpha}[n,v]}{\rightarrow} Y'}{X\langle Y\rangle \overset{\alpha[n,v]}{\rightsquigarrow} X'\langle Y'\{v/n,v\}\rangle} \quad > \quad \text{(cr2)}\ \frac{X \overset{\beta[m,u]}{\rightsquigarrow} X' \quad Y \overset{\underline{\alpha'}[k,w]}{\rightarrow} Y'}{X\langle Y\rangle \overset{\beta[m,u]}{\rightsquigarrow} X'\langle Y\{u/m,u\}\rangle}$$

Fig. 2. SOS rules for the control operator

Returning to Figure 2, the notation (cf1) $>$ (cf2) means that (cf2) can be applied to derive a transition of $P\langle C\rangle$ if no rules higher in the ordering $>$ can be applied, namely the rules (cf1) are not applicable for all α, n, v. So, if C can perform any forward $\alpha'[k,w]$ and P cannot perform any of the forward actions $\alpha[n,v]$ of C, then (cf2) can be used to derive $P\langle C\rangle \overset{\beta[m,u]}{\rightarrow} P'\langle C\{m,u/u\}\rangle$ if $P' \overset{\beta[m,u]}{\rightarrow} P'$. We note that $C\{m,u/u\}$ means that every occurrence of u in C is replaced with m,u. The controller keeps track of which actions to reverse or to perform by recording keys and key identifiers shared with its process.

Actions $\underline{\alpha}$ of the controller require X to reverse until α is undone. The rules (cr1) and (cr2) play the dual rôle to (cf1) and (cf2). Here, we replace the key and the key identifier in the controller with the key identifier alone, thus wiping out the record of the keys of the reversed transitions.

Terms such as $a[v].b[v]$ are not well formed as different actions cannot share identifiers. Some well formed terms are not very useful, for example only the first a can execute in $a[v].a[v]$.

Example 2. Consider $P \overset{\mathrm{df}}{=} a.a.b.b$. If $C' \overset{\mathrm{df}}{=} b.\underline{a}.b$ then $P\langle C'\rangle$ computes until after the first b of P, then reverses until the second a is undone and finally it computes until after the first b. If we wish to compute or reverse other occurrences of actions a and b in P, for example the first a and the second b, then we use key identifiers. The controller $C \overset{\mathrm{df}}{=} b[v].\underline{a}[u].b[v]$ achieves this provided that the appropriate actions a and b in P are marked with u respectively v. Let $P \overset{\mathrm{df}}{=} a[u].a.b.b[v]$. Then, using rule (cf2), we obtain $P\langle C\rangle \xrightarrow{a[1,u]} a[1,u].a.b.b[v]\ \langle b[v].\underline{a}[1,u].b[v]\rangle$. Note that prefixing with $\underline{a}[u]$ in the controller has been updated with the key 1. After another forward a and a b, we use rule (cf1) to perform $b[4,v]$ ($b[v]$ with the key 4); note that the second $b[v]$ in C is updated to $b[4,v]$:

$$\xrightarrow{a[2]}\xrightarrow{b[3]}\xrightarrow{b[4,v]} a[1,u].a[2].b[3].b[4,v]\ \langle \underline{a}[1,u].b[4,v]\rangle.$$

Then we reverse until we have undone the $a[1,u]$ using (cr2) and (cr1):

$$\overset{b[4,v]}{\rightsquigarrow}\overset{b[3]}{\rightsquigarrow}\overset{a[2]}{\rightsquigarrow}\overset{a[1,u]}{\rightsquigarrow} a[u].a.b.b[v]\ \langle b[v]\rangle.$$

Note that this reversal wipes out all the keys. Finally, we can compute forwards.

The control operator is very expressive. Consider a process P. Process $P\langle \mathbf{0}\rangle$ behaves as $\mathbf{0}$. If a is not in the sort of P (the set of actions that P can ever perform), then $P\langle a\,|\,\underline{a}\rangle$ and $P\langle a+\underline{a}\rangle$ behave exactly as P. If we allowed prefixing with τ in control terms, then $C \overset{\mathrm{df}}{=} \tau.C$ would force communications in P thus acting as the restriction operator of CCS.

The control operator can be used to make the forward actions of processes irreversible. Consider $a.b$ and $C \overset{\mathrm{df}}{=} b.\underline{b}.C$. Then $(a.b)\langle C\rangle \xrightarrow{a[1]} (a[1].b)\langle C\rangle \overset{a[1]}{\not\rightsquigarrow}$ since C insists on computing forwards with b. Also we can find examples where $Q \overset{\mu[n,v]}{\rightsquigarrow} P$ holds but there is no Q' such that $P \xrightarrow{\mu[n,v]} Q'$. Consider $(a \mid b)\langle C\rangle$ where $C \overset{\mathrm{df}}{=} a.b.\underline{a}.\underline{b}.C$. We have $(a\,|\,b)\langle C\rangle \xrightarrow{a[1]}\xrightarrow{b[2]} (a[1]\,|\,b[2])\langle \underline{a}.\underline{b}.C\rangle \overset{a[1]}{\rightsquigarrow} (a\,|\,b[2])\langle \underline{b}.C\rangle \overset{b[2]}{\rightsquigarrow} (a\,|\,b)\langle C\rangle$ but not $(a\,|\,b)\langle C\rangle \xrightarrow{b[2]} Q'$ since C insists on performing a first. This is an example of a computation that reaches a state after a reversal that cannot be reached by computing forwards only.

Example 3. A long-running transaction consists of many atomic steps which are represented here by a. A step may succeed, and then it is followed by the next step (or success s; this action never fails), or fail which results in the action f. When all steps are successfully completed the transaction succeeds and is irreversible. When f takes place all steps a performed successfully need to be undone. The transaction is modelled by T_0 as follows:

$$T_i \overset{\mathrm{df}}{=} a[v_{i+1}].T_{i+1} + f[u] \quad \text{for } 0 \le i < n, \qquad T_n \overset{\mathrm{df}}{=} s$$

The required controller is $C \overset{\mathrm{df}}{=} a[v_1].(a[v_n]+f[u].\underline{a}[v_1].C)+f[u].\underline{f}[u].C$. Let us see how $T_0\langle C\rangle$ computes. If the transaction fails immediately by performing $f[1,u]$, then this triggers the outermost action f in the controller:

$$T_0\langle C\rangle \xrightarrow{f[1,u]} (a[v_1].T_1 + f[1,u])\ \langle \underline{f}[1,u].C\rangle$$

The controller then requires undoing f: $\overset{f[1,u]}{\rightsquigarrow} (a[v_1].T_1 + f[u].\mathbf{0})\langle C\rangle \equiv T_0\langle C\rangle$.

If the transaction does not fail immediately, then $a[1, v_1]$ is performed (and is matched by the controller):

$$T_0\langle C\rangle \xrightarrow{a[1,v_1]} (a[1, v_1].T_1 + f[u])\ \langle a[v_n] + f[u].\underline{a}[1, v_1].C\rangle$$

The process then computes until the last step $a[v_n]$, or else it fails in the meantime by performing $f[k, u]$, for some key k. This is matched by the controller which becomes $\underline{a}[1, v_1].C$. Next, the execution is reversed until $a[1, v_1]$ is undone, thus returning to the original configuration: $\cdots \overset{a[1,v_1]}{\rightsquigarrow} T_0\langle C\rangle$.

In some transactions it may not be necessary to undo all successful steps a in case of failure. If these steps can be grouped into sequences, then only the steps of the most recently performed sequence need undoing. Let there be two such sequences, the first finishing with a_k with $2 \le k$ and $k + 2 \le n$. Then the controller D is defined as follows:

$$\begin{aligned} D &\overset{\mathrm{df}}{=} a[v_1].(a[v_k].D' + f[u].\underline{a}[v_1].D) + f[u].\underline{f}[u].D \\ D' &\overset{\mathrm{df}}{=} a[v_{k+1}].(a[v_n] + f[u].\underline{a}[v_{k+1}].D') + f[u].\underline{f}[u].D' \end{aligned}$$

We easily can check that $T_0\langle D\rangle$ works properly.

Example 4. Assume a long-running transaction has a compensation K which is triggered by action c and which completes with s, where both c and s never fail. We model this by adjusting the definition of T_1 from Example 3 and leaving other T_is unchanged: $T_1 \overset{\mathrm{df}}{=} a[v_1].T_2 + f[u] + c.K$. The controller is $C \overset{\mathrm{df}}{=} a[v_1].(a[v_n] + f[u].\underline{a}[v_1].c.s) + f[u].\underline{f}[u].c.s$. When a failure occurs the controller reverses all actions a that took place so far (or just the initial f), triggers c and insists that the compensation K computes forwards by demanding s.

Example 5. Consider the following system taken from [10], where (undo a) forces reversing of computation until a is undone (similarly for (undo b)).

$$(\overline{a} \,|\, a.\overline{d} \,|\, c.(\text{undo } a) \ \ | \ \ \overline{b} \,|\, b.\overline{c} \,|\, d.(\text{undo } b)) \setminus \{a, d, c, b\}$$

Inspecting the causal dependencies between actions, we note that undoing a is possible only after $c, \overline{c}$ have communicated, which requires $b, \overline{b}$ to communicate first. And, of course, after $a, \overline{a}$ have happened. If in the meantime a communication on $d, \overline{d}$ takes place, it disables undoing a. This is because a causes $\overline{d}$ and the cause a cannot be undone prior to undoing the effect d. Causal dependencies between these communications are shown in the left-hand diagram in Figure 3, where $b \to \overline{c}$ means that a communication involving b must precede a communication involving $\overline{c}$. The dashed line labelled 'undo a' indicates that reversing the communication involving a is possible only after the communications involving the actions that appear above the line (here $a, b, \overline{c}$) have taken place.

In our calculus, the left component is $\overline{a} \mid (a.\overline{d} \mid c)\langle(c.\underline{a}.\underline{e}) \mid e\rangle$. Since e cannot happen, the controller $(c.\underline{a}.\underline{e}) \mid e$ requires that 'after a forward c reverse a and

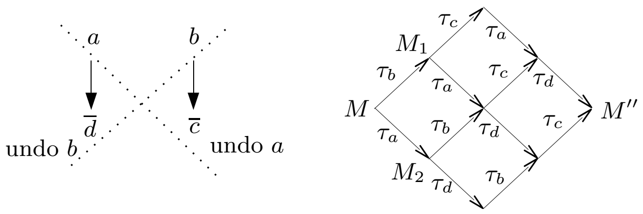


Fig. 3. Example 5

keep reversing, or independently compute forward'. Since all actions are distinct there is no need here for key identifiers. The system is

$$M \overset{\mathrm{df}}{=} (\overline{a} \,|\, (a.\overline{d} \,|\, c)\langle (c.\underline{a}.\underline{e}) \,|\, e\rangle \;|\; \overline{b} \,|\, (b.\overline{c} \,|\, d)\langle (d.\underline{b}.\underline{e}) \,|\, e\rangle) \setminus \{a,d,c,b\}.$$

In order to make transitions representing communications more readable we shall decorate labels τ with action labels: we shall write, for example, $\tau_b[1]$ instead of $\tau[1]$ for the communication on $b, \overline{b}$ with key 1. We shall also omit restriction. A communication on $b, \overline{b}$ leads to

$$M \overset{\tau_b[1]}{\rightarrow} \overline{a} \,|\, (a.\overline{d} \,|\, c)\langle (c.\underline{a}.\underline{e}) \,|\, e\rangle \;|\; \overline{b}[1] \,|\, (b[1].\overline{c} \,|\, d)\langle (d.\underline{b}[1].\underline{e}) \,|\, e\rangle \equiv M_1$$

Then we perform communications involving a and then c:

$$M_1 \overset{\tau_a[2]}{\rightarrow}\overset{\tau_c[3]}{\rightarrow} \overline{a}[2] \;|\; (a[2].\overline{d} \,|\, c[3])\langle (\underline{a}[2].\underline{e}) \,|\, e\rangle \;|\; \overline{b}[1] \;|\; (b[1].\overline{c}[3] \,|\, d)\langle (d.\underline{b}[1].\underline{e}) \,|\, e\rangle.$$

Next, a communication on $d, \overline{d}$ can take place or we can undo the communication involving a. Note that although the communication on b took place, it cannot be undone at this point since the communication on d has not taken place yet.

Consider a communication on $d, \overline{d}$:

$$\overset{\tau_d[4]}{\rightarrow} \overline{a}[2] \;|\; (a[2].\overline{d}[4] \,|\, c[3])\langle (\underline{a}[2].\underline{e}) \,|\, e\rangle \;|\; \overline{b}[1] \;|\; (b[1].\overline{c}[3] \,|\, d[4])\langle (\underline{b}[1].\underline{e}) \,|\, e\rangle) \equiv M''$$

The right-hand diagram in Figure 3 shows other sequences of communications involving a, b, c, d from M to M''. Controllers of M'' ask to undo a and undo b. But, since both d and c have now taken place, a and b can be reversed only after reversing other actions. Overall, our mechanism for controlling execution works well with this example and its operational formulation is simpler than the formulation of the rollback construct [10].

We finish this section with a remark on suitable behavioural equivalences and modal logics for our reversible calculus. A reverse interleaving bisimulation [15,16,18], which extends the standard bisimulation [13] with reverse transitions, seems a suitable behavioural equivalence. Also, a reverse pomset bisimulation may be very useful [18] as it talks directly about forward and reverse behaviour in terms of pomsets (partially ordered multisets) of actions. Event

Identifier Logic [17], a modal logic with both forward and reverse modalities, is the appropriate logic for our calculus since it characterises the mentioned above equivalences and many safety properties, such as precedence and exception, are naturally expressible with reverse modalities.

3 The ERK Signalling Pathway

The ERK signalling pathway is a realistic example of computation that comprises forward and reverse steps where some of the reverse steps violate the causal ordering established by the forward steps. We show how the new execution control and prefixing with multisets of actions allow us to represent naturally this form of out-of-order reversible computation. Signalling pathways were modelled more fully by PEPA [3] and by rule-based languages BioNetGen [2] and Kappa [6]. We shall comment on the PEPA model below.

We shall now define prefixing with multisets of actions. The actions of a given multiset of actions can execute in any order, and the computation progresses to the next multiset of actions only if all of the actions from the first multiset have taken place. Process terms are extended with $(\alpha[v], s).P$ and $(\alpha[n, v], s).P$ where s is a sequence of any actions or past actions. s' is a typical sequence consisting entirely of past actions. For simplicity, we do not allow prefixing with multisets of actions in control terms. The SOS rules are as follows:

$$\frac{\mathsf{std}(X)}{(\alpha[v], s).X \overset{\alpha[n,v]}{\rightarrow} (\alpha[n, v], s).X} \qquad \frac{X \overset{\mu[n,v]}{\rightarrow} X' \quad \mathsf{fsh}[n](s')}{(s').X \overset{\mu[n,v]}{\rightarrow} (s').X'}$$

$$\frac{\mathsf{std}(X)}{(\alpha[n, v], s).X \overset{\alpha[n,v]}{\rightsquigarrow} (\alpha[v], s).X} \qquad \frac{X \overset{\mu[n,v]}{\rightsquigarrow} X' \quad \mathsf{fsh}[n](s')}{(s').X \overset{\mu[n,v]}{\rightsquigarrow} (s').X'}$$

The Ras/Raf-1/MEK/ERK signalling pathway (ERK pathway for short) delivers mitogenic and differentiation signals from the membrane of a cell to its nucleus. This pathway is regulated by the protein RKIP. We borrow the description of the pathway and its reactions from [5,20].

The ERK pathway is spatially organised in such a way that a signal that arrives at the cell's membrane can be transmitted to the cell's nucleus via a cascade of reactions that involve proteins Ras, Raf-1, MEK and ERK. Initially, a G protein Ras is activated near a receptor on the cell's membrane. Ras then activates a kinase Raf-1 which becomes Raf*-1 (represented here by F). We shall not model Ras and its reactions here. Raf*-1 can then activate the MEK protein (M here) which gets phosphorylated to become pM. Or, this binding of Raf*-1 to MEK can be inhibited by RKIP, which binds to Raf*-1; we shall return to this sequence of reactions below. The phosphorylated MEK (pM) then activates ERK protein (E here) which, in turn becomes phosphorylated (represented by pE). Finally, at the end this cascade pE can translocate to the nucleus and pass the signal. Or, it binds to RKIP thus deactivating it temporarily (see below).

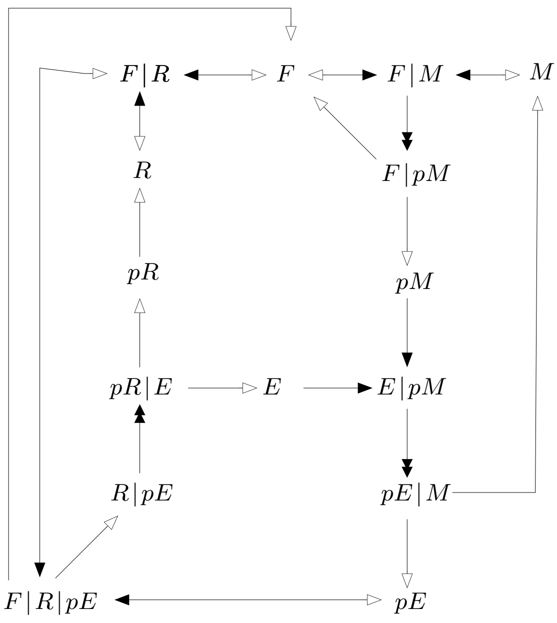


Fig. 4. The ERK pathway

When RKIP binds Raf*-1 and thus inhibits the activation of MEK, the resulting complex binds to a phosphorylated ERK (pE). Then the complex breaks releasing F, which can get involved in the cascade, E and a phosphorylated R.

Figure 4 represents the described reactions. A black-headed arrow represents a reaction that binds two molecules into a complex molecule, an open-headed arrow represents a reaction that breaks a complex into its component molecules and a bi-directional arrow represents a pair of forwards/reverse reactions: a binding and unbinding. A two-headed arrow represents a reaction that involves phosphorylation/de-phosphorylation of its reactants. The nodes in the diagram are the molecules or complexes of molecules.

We note that the ERK pathway was previously modelled in the setting of the stochastic process algebra PEPA in [3]. There, the states of the pathway as in Figure 4 are represented as indivisible processes so, for example, $F \mid R \mid pE$ is represented by a single process and not as a composition of three separate processes. These processes perform forward actions that represent creation and breaking of bonds, and the system evolves from one state to another via multi-way synchronisation of these actions. The transitions are timed and their durations are expressed as exponentially distributed random variables.

We represent the individual molecules of the ERK pathway as processes, for example F, M, E and R, and the pathway is modelled by a composition of these processes. The reactions between the molecules are represented by forward and reverse synchronisations between processes. We define F, M, E and R as follows using the new multiset prefixing operator (key identifiers are not necessary here):

$$F \overset{\mathrm{df}}{=} \overline{a}.F' \qquad M \overset{\mathrm{df}}{=} (a,p,\overline{c}).M' \qquad E \overset{\mathrm{df}}{=} (c,p,\overline{b},\overline{n}).E' \qquad R \overset{\mathrm{df}}{=} (a,b,p).R'$$

We also have molecules $P \overset{\mathrm{df}}{=} \overline{p}.P'$ which represent phosphate groups that bind with M, E and R and phosphorylate them. These phosphorylated molecules are denoted by pM, pE and pR respectively. The ERK pathway is

$$(F\,|\,M\,|\,E\,|\,P\,|\,R|pE) \setminus \{a,b,c,p\}$$

where we have a single copy of each molecule F, M, E, P, R and pE. Next, we list those synchronisations between the processes of the pathway that represent valid reactions; there are many other synchronisations that have no bio-chemical meaning and we shall see later how controllers can be employed to prune them.

The molecule F can bind with M and start a signal cascade or it can bind with a copy of inhibitor R. In order to show how F is released from the control of R we have included a copy of a phosphorylated ERK (pE). (A more realistic model would be a composition of a large numbers of copies of F, M, E, R, P, pM, pE and pR). These reactions start two alternative sequences of reactions, which we shall call the *cascade* and *regulation* sequences. We consider the cascade sequence first. For simplicity we omit restriction from now on. The binding of F and M is reversible; it is represented by blue arrows in Figure 4. The system evolves to

$$\xrightarrow{\tau_a[1]} \overline{a}[1].F' \;|(a[1],p,\overline{c}).M' \;|E\,|\,P\,|\,R|pE$$

where transition $\tau_a[1]$ indicates that a binding between $\overline{a}$ of F and a of M took place and $\overline{a}$ and a were marked with 1. Note that this binding can be immediately reversed.

$$\overset{\tau_a[1]}{\rightsquigarrow} \overline{a}.F'\,|\,(a,p,\overline{c}).M' \;|E\,|\,P\,|\,R \;\equiv F\,|\,M\,|\,E\,|\,P\,|\,R|pE$$

M gets phosphorylated and then releases F by reversing the binding on a:

$$\xrightarrow{\tau_p[2]} \overline{a}[1].F'\,|\,(a[1],p[2],\overline{c}).M'\,|\overline{p}[2].P'\,|\,E\,|\,R\,|pE$$
$$\overset{\tau_a[1]}{\rightsquigarrow} \overline{a}.F'\,|\,(a,p[2],\overline{c}).M'\,|\overline{p}[2].P'\,|\,E\,|\,R\,|pE \equiv F\,|\,(a,p[2],\overline{c}).M'\,|\overline{p}[2].P'\,|\,E\,|\,R\,|pE$$

Then, pM (which is $(a,p[2],\overline{c}).M'\,|\overline{p}[2].P'$) binds with E and phosphorylates it; M is released and pE is ready to convey the signal to the cell's nucleus:

$$\xrightarrow{\tau_c[3]} F\,|\,(a,p[2],\overline{c}[3]).M'\,|\overline{p}[2].P'\,|\,(c[3],p,\overline{b},\overline{n}).E' \;|R\,|pE$$
$$\overset{\tau_p[2]}{\rightsquigarrow} F\,|\,(a,p,\overline{c}[3]).M'\,|\,(c[3],p,\overline{b},\overline{n}).E' \;|P\,|\,R\,|pE$$
$$\xrightarrow{\tau_p[4]} F\,|\,(a,p,\overline{c}[3]).M'\,|\,(c[3],p[4],\overline{b},\overline{n}).E'\,|\overline{p}[4].P'\,|\,R\,|pE$$
$$\overset{\tau_c[3]}{\rightsquigarrow} F\,|\,(a,p,\overline{c}).M'\,|\,(c,p[4],\overline{b},\overline{n}).E'\,|\overline{p}[4].P'\,|\,R\,|pE$$

The last process is $\equiv$ equivalent to $F \,|\, M \,|\, (c, p[4], \overline{b}, \overline{n}).E' \,|\, \overline{p}[4].P' \,|\, R | pE$ which is $\equiv$ equivalent to $F \,|\, M | pE \,|\, R | pE$. Now, the newly created pE can communicate the signal with the nucleus via action $\overline{n}$ (we do not show this reaction). Note that there is now an extra copy of pE created out of E and P.

We return to the regulation sequence. We assume the binding in pE has the key 8. Instead of combining with M, the protein F can be inhibited by binding with R; this reaction is immediately reversible. Then the $R \,|\, F$ complex binds with pE. The system $F \,|\, M \,|\, E \,|\, P \,|\, R | pE \equiv R \,|\, F \,|\, pE \,|\, M \,|\, E \,|\, P$ evolves as follows:

$$\xrightarrow{\tau_a[5]} (a[5], b, p).R' \,|\, \overline{a}[5].F' \;\; | pE \,|\, M \,|\, E \,|\, P$$

$$\xrightarrow{\tau_b[6]} (a[5], b[6], p).R' \,|\, \overline{a}[5].F' \,|\, (c, p[8], \overline{b}[6], \overline{n}).E' \,|\, \overline{p}[8].P' \,|\, M \,|\, E \,|\, P$$

Next, F is released and then pE phosphorylates R:

$$\overset{\tau_a[5]}{\rightsquigarrow}\equiv F \,|\, (a, b[6], p).R' \,|\, (c, p[8], \overline{b}[6], \overline{n}).E' \,|\, \overline{p}[8].P' \,|\, M \,|\, E \,|\, P$$

$$\overset{\tau_p[8]}{\rightsquigarrow} F \,|\, (a, b[6], p).R' \,|\, (c, p, \overline{b}[6], \overline{n}).E' \,|\, \overline{p}.P' \,|\, M \,|\, E \,|\, P$$

$$\xrightarrow{\tau_p[7]} F \,|\, (a, b[6], p[7]).R' \,|\, (c, p, \overline{b}[6], \overline{n}).E' \,|\, \overline{p}[7].P' \,|\, M \,|\, E \,|\, P$$

Finally, E and pR are disassociated and R is de-phosphorylated:

$$\equiv\xrightarrow{\tau_b[6]} F \,|\, (a, b, p[7]).R' \,|\, \overline{p}[7].P' \,|\, (c, p, \overline{b}, \overline{n}).E' \,|\, M \,|\, E \,|\, P$$

$$\overset{\tau_p[7]}{\rightsquigarrow} F \,|\, (a, b, p).R' \,|\, \overline{p}.P' \,|\, (c, p, \overline{b}, \overline{n}).E' \,|\, M \,|\, E \,|\, P \equiv F \,|\, M | E \,|\, P \,|\, R \,|\, E \,|\, P$$

Note that this segment of the pathway deactivates pE into E and P.

A high level view of the behaviour of the ERK system $F \,|\, M \,|\, E \,|\, P \,|\, R | pE$ is represented abstractly by the cascade and regulation sequences:

$$F \,|\, M | E \,|\, P \,|\, R \,|\, E \,|\, P \underset{\rightarrow\, cas\, \rightarrow}{\overset{\leftarrow\, reg\, \leftarrow}{\rightleftarrows}} F \,|\, M \,|\, E \,|\, P \,|\, R | pE \underset{\leftarrow\, reg\, \leftarrow}{\overset{\rightarrow\, cas\, \rightarrow}{\rightleftarrows}} F \,|\, M | pE \,|\, R | pE$$

The cascade produces pE which can signal the nucleus and the regulation sequence consumes pE in order to stop R regulating F.

M, E and R exhibit the following patterns of behaviour (putting aside the undoing of immediately reversible reactions on a and b, and $\overline{n}$) which we write with controller actions: M : $a.p.\underline{a}.\overline{c}.\underline{p}.\underline{\overline{c}}$; E : $c.p.\underline{c}.\overline{b}.\underline{p}.\underline{\overline{b}}$ and R : $a.b.\underline{a}.p.\underline{b}.\underline{p}$. We note the common pattern (modulo action names) and also behaviours that break causal dependencies: for example $\underline{a}$ happens before $\underline{p}$ in M although a causes p.

Finally, we define controller terms for the proteins M, E and R that will ensure that reactions follow the order of the cascade and regulation sequences.

$$C_M \overset{\mathrm{df}}{=} a.C'_M \qquad C'_M \overset{\mathrm{df}}{=} \underline{a}.C_M + p.\underline{a}.\overline{c}.\underline{p}.\underline{\overline{c}}.C_M$$

$$C_E \overset{\mathrm{df}}{=} c.p.\underline{c}.C'_E \qquad C'_E \overset{\mathrm{df}}{=} \overline{n}.N + \overline{b}.C''_E \qquad C''_E \overset{\mathrm{df}}{=} \underline{\overline{b}}.C'_E + \underline{p}.\underline{\overline{b}}.C_E$$

$$C_R \overset{\mathrm{df}}{=} a.C'_R \qquad C'_R \overset{\mathrm{df}}{=} \underline{a}.C_R + b.C''_R \qquad C''_R \overset{\mathrm{df}}{=} \underline{b}.C'_R + \underline{a}.p.\underline{b}.\underline{p}.C_R$$

Claim. $(F \,|\, M\langle C_M\rangle \,|\, E\langle C_E\rangle \,|\, P \,|\, R\langle C_R\rangle \,|\, pE\langle C'_E\rangle) \setminus \{a, b, c, p\}$ exhibits precisely the cascade and regulation reactions of $(F \,|\, M \,|\, E \,|\, P \,|\, R | pE) \setminus \{a, b, c, p\}$.

4 Conclusion

We have presented a reversible process calculus with a new execution control operator and illustrated its usefulness and expressiveness with several examples, including long-running transactions with simple compensations and the ERK signalling pathway. The new operator allows us to model a variety of modes of reverse computation, ranging from strict backtracking to reversing which respects causal ordering of events, and even reversing which violates causal ordering. This last form of reversing has not been studied before and in our view it deserves further investigation. The execution control operator can also be used to encode irreversible actions, it can act as the restriction operator of CCS in contexts involving communicating processes, and it allows us to construct terms that reach a state after a reversal that cannot be reached by computing forwards only.

Acknowledgements. We are grateful to the Reversible Computation 2012 referees and participants for their helpful comments and suggestions. The second author acknowledges partial support by EPSRC grant EP/G039550/1 and the Japan Society for the Promotion of Science (JSPS) grants S-09053 and FU-019.

References

1. Berry, G., Boudol, G.: The chemical abstract machine. Theoretical Computer Science 96(1), 217–248 (1992)
2. Blinov, M.L., Yang, J., Faeder, J.R., Hlavacek, W.S.: Graph Theory for Rule-Based Modeling of Biochemical Networks. In: Priami, C., Ingólfsdóttir, A., Mishra, B., Riis Nielson, H. (eds.) Transactions on Computational Systems Biology VII. LNCS (LNBI), vol. 4230, pp. 89–106. Springer, Heidelberg (2006)
3. Calder, M., Gilmore, S., Hillston, J.: Modelling the Influence of RKIP on the ERK Signalling Pathway Using the Stochastic Process Algebra PEPA. In: Priami, C., Ingólfsdóttir, A., Mishra, B., Riis Nielson, H. (eds.) Transactions on Computational Systems Biology VII. LNCS (LNBI), vol. 4230, pp. 1–23. Springer, Heidelberg (2006)
4. Cardelli, L., Laneve, C.: Reversible structures. In: 9th International Conference on Computational Methods in Systems Biology, pp. 131–140. ACM (2011)
5. Cho, K.-H., Shin, S.-Y., Kim, H.-W., Wolkenhauer, O., McFerran, B., Kolch, W.: Mathematical Modeling of the Influence of RKIP on the ERK Signaling Pathway. In: Priami, C. (ed.) CMSB 2003. LNCS, vol. 2602, pp. 127–141. Springer, Heidelberg (2003)
6. Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J.: Rule-Based Modelling of Cellular Signalling. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 17–41. Springer, Heidelberg (2007)
7. Danos, V., Krivine, J.: Reversible Communicating Systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)
8. 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)

9. Danos, V., Krivine, J.: Formal molecular biology done in CCS-R. In: Proceedings of the 1st Workshop on Concurrent Models in Molecular Biology BioConcur 2003. ENTCS, vol. 180, pp. 31–49 (2007)
10. 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)
11. 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)
12. 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)
13. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
14. Mousavi, M., Phillips, I.C.C., Reniers, M.A., Ulidowski, I.: Semantics and expressiveness of Ordered SOS. Information and Computation 207(2), 85–119 (2009)
15. Phillips, I.C.C., Ulidowski, I.: Reversing Algebraic Process Calculi. In: Aceto, L., Ingólfsdóttir, A. (eds.) FOSSACS 2006. LNCS, vol. 3921, pp. 246–260. Springer, Heidelberg (2006)
16. Phillips, I.C.C., Ulidowski, I.: Reversing algebraic process calculi. Journal of Logic and Algebraic Programming 73, 70–96 (2007)
17. Phillips, I.C.C., Ulidowski, I.: A logic with reverse modalities for history-preserving bisimulations. In: Proceedings 18th International Workshop on Expressiveness in Concurrency. EPTCS, vol. 64, pp. 104–118 (2011)
18. Phillips, I.C.C., Ulidowski, I.: A hierarchy of reverse bisimulations on stable configuration structures. Mathematical Structures in Computer Science 22, 333–372 (2012)
19. Ulidowski, I., Phillips, I.C.C.: Ordered SOS rules and process languages for branching and eager bisimulations. Information and Computation 178(1), 180–213 (2002)
20. Vera, J., Rath, O., Balsa-Canto, E., Banga, J.R., Kolch, W., Wolkenhauer, O.: Investigating dynamics of inhibitory and feedback loops in ERK signalling using power-law models. Molecular BioSystems 6, 2174–2191 (2010)

Controlled Reversibility and Compensations*

Ivan Lanese[1], Claudio Antares Mezzina[2], and Jean-Bernard Stefani[3]

[1] Focus Team, University of Bologna/INRIA, Italy
lanese@cs.unibo.it
[2] SOA Unit, FBK, Trento, Italy
mezzina@fbk.eu
[3] INRIA Grenoble-Rhône-Alpes, France
jean-bernard.stefani@inria.fr

Abstract. In this paper we report the main ideas of an ongoing thread of research that aims at exploiting reversibility mechanisms to define programming abstractions for dependable distributed systems. In particular, we discuss the issues posed by concurrency in the definition of controlled forms of reversibility. We also discuss the need of introducing compensations to deal with irreversible actions and to avoid to repeat past errors.

1 Motivation

In this paper we report the main ideas of an ongoing thread of research that aims at exploiting reversibility mechanisms to define programming abstractions for dependable distributed systems. Many such abstractions have been proposed in the literature, concerning for instance exception handling, checkpointing, transactions and the like [6,11,12], and made available to programmers as language primitives, libraries or middleware functions. However these different proposals lack formal foundations and do not generally compose well. This raises the question of whether some unifying framework can be found to shed light on the relations among these apparently unrelated mechanisms. Clearly, a number of these mechanisms are based on some form of *undo*, allowing to annul the effect of actions that lead to an error. We thus ask the following question:

If we were able to undo every action in a distributed program execution, would we be able to understand and integrate those different mechanisms?

We started our endeavor in the framework of concurrency theory and in particular using process calculi, developing small reversible languages and trying to understand their properties and their expressive power.

Paper Outline. Section 2 recalls the main features of reversibility in a concurrency setting. Section 3 discusses and compares different mechanisms for controlling reversibility. Section 4 outlines some ideas for combining reversibility and compensations. Section 5 concludes the paper.

* This work has been partially supported by the French National Research Agency (ANR), project REVER n. ANR 11 INSE 007.

2 Reversibility in a Concurrency Setting

The problem of understanding reversibility in a process calculus scenario had already been considered in the seminal paper [8], with motivations coming from computational biology, where systems are naturally reversible. In [8] a reversible variant of CCS has been proposed. A main achievement of that paper is the definition of *causal consistency*, a formal criterion for reversibility in concurrent systems. In a concurrency setting (and even more in a distributed one) there may be no clear understanding of which was the last action performed by the system, or which was the previous state. With causally consistent reversibility one moves back by undoing any action that could have been the last one, i.e. any action on which no other action depends. Essentially, actions are undone in reverse order with respect to forward execution, up to possible swaps in the order of execution of concurrent actions.

To better understand this crucial concept, consider the example in Figure 1(a). From state M there are two possible paths leading to state N, one executing first a and then b (on the left), and the other executing first b and then a (on the right). If the two actions a and b are concurrent, possibly executed by physically remote components, it may be difficult to distinguish the two computations. Thus one should be able to reverse any of the two executions by reversing the other, i.e. if the forward computation proceeds by executing first a and then b (double-pointed arrow in Figure 1(b)), not only undoing first b and then a, but also undoing first a and then b (wave arrow in Figure 1(b)) is a valid reverse computation.

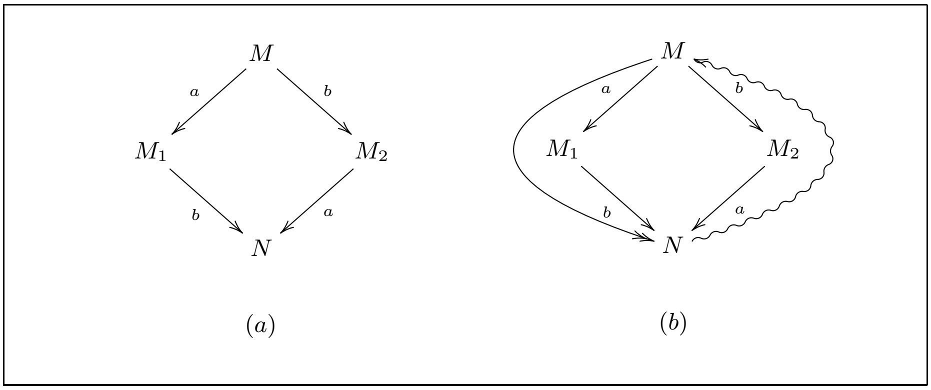


Fig. 1. Example of Causally Consistent Executions

To allow causally consistent reversibility one has to add history information to the different threads in a computation. Choosing threads as the granularity at which to store this information is suitable for causal consistency since actions inside the same thread are causally dependent, while actions in different threads are mostly concurrent. Subsequent works have shown that such a framework can be applied to different process calculi, in particular to a family of CCS-like calculi [19], to the (higher-order) π-calculus [16], and to a subset of Oz [17].

3 Controlling Reversibility

In the works discussed above, reversibility is essentially *non-deterministic*, in the sense that, when both forward and backward steps are possible, there is no way of deciding whether to go forward or to go backward. This means that those works specify *how* the system can reverse a forward computation and what kind of information it should exploit, but they give no hint about *when* forward execution should be preferred over backward one and vice-versa.

In the fault-tolerance setting, there is a general answer to this question: to use reversibility for error recovery, the system should normally execute forward, and backward execution should be exploited only when needed to recover from errors. Such a general guideline, however, can be implemented using different strategies. We describe below three main strategies, specifying scenarios where they can be applied. This allows us to structure the design space of controlled reversibility, and to categorize the approaches in the literature accordingly.

Internal Control: Specific commands inside processes specify whether the process itself should go forward or backward. A possibility along this line has been explored in [9], where *irreversible* actions, i.e. actions that once performed cannot be undone, have been integrated in reversible CCS and shown able to implement a simple form of transactions. In [9], however, it is not clear how to relate irreversible actions and error recovery. To make this relation more apparent, we proposed a dual approach [15] where an explicit rollback primitive is used to trigger backward execution. The idea is that when an error is spotted, the rollback primitive can be used to go back to a consistent state. To specify how far back to go the rollback primitive takes as parameter a label referring to a past action, and it undoes all (and only) the actions causally dependent on it, that is all the actions generated because of it. This choice is coherent with, and indeed forced by, causally consistent reversibility. In fact, in a concurrent scenario a specification such as "go back n steps" (typical of sequential reversible debuggers such as in [3]) is not meaningful, since there is no clear understanding of which the last n actions have been. Irreversible actions and explicit rollback are dual, one specifying when it is *forbidden* to go back, and the other one specifying when it is *required* to go back. Their combination is not trivial, however, since one has to decide what to do in case of conflicts, e.g. if a rollback requires to go back past an irreversible action. We will outline a possible solution to this problem when discussing compensations.

External Control: This approach follows the separation of concerns principle: a process is potentially able to go both backward and forward, while another process is in charge of controlling it by deciding when it has to go backward and when it has to go forward. Such an approach is suitable, e.g., for hierarchic component-based systems, where the father component may decide when and how to rollback its child, and the children notify the father in case of errors. Such a hierarchical structure for failure handling is typically the one advocated for Erlang systems [1]. External control naturally emerges in a reversible debugger: the user, through the debugger interface, decides whether the program under debugging should execute forward or should get back to a previous computation. Following the causally consistent approach, when going backward the user should specify which past action to undo, and the system should be in charge of finding its dependencies and undoing their execution. This is in contrast, e.g., to [7] where the user has to decide which actions to undo and in which order. External control has been applied also to biological reversible systems in [20]. There a reversible CCS process P is controlled by a controller process C, which is again a CCS process. The controller C always computes forward, and it constrains the possible actions of P, thus decreasing the non-determinism due to reversibility and to concurrency. This allows, together with a generalized form of prefix, to model different forms of reversibility, including reversibility that is not (always) causally consistent.

Semantic Control: In this approach the semantics of the language is extended with guidelines on whether to go forward or to go backward. Consider the following scenario: a reversible program is used to perform a state-space exploration looking for some solution of a given problem. In this case reversibility is needed to backtrack in case a branch with no solution has been taken. One can imagine to add to the history information about whether and how many times a particular path has been taken and favor paths (and directions of execution) leading to less explored areas. For instance one can label each action with the number of times it has been tried, and choose among the enabled actions (both forward and backward) one which has minimal value. It is clear that in such a way a finite state space is completely explored, allowing to find a solution if at least one exists. Another approach has recently been explored in [2], where computing steps are taken subject to some probability, and the rate of forward and backward computing steps are derived from a set of formal energy parameters. The contribution of the paper is to show that there exists a lower bound on energy costs to guarantee that a process commits a forward computation in finite average time.

4 Reversibility and Compensations

Using some forms of internal or external control, reversibility may lead to divergence. In particular, the process itself is not aware of the fact that a specific computation has already been executed, has failed, and has been rollbacked. Thus the same computation could be performed again and again, possibly forever. To avoid such a problem we put forward a solution based on *compensations.* Compensations have been proposed as a main building block for long running transactions, first in the area of database theory [13] and then in service-oriented computing [5,4,18,14]. A long running transaction is a computation that either succeeds, or, in case of failure, it is compensated. A compensation is an ad hoc piece of code which is in charge of leading the system back to a consistent state, possibly different from the ones the execution went through. Compensations seem antithetic to reversibility, since their aim is exactly to deal with situations where rollback is not possible or not desired. However, the two concepts can be fruitfully combined. Consider any form of controlled reversibility, e.g., one based on internal control. Assume that for some of the statements of the program a compensation is defined. One can consider for instance a statement of the form A `comp` B. The idea is that during forward execution A `comp` B behaves as A. However, if its execution has to be rollbacked (possibly as part of a larger rollback), the effect of A is annulled (that is A is actually rollbacked) and then the restored A is replaced by B. Both the steps are important: rollback is needed to undo some nasty effects of A on the state, while replacing A with B is needed to avoid re-doing a try that already failed, and would probably fail again. Note that the first aspect is completely missing from the compensation approaches in the literature [5,4,18,14].

Consider a typical web service scenario. A is an invocation of a flight reservation service of some airline. Possibilities for B are for instance to execute the same booking using another airline, or to update the database of preferred airlines by adding the information that the invocation of A has failed. These two possibilities are representatives of two classes of compensations with different features. We call compensations in the first class *replacing compensations*, since they aim at doing what action A was supposed to do, but in a different way. We call compensations in the second class *tracing compensations*, since they give up on what action A was trying to do, but they just aim at keeping trace of the failure. This information will be used later on by the application. In the example one may imagine that the application uses trust as a criterion to choose the airline to be invoked, and the tracing compensation decreases the trust value of the airline whose invocation failed. In the long running transactions field instead, the main aim of compensations is to remedy the nasty effects of A, e.g. annulling the previous booking to avoid to pay for it. In our case this is done automatically by the reversibility mechanism. We call this last form of compensation *repairing compensation.*

The replacing compensation example, trying again the booking with a different airline, makes it intuitive that compensations may have their own compensations, recursively. A less evident issue is the following: as we have seen, the reversibility machinery tracks causal dependencies, thus one has to specify how the compensation is inserted in the causality relation. There are two main possibilities, one suitable for replacing compensations, and the other for tracing compensations. For replacing compensations, compensation B should take the place of A in the causality relation, since it is an alternative to it. On the other hand, a tracing compensation B is just used to update the state with information obtained by the failed attempt, thus it is not causally related to A causes, but it is independent. To better understand the difference let us analyze what happens if a larger part of the execution has to be annulled, e.g. since, in the airline booking scenario above, the user changed its mind and decided not to travel any more. The replacing compensation "book with another airline" has to be reverted, since it is no more meaningful. This is exactly what happens since the compensation is now causally dependent on A causes. Instead, the tracing compensation "remember that the chosen airline is not good" should be applied anyway, since it can be used for further bookings later on, and thus should not depend on A causes.

So far we considered the use of compensations to avoid repeating past errors. However compensations can also be used in a way closer to their original purpose, i.e. to deal with irreversible actions. Keep in mind that whatever the support for reversibility is, there will always be actions which cannot be undone. This is mainly related to two situations: the action may be inherently irreversible, e.g. a side effect on the real world such as printing a document, or the action is in principle reversible but it is out of the control of the considered system. An example of this last possibility is a distributed application where some of the components provide support for reversibility, while others do not. In both the cases one may attach compensations to irreversible actions: reversible actions are reverted, and irreversible actions are compensated. In this last case repairing compensations are normally needed.

We can now clarify the issue of the combination of irreversible actions and rollback: in case a rollback request is issued, and it requires to undo an irreversible action, actions are undone till the irreversible action is found, and its (repairing) compensation is executed, instead of reversing it and all the actions it depends on. Note that in this setting some actions are equipped with compensations, while others are reversed, while in the previous setting each action was both reversed and compensated.

5 Conclusion

In this paper we discussed three main issues related to defining and exploiting reversibility in a concurrent scenario. The first issue was how to define mechanisms allowing to go backward and forward in a concurrent execution. The second issue was how to control reversibility, i.e. how to specify whether to go forward or backward, and up to where. The third issue was how to avoid repeating the same errors, and how to deal with irreversible actions. The first such issue has been discussed in the literature to some extent, relying on the main concept of causally consistent reversibility. Related to the second issue, this is the first time, as far as we know, that a taxonomy of the possible approaches has been defined. Concerning the third issue, the only related work in the literature is [10], which proposes a transactional mechanism with some reversibility features. This work however comes from the opposite direction, since it starts from the problem of how to define interacting transactions.

Many other issues remain to answer our original question. In particular, the mechanisms we sketched above should be fully specified, and their expressive power has to be assessed against proposals in the literature. Our endeavor would be fully successful only if we show that using a reversible framework allows to recover, improve and combine existing techniques for dependable concurrent systems, and possibly to define new ones. Also, practical issues must be solved to make the approach usable in a real programming language. For instance, the interplay between reversibility and language features such as the type system or modules should be considered. Also, the space and time overhead due to reversibility have to be measured and minimized. See [17] for a preliminary analysis of this issue.

References

1. Armstrong, J.: Making Reliable Distributed Systems in the Presence of Software Errors. PhD thesis, KTH, Stockholm, Sweden (2003)
2. 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)
3. Boothe, B.: Efficient Algorithms for Bidirectional Debugging. In: Proc. of PLDI 2000, pp. 299–310. ACM Press (2000)
4. Bruni, R., Melgratti, H., Montanari, U.: Theoretical Foundations for Compensations in Flow Composition Languages. In: Proc. of POPL 2005, pp. 209–220. ACM Press (2005)
5. Butler, M., Hoare, S.T., Ferreira, C.: A Trace Semantics for Long-Running Transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525, pp. 133–150. Springer, Heidelberg (2005)
6. Collet, R., Van Roy, P.: Failure Handling in a Network-Transparent Distributed Programming Language. In: Dony, C., Lindskov Knudsen, J., Romanovsky, A., Tripathi, A. (eds.) Exception Handling. LNCS, vol. 4119, pp. 121–140. Springer, Heidelberg (2006)
7. Cook, J.J.: Reverse Execution of Java Bytecode. Comput. J. 45(6), 608–619 (2002)
8. Danos, V., Krivine, J.: Reversible Communicating Systems. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 292–307. Springer, Heidelberg (2004)
9. 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)
10. de Vries, E., Koutavas, V., Hennessy, M.: Communicating Transactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 569–583. Springer, Heidelberg (2010)
11. Elnozahy, E.N., Alvisi, L., Wang, Y.M., Johnson, D.B.: A Survey of Rollback-Recovery Protocols in Message-Passing Systems. ACM Comput. Surv. 34(3) (2002)
12. Eppinger, J.L., Mummert, L.B., Spector, A.Z.: Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann (1991)
13. Garcia-Molina, H., Gawlick, D., Klein, J., Kleissner, K., Salem, K.: Coordinating Multi-Transaction Activities. Technical Report CS-TR-2412, University of Maryland, Dept. of Computer Science (1990)
14. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: On the Interplay Between Fault Handling and Request-Response Service Invocations. In: Proc. of ACSD 2008, pp. 190–199. IEEE Computer Society Press (2008)

15. 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)
16. 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)
17. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.-B.: A Reversible Abstract Machine and Its Space Overhead. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 1–17. Springer, Heidelberg (2012)
18. Oasis. Web Services Business Process Execution Language Version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
19. Phillips, I., Ulidowski, I.: Reversing Algebraic Process Calculi. J. Log. Algebr. Program. 73(1-2) (2007)
20. 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, pp. 218–232. Springer, Heidelberg (2012)

Author Index

Axelsen, Holger Bock 171

Burignat, Stéphane 160

Coecke, Bob 1

De Vos, Alexis 160
Drechsler, Rolf 183, 197
Dueck, Gerhard W. 125
Duncan, Ross 1

Early, Diarmuid 85

Fujimoto, Richard 72

Gajardo, Anahí 44
Gao, Ang 85

Hou, Cong 72

James, Roshan P. 57
Jefferson, David 72

Kerntopf, Paweł 138
Krivine, Jean 204
Kutrib, Martin 14

Lanese, Ivan 233

Malcher, Andreas 14
Mezzina, Claudio Antares 233
Miller, D. Michael 112
Minato, Shin-ichi 197
Morita, Kenichi 29

Nayeem, Noor M. 152

Ollinger, Nicolas 44

Phillips, Iain 218

Quinlan, Daniel 72

Rahman, Md. Mazder 125
Rice, Jacqueline E. 152

Sabry, Amr 57
Sasanian, Zahra 112
Schellekens, Michel 85
Schultz, Ulrik Pagh 97
Seiter, Julia 183
Soeken, Mathias 183, 197
Stefani, Jean-Bernard 233
Szyprowski, Marek 138

Thomsen, Michael Kirkedal 160, 171
Torres, Rodrigo 44

Ulidowski, Irek 218

Vermeirsch, Kenneth 160
Vuduc, Richard 72

Wille, Robert 183, 197

Yuen, Shoji 218