Jérôme Durand-Lose
Sergey Verlan (Eds.)

LNCS 10881

Machines, Computations, and Universality

8th International Conference, MCU 2018
Fontainebleau, France, June 28–30, 2018
Proceedings

M
2018
U C

Springer

Jérôme Durand-Lose · Sergey Verlan (Eds.)

Machines, Computations, and Universality

8th International Conference, MCU 2018
Fontainebleau, France, June 28–30, 2018
Proceedings

Springer

Editors
Jérôme Durand-Lose
Université d'Orléans
Orleans
France

Sergey Verlan
Université Paris Est
Creteil
France

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-92401-4 ISBN 978-3-319-92402-1 (eBook)
https://doi.org/10.1007/978-3-319-92402-1

Library of Congress Control Number: 2018944397

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

Printed on acid-free paper

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

Preface

This volume contains the papers presented at MCU 2018: the 8th Conference on Machines, Computations and Universality, held during June 28–30, 2018 on the campus of the IUT de Fontainebleau at the University of Paris-Est Créteil, Fontainebleau, France. This edition was co-located with the 17th International Conference on Unconventional Computation and Natural Computation (UCNC 2018).

The MCU series of international conferences traces its roots back to the mid-1990s, and has since been concerned with gaining a deeper understanding of computation through the study of models of general purpose computation. MCU explores computation in the setting of various discrete models (Turing machines, register machines, cellular automata, tile assembly systems, rewriting systems, molecular computing models, neural models, concurrent systems, etc.) and analog and hybrid models (BSS machines, infinite time cellular automata, real machines, quantum computing, etc.). There is a particular (but not exclusive) emphasis given on the following:

– The search for frontiers between decidability and undecidability in the various models. (For example, what is the smallest number of pairs of words for which the Post correspondence problem is undecidable, or what is the largest state-symbol product for which the halting problem is decidable for Turing machines?)
– The search for the simplest universal models (such as small universal Turing machines, universal rewriting systems with few rules, universal cellular automata with small neighborhoods and a small number of states, etc.)
– The computational complexity of predicting the evolution of computations in the various models. (For example, is it possible to predict an arbitrary number of time steps for a model more efficiently than explicit step by step simulation of the model?)
– How parallelism can be connected to decidability, complexity and universality.
– Universality and undecidability in continuous models of computation.

Previous MCU conferences took place in Famagusta, North Cyprus (2015), Zürich, Switzerland (2013), Orléans, France (2007), Saint Petersburg, Russia (2004), Chişinău, Moldova (2001), Metz, France (1998), and Paris, France (1995). More information on MCU 2018 can be found at https://mcu2018.lacl.fr/.

This year, the number of submission was very low (10) but all in the scope of the conference. Each submission was reviewed by at least three (and on average 3.3) Program Committee members. The committee decided to accept seven papers for presentation and publication in these proceedings. The submission process was handled through EasyChair and went smoothly as usual.

The program also includes five invited talks.

– Erzsébet Csuhaj-Varjú provided in “Watson-Crick Complementarity, L Systems, Computation” a survey on properties of such systems that apply derivation on a

string or its complement with special emphasis on the computational power of the different variants.
- Rudolf Freund presented in "Control Mechanisms for Array Grammars on Cayley Grids" a huge survey on all the results on systems in this area, including variants and computing power.
- Natasha Jonoska presented in "The Shape of Computation" models and a topology for computation that build objects from nano to macro scales and in particular DNA-based ones.
- Michel Raynal introduced in "A Pleasant Stroll Through the Land of Distributed Machines, Computation, and Universality" non-compubatility issues in distributed computing which differ from the classic context: The abstract state/dynamics of the system is concerned, not the output.
- Damien Woods presented in "Molecular Computation with DNA Self-Assembly" recent work with both theoretical and experimental results in the field.

We are thankful to EasyChair for the easy management of the submissions as well as preparing the proceedings.

We want to thank everybody in the Organizing Committee who worked hard to make this edition successful.

Partial financial support for the conference was provided by IUT de Sénart-Fontainebleau, the University of Paris-Est, Laboratoire d'Algorithmique Complexité et Logique, the University of Paris-Est Créteil (UPEC), and Faculté des Sciences et Technologies of the University of Paris-Est Créteil. We also thank the administration of IUT de Fontainebleau for the perfect infrastructure made available to MCU 2018. Finally, we would like to thank our secretaries Nathalie Gillet and Flore Tsila for extensive assistance in organizing the event and for smoothly running the conference.

The editors warmly thank the Program Committee, the organizers, the invited speakers, the authors of the papers, the external reviewers, the speakers of informal presentations, and all the participants for their contribution to the success of the conference.

June 2018

Jérôme Durand-Lose
Sergey Verlan

Organization

Program Committee

Artiom Alhazov	Academy of Sciences of Moldova, Moldova
Pablo Arrighi	University of Marseille, France
Nathalie Aubrun	CNRS, ENS Lyon, France
Laurent Bienvenu	CNRS, Université de Montpellier, France
Olivier Bournez	Ecole Polytechnique, France
Jérôme Durand-Lose (Co-chair)	Université d'Orléans, France
François Fages	Inria, Saclay, France
Henning Fernau	University of Trier, Germany
Rudolf Freund	Technical University of Vienna, Austria
Christine Gaßner	University Greifswald, Germany
Frédéric Gruau	Université Paris Sud, France
Hendrik Jan Hoogeboom	Universiteit Leiden, The Netherlands
Gabriel Istrate	West University of Timisoara, Romania
Emmanuel Jeandel	Université de Lorraine, France
Seth Lloyd	Massachusetts Institute of Technology, USA
Benedek Nagy	University of Debrecen, Hungary and Eastern Mediterranean University, North Cyprus
Turlough Neary	University of Zürich and ETH Zürich, Switzerland
Matthew Patitz	University of Arkansas, USA
Simon Perdrix	CNRS, LORIA, France
Nicolas Schabanel	CNRS, Université Paris Diderot (Paris 7), France
Klaus Sutner	Carnegie Mellon University, USA
Sergey Verlan (Co-chair)	Université Paris-Est Créteil, France

MCU Steering Committee

Jérôme Durand-Lose (Chair)	University of Orleans, France
Matthew Cook	University of Zürich and ETH Zürich, Switzerland
Erzsébet Csuhaj-Varjú	Eötvös Loránd University, Budapest, Hungary
Natasha Jonoska	University of South Florida, USA
Maurice Margenstern	University of Metz, France
Kenichi Morita	Hiroshima University, Japan
Benedek Nagy	University of Debrecen, Hungary and Eastern Mediterranean University, North Cyprus

Arto Salomaa	University of Turku, Finland
Kumbakonam Govindarajan Subramanian	University of Science, Malaysia

Organizing Committee

Patrick Cegielski (Co-chair)	Université Paris-Est Créteil, France
Julien Cervelle	Université Paris-Est Créteil, France
Luidnel Maignan	Université Paris-Est Créteil, France
Antoine Spicher	Université Paris-Est Créteil, France
Pierre Valarcher	Université Paris-Est Créteil, France
Pascal Vanier	Université Paris-Est Créteil, France
Sergey Verlan (Co-chair)	Université Paris-Est Créteil, France

Additional Reviewers

Martin Delacourt
Emmanuel Hainry
Sergiu Ivanov
François Laroussinie
Antoine Spicher
Jan van Leeuwen
Zizhu Wang

Sponsoring Institutions

IUT de Sénart-Fontainebleau
University of Paris-Est
Laboratoire d'Algorithmique Complexité et Logique
University of Paris-Est Créteil (UPEC)
Faculté des Sciences et Technologies of the University of Paris-Est Créteil

Invited Talks

Watson-Crick Complementarity, L Systems, Computation

Erzsébet Csuhaj-Varjú

Faculty of Informatics, ELTE Eötvös Loránd University, Pázmány Péter sétány 1/c, Budapest 1117, Hungary
csuhaj@inf.elte.hu

Watson-Crick Lindenmayer systems (WL systems) are variants of L systems, inspired by the well-known phenomenon of the double helix of DNA. These systems are defined over a DNA-like alphabet (each letter has a complementary letter and this relation is symmetric). Depending on whether or not a special condition holds, the derivation step is applied either to the string or to its complementary string. The trigger for turning to the complementary string is given by a language over the DNA-like alphabet of the system: if the generated string is an element of the trigger, then the derivation continues with its complementary string, otherwise the obtained string is considered. For a DNA alphabet $V = \{a_1, \ldots, a_n, \bar{a}_1, \ldots, \bar{a}_n\}$, $n \geq 1$, a trigger $TR \subseteq V^*$ is called standard if it consists of all words of V^* which have more occurrences of barred letters than non-barred ones. The first model, the Watson-Crick D0L system, with standard trigger, was introduced in [9, 10].

A network of Watson-Crick L systems (an NWL system) is a finite set of WL systems over a common DNA-like alphabet and with the same trigger. The WL systems act on their own strings in a synchronized manner and after each derivation step communicate some of the obtained words to the other WL systems of the network. The condition for communication (the communication protocol) is based on the trigger for turning to the complementary string. The first variant of these models, the network of Watson-Crick D0L systems, was defined in [5].

Watson-Crick L systems and their networks have been examined in detail during the years (see, for example [1–3, 6, 8, 11–14]).

In this talk, we provide a survey on properties of WL systems and NWL systems, with special emphasis on their computational power, in particular on the computational completeness of the different variants. We discuss decidability problems of WL systems with standard and with regular triggers and with the customary and with certain relaxed forms of derivation (for example, problems of stability, ultimate stability, sequence and language equivalence of WD0L systems). We report how WL systems as computing devices with infinite runs may "go beyond Turing" [4]. In case of networks of WL systems, we focus on their properties concerning communication: for example, string population growth at the components under functioning, black and white wholes, communication patterns.

Our overview uses new approaches, motivated by a recent generalization of WD0L systems to discrete Watson-Crick dynamical systems [7]. Some open problems, research directions for future study are also proposed.

Acknowledgement. Supported by grant No. 120558, NKFIH–National Research, Development and Innovation Office, Hungary.

References

1. Csima, J., Csuhaj-Varjú, E., Salomaa, A.: Power and size of extended Watson-Crick L systems. Theoret. Comput. Sci. **290,** 1665–1678 (2003)
2. Csuhaj-Varjú, E.: Computing by networks of Watson Crick D0L systems. In: Ito, M. (ed.) In: Proceedings Algebraic Systems, Formal Languages and Computation. RIMS Kokyuroku, vol. 1166, pp. 42–51, August 2000. Research Institute for Mathematical Sciences. Kyoto University, Kyoto
3. Csuhaj-Varjú, E.: Networks of standard watson-crick D0L systems with incomplete information communication. In: Karhumäki, J., Maurer, H., Rozenberg, G. et al. (eds.) Theory is Forever. LNCS, vol. 3113, pp. 35–48. Springer, Heidelberg (2004)
4. Csuhaj-Varjú, E., Freund, R., Vaszil, Gy.: Watson-crick T0L systems and red-green register machines. Fundamenta Informaticae **155** (1–2), 111–129 (2017)
5. Csuhaj-Varjú, E., Salomaa, A.: Networks of Watson-Crick D0L systems. In: Ito, M., Imaoka, T. (eds.) In: Proceedings of the International Colloquium of Words, Languages, & Combinatorics III, pp. 14–21. Kyoto, Japan, March, 2000. World Scientific Publishing Co., Singapore, pp. 134–149 (2003)
6. Csuhaj-Varjú, E., Salomaa, A.: The power of networks of watson-crick D0L systems. In: Jonoska, N., Păun, G., Rozenberg, G. et al. (eds.) Aspects of Molecular Computing. LNCS, vol. 2950, pp. 106–118. Springer, Heidelberg (2004)
7. Honkala, J.: Discrete watson-crick dynamical systems. Theoret. Comput. Sci. **701,** 125–131 (2017)
8. Honkala, J., Salomaa, A.: Watson-crick D0L systems with regular triggers. Theoret. Comput. Sci. **259**, 689–698 (2001)
9. Mihalache, V. Salomaa, A.: Lindenmayer and DNA: watson-crick D0L systems. EATCS Bull. **62**, 160–175 (1997)
10. Mihalache, V., Salomaa, A.: Language-theoretic aspects of DNA complementarity. Theoret. Comput. Sci. **250**, 163–178 (2001)
11. Salomaa, V.: Uni-transitional watson-crick D0L systems. Theoret. Comput. Sci. **281**, 537–553 (2002)
12. Salomaa, A. Sosík, P.: Watson-crick D0L systems: the power of one transition. Theoret. Comput. Sci. **301**, 187–200 (2003)
13. Sears, D., Salomaa, K.: Extended watson-crick L systems with regular trigger languages and restricted derivation modes. Nat. Comput. **11**(4), 653–664 (2012)
14. Sosík, P.: Watson-crick D0L systems: generative power and undecidable problems. Theoret. Comput. Sci. **306** (1–2), 101–112 (2003)

Control Mechanisms for Array Grammars on Cayley Grids (Abstract)

Rudolf Freund

Faculty of Informatics, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
rudi@emcc.at

In this paper, the computational power of several control mechanisms for specific variants of (sequential, isometric) array grammars generating arrays on Cayley grids of finitely presented groups is investigated. Using #-context-free array productions together with control mechanisms as control graphs, matrices, permitting and forbidden rules, partial order on rules or activation and blocking of rules the same computational power is obtained as when using arbitrary array productions.

The Shape of Computation

Nataša Jonoska

Department of Mathematics and Statistics, University of South Florida, Tampa, Florida, USA

We are used to think about computation as processes performed by our various electronic devices. Theoretically, we often characterize computation through models describing, or simulating, symbol manipulation and/or compositions of logic gates operating on binary alphabet. On the other side, one can think of the processes that occur in our 3D world (from nano to macro level) as results of "shape processing" operations. The "shape" can result of a computation, or describe the computational process itself.

Self-organization of 3D objects can be seen as computational result. A complex structure, sometimes even without it achieving minimal energy conformation, can result from local information processing between modular components, and the final shape can actually represent the answer to a computational question. An example of such computational shape formation is the molecular self-assembly, in particular DNA self-assembly. We present four models that describe computations by construction of complex spatial formations and show an experimental proof-of-principle for one such model. Further, we associate an algebra to DNA origami whose elements and their product correspond to different assemblies.

One can also consider the "shape" of the assembly process, that is, the computational process itself, and describe it by directed spatial graphs. In this way computations can have different graph topologies. How do these topologies distinguish computations? We show a way to associate topology, and a topological measure for assembly processes.

A Pleasant Stroll Through the Land of Distributed Machines, Computation, and Universality

Michel Raynal[1, 2] and Jiannong Cao[2]

[1] Institut Universitaire de France and Univ Rennes, IRISA CNRS Inria, France
[2] Department of Computing, Polytechnic University, Hong Kong
raynal@irisa.fr

Not only the world is distributed, but more and more applications are distributed. Hence, a fundamental question is the following one: What can be computed in a distributed system? The answer to this question depends on the environment in which evolves the considered distributed system, i.e., on the assumptions the system relies on. This environment is very often left implicit and nearly always not formulated in terms of precise underlying requirements. In the extreme case where the environment is such that there is no synchrony assumption and the computing entities may commit failures, some problems become impossible to solve. Given a distributed computing problem, it is consequently important to know the weakest assumptions (lower bounds) that give the limits beyond which the considered distributed problem cannot be solved. This paper is a short introduction to this kind of issues. It is made up of short sections, each addressing an important point of the theory of distributed computing. Its style is voluntarily informal.

Keywords: Agreement · Asynchronous system · Atomicity · Concurrency · Consensus object · Consensus number · Crash failure · Distributed complexity · Distributed computability · Distributed computing · Environment · Fault-tolerance · Impossibility result · Indistinguishability · Message adversary · Message-passing system · Non-blocking · Obstruction-freedom · Progress condition · Read/write system · Synchronous system · Task · Universal construction · Wait-freedom

Molecular Computation with DNA Self-Assembly

Damien Woods

Inria, Paris, France

The field of algorithmic self-assembly is concerned with the theory and practice of molecules sticking together to grow computational structures in an autonomous bottom-up fashion. Theoretical work is concerned with characterising the computational expressiveness of self-assembly models. Practice is concerned with using molecules, such as DNA, to implement algorithmic self-assembly programs in the wet-lab. The presentation will cover both topics. First, there will be an introduction to what it means to compute during a self-assembly process, an overview of some computational models, as well as basic mathematical results. Attendees will then hear about how one goes about designing and experimentally implementing algorithmic self-assembling DNA tiles in the wet lab, and will see some of our latest results from recent joint work with David Doty, Cameron Myhrvold, Joy Hui, Felix Zhou, Peng Yin and Erik Winfree.

Contents

Control Mechanisms for Array Grammars on Cayley Grids 1
Rudolf Freund

A Pleasant Stroll Through the Land of Distributed Machines, Computation, and Universality 34
Michel Raynal and Jiannong Cao

Sequential Grammars with Activation and Blocking of Rules 51
Artiom Alhazov, Rudolf Freund, and Sergiu Ivanov

The Language (and Series) of Hammersley-Type Processes 69
Cosmin Bonchiş, Gabriel Istrate, and Vlad Rochian

Minimizing Rules and Nonterminals in Semi-conditional Grammars: Non-trivial for the Simple Case. 88
Henning Fernau, Lakshmanan Kuppusamy, Rufus O. Oladele, and Indhumathi Raman

Minimal Useful Size of Counters for (Real-Time) Multicounter Automata . . . 105
Viliam Geffert and Zuzana Bednárová

A Framework for (De)composing with Boolean Automata Networks 121
Kévin Perrot, Pacôme Perrotin, and Sylvain Sené

An Extension of Interval-Valued Computing Equivalent to Red-Green Turing Machines. 137
Benedek Nagy and Sándor Vályi

Physical Computation and First-Order Logic. 153
Richard Whyman

Author Index 171

Control Mechanisms for Array Grammars on Cayley Grids

Rudolf Freund(✉)

Faculty of Informatics, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
rudi@emcc.at

Abstract. In this paper, the computational power of several control mechanisms for specific variants of (sequential, isometric) array grammars generating arrays on Cayley grids of finitely presented groups is investigated. Using #-context-free array productions together with control mechanisms as control graphs, matrices, permitting and forbidden rules, partial order on rules or activation and blocking of rules the same computational power is obtained as when using arbitrary array productions.

1 Introduction

As a natural extension of string languages (e.g., see [31,32]), arrays on the d-dimensional grid $\mathbb{Z}^d$ have been introduced and investigated since more than four decades, for example, see [6,26]. Applications of array grammars and array automata especially can be found in the area of pattern and picture recognition, for instance, see [29,30,33].

Following some ideas of Csuhaj-Varjú and Mitrana [7], the investigation of arrays on Cayley grids of finitely presented groups was started in [22], presented at MCU 2013 in Zürich, Switzerland; first definitions and results for array automata on Cayley grids can be found there. Array grammars and automata on Cayley grids then were investigated in more detail in [23]. As a first example of arrays on a Cayley grid of a non-Abelian group we refer to [1], where arrays on the hexagonal grid were considered.

In this paper, first the notions and definitions for arrays defined on Cayley grids of finitely presented groups as well as for array grammars generating sets of such arrays are recalled from [23]. Following the general notions for regulated rewriting based on the applicability of rules as introduced in [21], then the control mechanisms using control graphs, matrices, permitting and forbidden rules, partial order on rules or activation and blocking of rules are defined. We elaborate some relations between these control mechanisms in the general setting of sequential grammars as already done in [21] and also prove some new ones. When using #-context-free array productions in the underlying array grammars, together with any of these control mechanisms, the same computational power as with arbitrary array productions can be obtained.

2 Preliminaries

The set of integers is denoted by $\mathbb{Z}$, the set of positive integers by $\mathbb{N}$, the set of non-negative integers by $\mathbb{N}_0$. An *alphabet* V is a non-empty set of abstract *symbols.* Given V, the free monoid generated by V under the operation of concatenation is denoted by V^*; the elements of V^* are called strings, and the *empty string* is denoted by λ; $V^* \setminus \{\lambda\}$ is denoted by V^+. The cardinality of a set M is denoted by $|M|$.

For the basic notions and results of formal language theory the reader is referred to the monographs and handbooks in this area as [8,31,32], and for the basics of group theory and group presentations to [25].

2.1 Groups and Group Presentations

Now let $G = (G', \circ)$ be a group with group operation $\circ$. As is well-known, the group axioms are

- *closure*: for any $a, b \in G'$, $a \circ b \in G'$,
- *associativity*: for any $a, b, c \in G'$, $(a \circ b) \circ c = a \circ (b \circ c)$,
- *identity*: there exists a (unique) element $e \in G'$, called the *identity*, such that $e \circ a = a \circ e$ for all $a \in G'$, and
- *invertibility:* for any $a \in G'$, there exists a (unique) element a^{-1}, called the *inverse* of a, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

Moreover, the group is called *commutative*, if for any $a, b \in G'$, $a \circ b = b \circ a$. In the following, we will not distinguish between G' and G if the group operation is obvious from the context.

For any element $b \in G'$, the order of b is the smallest number $n \in \mathbb{N}$ such that $b^n = e$ provided such an n exists, and then we write $ord\,(b) = n$; if no such n exists, $\{b^n \mid n \geq 1\}$ is an infinite subset of G' and we write $ord\,(b) = \infty$.

For any set B, B^{-1} is defined as the set of symbols representing the inverses of the elements of B, i.e., $B^{-1} = \left\{b^{-1} \mid b \in B\right\}$. We now consider the strings in $\left(B \cup B^{-1}\right)^*$ and two strings as different unless their equality follows from the group axioms, i.e., for any $a, b, c \in \left(B \cup B^{-1}\right)^*$, $abb^{-1}c = ac$; using these reductions, we obtain a set of irreducible strings from those in $\left(B \cup B^{-1}\right)^*$, the set of which we denote by $I\,(B)$. Then the *free group* generated by B is $F\,(B) = (I\,(B)\,, \circ)$ with the elements being the irreducible strings over $B \cup B^{-1}$ and the group operation to be interpreted as the usual string concatenation, yet, obviously, if we concatenate two elements from $I\,(B)$, the resulting string eventually has to be reduced again. The identity in $F\,(B)$ is the empty string.

In general, B (not containing the identity) is called a *generator* of the group G if every element a from G can be written as a finite product/sum of elements from B, i.e., $a = b_1 \circ \cdots \circ b_m$ for $b_1, \ldots, b_m \in B$. In this paper, we restrict ourselves to finitely presented groups, i.e., having a finite presentation $\langle B \mid R \rangle$ with B being a finite generator set and moreover, R being a finite set of relations among these generators. In a similar way as in the definition of the free group generated by B, we here consider the strings in B^* reduced according to the group axioms and the relations given in R. Informally, the group $G = \langle B \mid R\rangle$ is the largest one generated by B subject only to the group axioms and the relations in R. Formally, we will restrict ourselves to relations of the form $b_1 \circ \cdots \circ b_m = c^{-1}$ with $b_1, \ldots, b_m, c \in B$, which equivalently may be written as $b_1 \circ \cdots \circ b_m \circ c = e$; hence, instead of such relations we may specify R by strings over B yielding the group identity, i.e., instead of $b_1 \circ \cdots \circ b_m = c^{-1}$ we take $b_1 \circ \cdots \circ b_m \circ c$ (these strings then are called *relators*).

Example 1. The free group $F(B) = (I(B), \circ)$ can be written as $\langle B \mid \emptyset\rangle$ (or even simpler as $\langle B\rangle$) because it has no restricting relations.

Example 2. The *cyclic group* of order n has the presentation $\langle\{a\} \mid \{a^n\}\rangle$ (or, omitting the set brackets, written as $\langle a \mid a^n\rangle$); it is also known as $\mathbb{Z}_n$ or as the quotient group $\mathbb{Z}/\mathbb{Z}_n$.

Example 3. $\mathbb{Z}$ is a special case of an Abelian group generated by (1) and its inverse (-1), i.e., $\mathbb{Z}$ is the free group generated by (1). $\mathbb{Z}^d$ is an Abelian group generated by the unit vectors $(0, \ldots, 1, \ldots, 0)$ and their inverses $(0, \ldots, -1, \ldots, 0)$. It is well known that every finitely generated Abelian group is a direct sum of a torsion group and a free Abelian group where the torsion group may be written as a direct sum of finitely many groups of the form $\mathbb{Z}/p^k\mathbb{Z}$ for p being a prime, and the free Abelian group is a direct sum of finitely many copies of $\mathbb{Z}$.

Remark 1. Given a finite presentation of a group $\langle B \mid R\rangle$, in general it is not even decidable whether the group presented in that way is finite or infinite. If we consider (infinite) groups where the word equivalence problem $u = v$ is decidable, or equivalently, there is a decision procedure telling us whether, given two strings u and v, $uv^{-1} = e$, then we call $\langle B \mid R\rangle$ a *recursive* or *computable* finite group presentation.

2.2 Cayley Graphs

Let $G = \langle B \mid R\rangle$ be a finitely presented group with G' denoting the set of group elements. Then we define the corresponding Cayley graph of G with respect to the generating set B as the directed graph $C(G, B) = (G', E)$ with the set of nodes G' and the set E of directed edges labeled by elements of B by $E = \{(x, a, y) \mid x, y \in G', a \in B, xa = y\}$, i.e., from an element x an edge labeled by the generator a leads to y if and only if $xa = y$.

As can be seen directly from the definition, the Cayley graph for a group G depends on its presentation by the generator set B and the relators in R.

Example 4. The dihedral group D_∞ corresponds with the Cartesian product of $\mathbb{Z}$ and $\mathbb{Z}_2$. One presentation of D_∞ is as $\left\langle r, s \mid s^2, (sr)^2\right\rangle$, another one is $\left\langle r, s \mid r^2, s^2\right\rangle$, the Cayley graph of which can be represented easily in the following way:

$$\begin{tikzcd}
\ldots srs \arrow[r, shift left, "s"] & sr \arrow[l, shift left, "s"] \arrow[r, shift left, "r"] & s \arrow[l, shift left, "r"] \arrow[r, shift left, "s"] & e \arrow[l, shift left, "s"] \arrow[r, shift left, "r"] & r \arrow[l, shift left, "r"] \arrow[r, shift left, "s"] & rs \arrow[l, shift left, "s"] \arrow[r, shift left, "r"] & rsr \ldots \arrow[l, shift left, "r"]
\end{tikzcd}$$

In this presentation, both generators have order two; on the other hand, an infinite line can be obtained by taking the group element rs and its powers $(rs)^n$ for $n \geq 0$, as the order of rs is infinite.

The Cayley graph for the presentation of D_∞ as $\langle r, s \mid s^2, srsr \rangle$ can be depicted as follows:

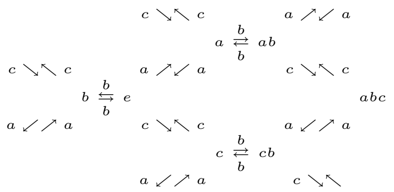
$$\begin{tikzcd}
\ldots & sr^2 \arrow[d, shift right, "s"'] & sr \arrow[l, "r"'] \arrow[d, shift right, "s"'] & s \arrow[l, "r"'] \arrow[d, shift right, "s"'] & sr^{-1} \arrow[l, "r"'] \arrow[d, shift right, "s"'] & sr^{-2} \arrow[l, "r"'] \arrow[d, shift right, "s"'] & \ldots \\
\ldots & r^{-2} \arrow[r, "r"] \arrow[u, shift right, "s"'] & r^{-1} \arrow[r, "r"] \arrow[u, shift right, "s"'] & e \arrow[r, "r"] \arrow[u, shift right, "s"'] & r \arrow[r, "r"] \arrow[u, shift right, "s"'] & r^2 \arrow[u, shift right, "s"'] & \ldots
\end{tikzcd}$$

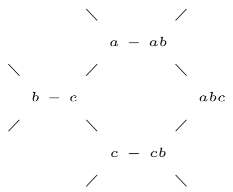
As s is self-inverse, instead of the two directed edges $s \downharpoonleft\upharpoonright s$ often only the corresponding non-directed edge $\mid s$ is depicted, i.e.,

$$\begin{tikzcd}
\ldots sr^2 \arrow[d, dash, "s"] & sr \arrow[l, "r"'] \arrow[d, dash, "s"] & s \arrow[l, "r"'] \arrow[d, dash, "s"] & sr^{-1} \arrow[l, "r"'] \arrow[d, dash, "s"] & sr^{-2} \ldots \arrow[l, "r"'] \arrow[d, dash, "s"] \\
\ldots r^{-2} \arrow[r, "r"] & r^{-1} \arrow[r, "r"] & e \arrow[r, "r"] & r \arrow[r, "r"] & r^2 \ldots
\end{tikzcd}$$

The lower and the upper lines are going into opposite directions, which nicely fits as a representation of double-stranded DNA molecules, i.e., the lower line going from the left 5′-end to the right 3′-end, whereas the complementary upper line goes from the right 5′-end to the left 3′-end.

Example 5. The hexagonal grid is the Cayley graph assigned to the presentation of the group $\langle a, b, c \mid a^2, b^2, c^2, (abc)^2 \rangle$. As all three generators a, b, c are self-inverse and the direction of these elements indicates which generator is meant, we obtain a simpler picture for the hexagonal grid by replacing $a \nearrow\swarrow a$, $\overset{b}{\underset{b}{\rightleftarrows}}$, and $c \searrow\nwarrow c$ by $\diagup$, $-$, and $\diagdown$, respectively. Both representations are depicted in the following:





3 Arrays and Array Grammars

In this section we generalize the concept of d-dimensional arrays to arrays defined on Cayley grids. Let $G = \langle B \mid R \rangle$ be a finitely presented group with $B = \{e_1, \ldots, e_m\}$ and G' denoting the set of group elements; moreover, let $C(G)$ be the Cayley graph of G with respect to B. Throughout the paper we will assume that $B^{-1} \subseteq B$, i.e., B contains all inverses of its elements. For paths in the Cayley graph this means that for each path $v = w_1 \to \ldots \to w_n = w$ in $C(G)$ from v to w also its inverse $w = w_n \to \ldots \to w_1 = v$ is a path in $C(G)$.

A finite *array* $\mathcal{A}$ over an alphabet V on G' is a function $\mathcal{A} : G' \to V \cup \{\#\}$, where $shape(\mathcal{A}) = \{v \in G' \mid \mathcal{A}(v) \neq \#\}$ is finite and $\# \notin V$ is called the *background* or *blank symbol*, i.e., the nodes of $C(G)$ get assigned elements of $V \cup \{\#\}$. We usually will write $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in \mathit{shape}(\mathcal{A})\}$.

By V^G we denote the set of arrays over V on G', any subset of $V^{\langle G\rangle}$ is called an array language over V on G. With respect to the finite presentation of G by $C(G)$, instead of V^G we also write $V^{C(G)}$.

The *empty array* in V^G has empty shape and is denoted by Λ_G. Ordering the generators in B in a specific way as $e_1 < \cdots < e_m$, for each array $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in \mathit{shape}(\mathcal{A})\}$ in $V^{\langle G\rangle} \setminus \{\Lambda_G\}$ we get a canonical representation as a list $\langle (v_1, \mathcal{A}(v_1)), \ldots, (v_n, \mathcal{A}(v_n))\rangle$ such that $\{v_i \mid 1 \leq i \leq n\} = \mathit{shape}(\mathcal{A})$ and $v_i < v_{i+1}$, $1 \leq i < n$, with respect to the length-plus-lexicographic ordering of strings with the elements of G written as sums of the elements in B (the length-plus-lexicographic ordering $\prec$ is a well-ordering, where for two strings u and v, $u \prec v$ if either $|u| < |v|$ or $|u| = |v|$, $u = xay$, $v = xby$, and $a < b$). In terms of $C(G)$ this means that the elements of the array are listed in the length-plus-lexicographic ordering of the paths in $C(G)$ seen from the origin (the identity).

Example 6. Consider the hexagonal grid from Example 5. Then the "position" abc can also be reached by taking the path cba from the "origin" (the identity e). Hence, with taking the ordering $a < b < c$, the canonical representation of the array $\mathcal{A} = \{(ab, X), (abc, Y) \mid v \in \mathit{shape}(\mathcal{A})\} \in \{X, Y\}^{C(\langle a,b,c \mid a^2, b^2, c^2, (abc)^2\rangle)}$ is $\langle (ab, X), (abc, Y)\rangle$.

Example 7. A *d-dimensional array* is an array over the free group $\mathbb{Z}^d$. If we take the unit vectors $e_k = (0, \ldots, 1, \ldots, 0)$ and their inverses $(0, \ldots, -1, \ldots, 0)$, the resulting Cayley graph is the well-known d-dimensional grid, which in the 2-dimensional case can be depicted in the following way, where each horizontal line $-$ represents the two directed edges $\overset{(1,0)}{\underset{(-1,0)}{\rightleftarrows}}$ and each vertical line $|$ represents the two directed edges $(0,-1) \downharpoonleft\upharpoonright (0,1)$:

$$\begin{array}{ccccccccccc}
 & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & \\
 & & | & & | & & | & & | & & | & \\
\cdots - & & (-2,1) & - & (-1,1) & - & (0,1) & - & (1,1) & - & (2,1) & - \cdots \\
 & & | & & | & & | & & | & & | & \\
\cdots - & & (-2,0) & - & (-1,0) & - & (0,0) & - & (1,0) & - & (2,0) & - \cdots \\
 & & | & & | & & | & & | & & | & \\
\cdots - & & (-2,-1) & - & (-1,-1) & - & (0,-1) & - & (1,-1) & - & (2,-1) & - \cdots \\
 & & | & & | & & | & & | & & | & \\
 & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots &
\end{array}$$

With respect to the origin $(0,0)$, the four vectors $(1,0), (-1,0), (0,1), (0,-1)$ are known as the *von Neumann neighborhood*, whereas adding the diagonal positions $(1,1), (-1,1), (-1,-1), (1,-1)$ yields the *Moore neighborhood* and thus a different Cayley grid based on these eight generators for $\mathbb{Z}^2$:

$$\begin{tikzcd}
& \vdots & \vdots & \vdots & \\
\cdots & (-1,1) \arrow[r, dash] \arrow[d, dash] \arrow[dr, dash] & (0,1) \arrow[r, dash] \arrow[d, dash] & (1,1) \arrow[r, dash] \arrow[d, dash] \arrow[dl, dash] & \cdots \\
\cdots & (-1,0) \arrow[r, dash] \arrow[d, dash] & (0,0) \arrow[r, dash] \arrow[d, dash] \arrow[dl, dash] \arrow[dr, dash] & (1,0) \arrow[r, dash] \arrow[d, dash] & \cdots \\
\cdots & (-1,-1) \arrow[r, dash] & (0,-1) \arrow[r, dash] & (1,-1) \arrow[r, dash] & \cdots \\
& \vdots & \vdots & \vdots &
\end{tikzcd}$$

For any $v \in G'$, the *translation* $\tau_v : G' \to G'$ is defined by $\tau_v(w) = w \circ v$ for all $w \in G'$, and for any array $\mathcal{A} \in V^{C(G)}$ we define $\tau_v(\mathcal{A})$, the corresponding array translated by v, by

$$(\tau_v(\mathcal{A}))(w) = \mathcal{A}\left(w \circ v^{-1}\right) \text{ for all } w \in G'.$$

An array $\mathcal{A} \in V^{C(G)}$ is called *k-connected* if for any two elements v and w in *shape* $(\mathcal{A})$ there is a path $v = w_1 \to \cdots \to w_n = w$ in $C(G)$ with $\{w_1, \ldots, w_n\} \subseteq$ *shape* $(\mathcal{A})$ such that for the distance in $C(G)$ between w_i and w_{i-1}, $d(w_i, w_{i-1})$, we have $d(w_i, w_{i-1}) \leq k$ for all $1 < i \leq n$; the distance $d(x,y)$ between two nodes x and y in $C(G)$ is defined as the length of the shortest path between x and y in $C(G)$. The subset of k-connected arrays in $V^{C(G)}$ is denoted by $V^{C(G)_k}$.

Example 8. Consider the set of one-dimensional arrays over the alphabet $\{a\}^*$, i.e., $\{a\}^{C(\langle(1),(-1)\rangle)}$, which in a simpler way we will also write as $\{a\}^{\mathbb{Z}^1}$. Then the 1-dimensional array $\{((0), a), ((k), a)\} \in \{a\}^{\mathbb{Z}^1}$ is m-connected, i.e., in $\{a\}^{\mathbb{Z}^1{}_m}$, if and only if $m \geq k$.

3.1 Array Grammars

For a finitely presented group $G = \langle B \mid R \rangle$ with the set of elements G', we define an *array production* p over V and G as a triple $(W, \mathcal{A}_1, \mathcal{A}_2)$, where $W \subseteq G'$ is a finite set and $\mathcal{A}_1$ and $\mathcal{A}_2$ are mappings from W to $V \cup \{\#\}$ such that *shape* $(\mathcal{A}_1) \neq \emptyset$, where again the shape is defined to exactly contain the non-blank positions, i.e., *shape*$(\mathcal{A}_1) = \{v \in W \mid \mathcal{A}(v) \neq \#\}$. We say that the array $\mathcal{C}_2 \in V^{C(G)}$ is *directly derivable* from the array $\mathcal{C}_1 \in V^{C(G)}$ by the array production $(W, \mathcal{A}_1, \mathcal{A}_2)$ if and only if there exists a $v \in G'$ such that, for all $w \in G' \setminus \tau_v(W)$, $\mathcal{C}_1(w) = \mathcal{C}_2(w)$, as well as, for all $w \in \tau_v(W)$, $\mathcal{C}_1(w) = \mathcal{A}_1(\tau_{-v}(w))$ and $\mathcal{C}_2(w) = \mathcal{A}_2(\tau_{-v}(w))$, i.e., the sub-array of $\mathcal{C}_1$ corresponding to $\mathcal{A}_1$ is replaced by $\mathcal{A}_2$, thus yielding $\mathcal{C}_2$; we also write $\mathcal{C}_1 \Longrightarrow_p \mathcal{C}_2$.

As we already see from the definitions of an array production, the conditions for an application to an array $\mathcal{B}$ and the result of an application to $\mathcal{B}$, an array production $(W, \mathcal{A}_1, \mathcal{A}_2)$ is a representative for the infinite set of equivalent array productions of the form $(\tau_v(W), \tau_v(\mathcal{A}_1), \tau_v(\mathcal{A}_2))$ with $v \in G'$. Hence, without loss of generality, we can assume $e \in W$ (e is the identity in G) as well as $\mathcal{A}_1(e) \neq \#$. Moreover, we often will omit the set W, because it is uniquely reconstructible from the description of the two mappings $\mathcal{A}_1$ and

$\mathcal{A}_2$ by $\mathcal{A}_i = \{ (v, \mathcal{A}_i(v)) \mid v \in W \}$, for $1 \leq i \leq 2$. Thus, in the following, we represent the array production $(W, \mathcal{A}_1, \mathcal{A}_2)$ also by writing $\mathcal{A}_1 \to \mathcal{A}_2$, i.e., $\{(v, \mathcal{A}_1(v)) \mid v \in W\} \to \{(v, \mathcal{A}_2(v)) \mid v \in W\}$. If $|W| = 2$, i.e., $W = \{e, v\}$ for some $v \in G'$, then, for $\{(e, \mathcal{A}_1(e)), (v, \mathcal{A}_1(v))\} \to \{(e, \mathcal{A}_2(e)), (v, \mathcal{A}_2(v))\}$ we will only write $\mathcal{A}_1(e) v \mathcal{A}_1(v) \to \mathcal{A}_2(e) \mathcal{A}_2(v)$. If $|W| = 1$, i.e., $W = \{e\}$, we simply write $\mathcal{A}_1(e) \to \mathcal{A}_2(e)$.

An *array grammar* (over $C(G)$) is a septuple

$$G_A = (C(G), N, T, \#, P, \mathcal{A}_0, \Longrightarrow_{G_A}),$$

where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, $\# \notin N \cup T$; P is a finite non-empty set of array productions over V, where $V = N \cup T$; $\mathcal{A}_0 \in V^{C(G)}$ is the *initial array* (axiom), for which, as usually done in the literature, we shall assume $\mathcal{A}_0 = \{(v_0, S)\}$, where $v_0 \in G'$ is the *start node*, and $S \in N$ is the *start symbol*. Moreover, $\Longrightarrow_{G_A}$ denotes the derivation relation induced by the array productions in P. In the examples given below, we will omit $\Longrightarrow_{G_A}$ in the description of the array grammars.

We say that the array $\mathcal{B}_2 \in V^{C(G)}$ is *directly derivable* from the array $\mathcal{B}_1 \in V^{C(G)}$ in G_A, denoted $\mathcal{B}_1 \Longrightarrow_{G_A} \mathcal{B}_2$, if and only if there exists an array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P such that $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$. Let $\Longrightarrow_{G_A}^*$ be the reflexive transitive closure of $\Longrightarrow_{G_A}$. The *array language generated* by the array grammar G_A, $L(G_A)$, is defined by

$$L(G_A) = \left\{ \mathcal{A} \mid \mathcal{A} \in T^{C(G)},\ \mathcal{A}_0 \Longrightarrow_{G_A}^* \mathcal{A} \right\}.$$

An array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P is called

- $\#$*-context-free* (*of type* $\#$*-CFA*), if $|shape(\mathcal{A}_1)| = 1$, i.e., $shape(\mathcal{A}_1) = \{e\}$, and $\mathcal{A}_1(e) \in N$;
- *context-free* (*of type CFA*), if it is of type $\#$-CFA and $\mathcal{A}_2(e) \neq \#$;
- *strictly context-free* (*of type SCFA*), if it is of type $\#$-CFA and $shape(\mathcal{A}_2) = W$.

For $X \in \{ARBA, \#\text{-}CFA, CFA, SCFA\}$, an array grammar G is called to be of type X, if every array production in P is of the corresponding type, where $ARBA$ means that there are no restrictions on the form of the array productions. The family of k-connected array languages generated by array grammars on $C(G)$ of type X is denoted by $\mathcal{L}_k(C(G)\text{-}X)$; the family of arbitrary array languages generated by array grammars on $C(G)$ of type X is denoted by $\mathcal{L}(C(G)\text{-}X)$.

The main difference between array grammars of type $\#$-CFA and of type CFA is that already by the definition of the array productions of type CFA it is guaranteed that all intermediate arrays derived from the initial arrays as well as the terminal arrays are k-connected, if all W in the array productions $(W, \mathcal{A}_1, \mathcal{A}_2)$ are k-connected due to the condition that no symbol can be *erased*, i.e., replaced by the blank symbol $\#$. On the other hand, array productions of type $\#$-CFA allow symbols to move as far as they want from their original position:

Example 9. Consider the #-context-free 1-dimensional array grammar

$$G_1 = \left(\mathbb{Z}^1, N = \{S, A\}, T = \{a\}, \#, P, \mathcal{A}_0 = \{((0), S)\}\right),$$
$$P = \{(0)S(1)\# \to (0)a(1)A, (0)A(1)\# \to (0)\#(1)A, (0)A \to (0)a\}.$$

According to our conventions, in a simpler way we can write

$$P = \{S(1)\# \to aA, A(1)\# \to \#A, A \to a\}.$$

The array language generated by G_1 is the subset of $\{a\}^{\mathbb{Z}^1}$ that can be written as $\{(0)a(m)a \mid m \in \mathbb{N}\}$ and thus for no k and no type X of array grammars is in $\mathcal{L}_k\left(\mathbb{Z}^1 - X\right)$.

For arbitrary and #-context-free array grammars the condition to only consider languages of k-connected arrays corresponds to intersecting the generated array language with $V^{C(G)_k}$, which can be carried out by arbitrary array grammars by themselves (which will be proved later, see Lemma 2), but is a condition imposed from "outside" when dealing with #-context-free array grammars. Yet as later we are going to show that some #-context-free array grammars equipped with specific control mechanisms can simulate any arbitrary array grammar this makes no difference any more.

Example 10. Let $G = \langle B \mid R\rangle$ be a finitely presented group and $x \in G$ with $ord\,(x) = \infty$. Let $b_1 \circ \ldots \circ b_k$ be the canonical representation of x in $\langle B \mid R\rangle$; then $(\{x^n \mid n \in \mathbb{Z}\}, \circ)$ is an infinite subgroup of G, and $x^n \neq x^m$ for $n \neq m$. Hence, along this "line" we can argue many results obtained for $\mathbb{Z}^1$, e.g., we can argue that, for any Cayley grid $C\left(\langle B \mid R\rangle\right)$,

$$\mathcal{L}\left(C\left(\langle B \mid R\rangle\right), CFA\right) \subset \mathcal{L}\left(C\left(\langle B \mid R\rangle\right), ARBA\right),$$

because the inclusion directly follows from the definitions, and the strictness follows from the well-known corresponding result for string languages using as a witness the language $(L) = \left\{a^{2^n} \mid n \in \mathbb{N}\right\}$ and the representation of the strings in it as 1-dimensional arrays. As a small technical detail we have to mention that for $x = b_1 \circ \ldots \circ b_k$, $b_1, \ldots, b_k \in B$, such witness languages have to be expanded by the homomorphism h_k with $h_k\,(a) = a^k$ for every symbol a in the alphabet, as in $C\left(\langle B \mid R\rangle\right)$ we now have to fill k positions instead of only one in $\mathbb{Z}^1$.

Such infinite lines can be found in various Cayley graphs described so far. For example, consider the presentation of D_∞ as $\left\langle r, s \mid r^2, s^2\right\rangle$ from Example 4; its Cayley graph $\ldots srs \underset{s}{\overset{s}{\rightleftarrows}} sr \underset{r}{\overset{r}{\rightleftarrows}} s \underset{s}{\overset{s}{\rightleftarrows}} e \underset{r}{\overset{r}{\rightleftarrows}} r \underset{s}{\overset{s}{\rightleftarrows}} rs \underset{r}{\overset{r}{\rightleftarrows}} rsr \ldots$ can somehow be seen as the line $(rs)^n$, $n \in \mathbb{Z}$, when only taking the elements having a canonical representation of even length.

Remark 2. The possibility to compute along such infinite lines is also important if we want to (describe how to) simulate computations of a Turing machine – or similar computationally complete mechanisms (for strings) – using specific variants of (controlled) array grammars on Cayley graphs. For instance, for any computable finite group presentation of a group $\langle B \mid R\rangle$, we can effectively construct an encoding of any array language in $\mathcal{L}\left(C\left(G\right)\text{-}ARBA\right)$ given by an (arbitrary) array grammar and vice versa. The finite group presentation of the group $\langle B \mid R\rangle$ being computable is crucial for this result.

For simulating array grammars of type $C\left(G\right)\text{-}ARBA$, a special normal form we call *marked normal form* is very helpful; it has already been described for 1-dimensional array grammars in [20] as a special variant of the Chomsky normal form for array grammars, shown, for example, in [16].

Lemma 1 (marked normal form)**.** *For every array grammar of type* $C\left(G\right)\text{-}ARBA$

$$G_A = \left(C\left(G\right), N, T, \#, P, \left\{\left(v_0, S\right)\right\}, \Longrightarrow_{G_A}\right),$$

we can effectively construct an equivalent array grammar of type $C\left(G\right)\text{-}ARBA$

$$\bar{G}_A = \left(C\left(G\right), N', T, \#, P', \left\{\left(v_0, \bar{S}\right)\right\}, \Longrightarrow_{\bar{G}_A}\right),$$

such that $N \subseteq N'$ *and all array productions in* P' *are of one of the following forms:*

1. $\bar{A}B \to C\bar{D}$*, where* $A, B, C, D \in N' \cup T$*,* or
2. $\bar{\#} \to \#$*.*

Before the final array production $\bar{\#} \to \#$ *is applied, any intermediate array derived from the initial array* $\left\{\left(v_0, \bar{S}\right)\right\}$ *contains exactly one barred symbol.*

We omit the proof as the arguments given in [20] for 1-dimensional array grammars can be taken over for the general case of arbitrary array grammars over Cayley grids.

We now exhibit the promised algorithm how array grammars with arbitrary rules can filter out the terminal arrays which are k-connected.

Lemma 2 (filtering out all k-connected arrays)**.** *Let* $k \in \mathbb{N}$*. For every array grammar of type* $C\left(G\right)\text{-}ARBA$

$$G_A = \left(C\left(G\right), N, T, \#, P, \left\{\left(v_0, S\right)\right\}, \Longrightarrow_{G_A}\right),$$

we can effectively construct an equivalent array grammar of type $C\left(G\right)\text{-}ARBA$

$$G'_A = \left(C\left(G\right), N', T, \#, P', \left\{\left(v_0, S'\right)\right\}, \Longrightarrow_{G'_A}\right),$$

such that $L\left(G'_A\right)$ *contains exactly those arrays from* $L\left(G_A\right)$ *that are* k*-connected.*

Proof (sketch). According to Lemma 1, without loss of generality we may assume that G_A is in the marked normal form. First, we replace every terminal symbol $a \in T$ by a corresponding non-terminal symbol X_a in all the array productions from P, which allows us to simulate any derivation of G_A in a suitable way with the only difference that instead of the terminal symbols $a \in T$ we have the corresponding non-terminal symbols X_a in all arrays occurring in any derivation.

Finally, instead of applying the final rule $\bar{\#} \to \#$ we move the bar to a symbol X_a and apply the rule $\bar{X}_a \to a$. This terminal seed a at some position v_0 now may propagate the signal *become terminal* to all positions v in the array derived so far which allow for a k-connected path on non-blank symbols from v_0 to v by using the rules of the form $buX_c \to bc$, $b, c \in T$, and any u being an element from the underlying group reachable from e in at most k steps in $C(G)$. This condition guarantees that when no rule is applicable any more, the obtained subarray only containing terminal symbols is k-connected, hence, if no non-terminal symbol has remained, the final array is terminal *and* k-connected.

As a technical detail we mention that to obtain the empty array we immediately apply the array production $\bar{\#} \to \#$. $\square$

Remark 3. The idea of first working with non-terminal symbols X_a instead of terminal symbols a and then turning them into the corresponding terminal symbols a can be taken over for any arbitrary array grammar. Hence, without loss of generality, we may always assume that any array production contains at least one non-terminal symbol in the array on its left-hand side, i.e., in any array production $\{(v, \mathcal{A}_1(v)) \mid v \in W\} \to \{(v, \mathcal{A}_2(v)) \mid v \in W\}$ we find at least one $v_1 \in W$ such that $\mathcal{A}_1(v_1) \in N$.

Therefore, throughout the rest of the paper, when using the notion of an array grammar of type $C(G)$-$ARBA$ we will assume this condition to hold.

4 Standard Control Mechanisms

In this section we recall the notions and basic results for the general model of sequential grammars equipped with specific control mechanisms as elaborated in [21], based on the applicability of rules, as well as for the new concept of activation and blocking of rules as exhibited in [3,4].

Although in this paper we are only dealing with array grammars (over a Cayley graph $C(G)$), the control mechanisms will be defined for arbitrary sequential grammars; hence, we first recall the notion of a general model for sequential grammars, and then also the control mechanisms are introduced for this general model.

4.1 A General Model for Sequential Grammars

We first recall the main definitions of the general model for sequential grammars as established in [21], grammars generating a set of terminal objects by derivations where in each derivation step exactly one rule is applied to exactly one object.

A *(sequential) grammar* G_s is a construct $(O, O_T, w, P, \Longrightarrow_{G_s})$ where

- O is a set of *objects*;
- $O_T \subseteq O$ is a set of *terminal objects*;

- $w \in O$ is the *axiom (start object)*;
- P is a finite set of *rules*;
- $\Longrightarrow_{G_s} \subseteq O \times O$ is the *derivation relation* of G_s.
  Each of the rules $p \in P$ induces a relation $\Longrightarrow_p \subseteq O \times O$ with respect to $\Longrightarrow_{G_s}$. A rule $p \in P$ is called *applicable* to an object $x \in O$ if and only if there exists at least one object $y \in O$ such that $(x, y) \in \Longrightarrow_p$; we also write $x \Longrightarrow_p y$. The derivation relation $\Longrightarrow_{G_s}$ is the union of all $\Longrightarrow_p$, i.e., $\Longrightarrow_{G_s} = \cup_{p \in P} \Longrightarrow_p$. The reflexive and transitive closure of $\Longrightarrow_{G_s}$ is denoted by $\overset{*}{\Longrightarrow}_{G_s}$.

Specific conditions on the rules in P define a special type X of grammars which then will be called *grammars of type X*.

The *language generated by G* is the set of all terminal objects that can be derived from the axiom, i.e.,

$$L(G_s) = \left\{ v \in O_T \mid w \overset{*}{\Longrightarrow}_{G_s} v \right\}.$$

The family of languages generated by grammars of type X is denoted by $\mathcal{L}(X)$.

Let $G_s = (O, O_T, w, P, \Longrightarrow_{G_s})$ be a (sequential) grammar of type X. If for every G_s of type X we have $O_T = O$, then X is called a *pure* type, otherwise it is called *extended*; X is called *strictly extended* if for any grammar G_s of type X, $w \notin O_T$ and for all $x \in O_T$, no rule from P can be applied to x.

In many cases, the type X of the grammar allows for one or even both of the following features:

A type X of grammars is called a *type with unit rules* if for every grammar $G_s = (O, O_T, w, P, \Longrightarrow_G)$ of type X there exists a grammar $G'_s = \left(O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'_s}\right)$ of type X such that $\Longrightarrow_{G_s} \subseteq \Longrightarrow_{G'_s}$ and

- $P^{(+)} = \left\{p^{(+)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(+)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(+)}$ is applicable to x, the application of $p^{(+)}$ to x yields x back again.

A type X of grammars is called a *type with trap rules* if for every grammar $G_s = (O, O_T, w, P, \Longrightarrow_G)$ of type X there exists a grammar $G'_s = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'_s}\right)$ of type X such that $\Longrightarrow_{G_s} \subseteq \Longrightarrow_{G'_s}$ and

- $P^{(-)} = \left\{p^{(-)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

In the following, we shall deal with array grammars of type $C(G)$-*ARBA* and $C(G)$-#-*CFA*. In the general framework of sequential grammars as defined above, an array grammar over $C(G)$ of type *ARBA* or #-*CFA* originally defined as

$$G_A = (C(G), N, T, \#, P, \mathcal{A}_0, \Longrightarrow_{G_A})$$

should be written as

$$G_A = \left((N \cup T)^{C(G)}, T^{C(G)}, \mathcal{A}_0, P, \Longrightarrow_{G_A} \right)$$

which should be kept in mind for the definitions of the control mechanisms given below.

For applying the general results on the relation between different control mechanisms as elaborated in the rest of this section to array grammars of the types $C(G)$-$ARBA$ and $C(G)$-$\#$-CFA, the following feature of these types is essential in some cases:

Lemma 3. *The types $C(G)$-$ARBA$ and $C(G)$-$\#$-CFA for array grammars over a Cayley grid $C(G)$ are strictly extended types with unit rules and trap rules.*

Proof. According to Remark 3, $C(G)$-$ARBA$ can be seen as a strictly extended type for the succeeding proofs; $C(G)$-$\#$-CFA is a strictly extended type already by definition.

Now let

$$G_A = (C(G), N, T, \#, P, \{(v_0, S)\}, \Longrightarrow_{G_A})$$

be an array grammar of type $C(G)$-$ARBA$ or $C(G)$-$\#$-CFA.

Then for every array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ the corresponding *unit rule* is $p^+ = (W, \mathcal{A}_1, \mathcal{A}_1)$, which, when being applied, obviously does not change the underlying array.

Moreover, for the trap rules, take a new non-terminal symbol F, the *trap symbol*, which never can be erased any more, and for every array production $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ we then define the corresponding *trap rule* $p^- = (W, \mathcal{A}_1, \mathcal{F}_W)$ with $\mathcal{F}_W(v) = F$ for all $v \in W$, which, when being applied, prohibits the derived array to become terminal no matter how the derivation proceeds.

In sum, we conclude that both $C(G)$-$ARBA$ and $C(G)$-$\#$-CFA are strictly extended types with unit rules and trap rules. □

Remark 4. The constructions given in the preceding proof verbatim hold true for the type $C(G)$-CFA, as the additional restriction that the non-terminal symbol in the array on the left-hand side of the array production must not be replaced by the blank symbol $\#$ does not affect the validity of the construction, hence, $C(G)$-CFA is a strictly extended type with unit rules and trap rules, too.

On the other hand, $C(G)$-$SCFA$ only is a strictly extended type with trap rules, as for a rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ with $|W| > 1$ no unit rule p^+ not changing the underlying array can be found, as this would violate the condition that all positions $\neq e$ in W have to be occupied by non-blank symbols in $\mathcal{A}_2$.

4.2 Graph-Controlled and Programmed Grammars

A *graph-controlled grammar* (with applicability checking) of type X is a construct

$$G_{GC} = (G_s, g, H_i, H_f, \Longrightarrow_{GC})$$

where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X; $g = (H, E, K)$ is a labeled graph where H is the set of node labels identifying the nodes of the graph in a one-to-one manner, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and $H_f \subseteq H$ is the set of final labels. The derivation relation $\Longrightarrow_{GC}$ is defined based on $\Longrightarrow_{G_s}$ and the control graph g as follows: For any $i, j \in H$ and any $u, v \in O$, $(u, i) \Longrightarrow_{GC} (v, j)$ if and only if

– $u \Longrightarrow_p v$ by some rule $p \in K(i)$ and $(i, Y, j) \in E$ *(success case)*, **or**
– $u = v$, no $p \in K(i)$ is applicable to u, and $(i, N, j) \in E$ *(failure case)*.

The language generated by G_{GC} is defined by

$$L(G_{GC}) = \left\{ v \in O_T \mid (w, i) \Longrightarrow_{G_{GC}}^{*} (v, j),\ i \in H_i, j \in H_f \right\}.$$

If $H_i = H_f = H$, then G_{GC} is called a *programmed grammar*. The families of languages generated by graph-controlled and programmed grammars of type X are denoted by $\mathcal{L}(X\text{-}GC_{ac})$ and $\mathcal{L}(X\text{-}P_{ac})$, respectively. If the set E contains no edges of the form (i, N, j), then the graph-controlled grammar is said to be *without applicability checking*; the corresponding families of languages are denoted by $\mathcal{L}(X\text{-}GC)$ and $\mathcal{L}(X\text{-}P)$, respectively.

As a special variant of graph-controlled grammars we consider those where all labels are final; the corresponding family of languages generated by graph-controlled grammars of type X is abbreviated by $\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right)$. By definition, programmed grammars are just a subvariant where in addition all labels are also initial.

The notions *with/without applicability checking* in the original definition for string grammars were introduced as *with/without appearance checking* because the appearance of the non-terminal symbol on the left-hand side of a context-free rule was checked, which coincides with checking for the applicability of this rule in our general model; in both cases – applicability checking and appearance checking – we can use the abbreviation *ac*.

4.3 Matrix Grammars

A *matrix grammar* (with applicability checking) of type X is a construct

$$G_M = (G_s, M, F, \Longrightarrow_{G_M})$$

where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, M is a finite set of sequences of the form $(p_1, \dots, p_n)$, $n \geq 1$, of rules in P, and $F \subseteq P$. For $w, z \in O$ we write $w \Longrightarrow_{G_M} z$ if there are a matrix $(p_1, \dots, p_n)$ in M and objects $w_i \in O$, $1 \leq i \leq n + 1$, such that $w = w_1$, $z = w_{n+1}$, and, for all $1 \leq i \leq n$, either

– $w_i \Longrightarrow_{G_s} w_{i+1}$ or
– $w_i = w_{i+1}$, p_i is not applicable to w_i, and $p_i \in F$.

$L(G_M) = \{v \in O_T \mid w \Longrightarrow_{G_M}^{*} v\}$ is the language generated by G_M. The family of languages generated by matrix grammars of type X is denoted by $\mathcal{L}(X\text{-}MAT_{ac})$. If the set F is empty, then the grammar is said to be *without applicability checking* (*without ac* for short); the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}MAT)$.

We mention that in this paper we choose the definition where the sequential application of the rules of the final matrix may stop at any moment.

4.4 Random-Context Grammars

A *random-context grammar G_{RC} of type X* is a construct

$$(G_s, P', \Longrightarrow_{G_{RC}})$$

where

- $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
- P' is a set of rules of the form (q, R, Q) where $q \in P$, $R \cup Q \subseteq P$;
- $\Longrightarrow_{G_{RC}}$ is the derivation relation assigned to G_{RC} such that for any $x, y \in O$, $x \Longrightarrow_{G_{RC}} y$ if and only if for some rule $(q, R, Q) \in P'$, $x \Longrightarrow_q y$ and, moreover, all rules from R are applicable to x as well as no rule from Q is applicable to x.

A random-context grammar $G_{RC} = (G_s, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with permitting contexts of type X* if for all rules (q, R, Q) in P' we have $Q = \emptyset$, i.e., we only check for the applicability of the rules in R.

A random-context grammar $G_{RC} = (G_s, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with forbidden contexts of type X* if for all rules (q, R, Q) in P' we have $R = \emptyset$, i.e., we only check for the non-applicability of the rules in Q.

$L(G_{RC}) = \{v \in O_T \mid w \Longrightarrow_{G_{RC}}^{*} v\}$ is the language generated by G_{RC}. The families of languages generated by random context grammars, grammars with permitting contexts, and grammars with forbidden contexts of type X are denoted by $\mathcal{L}(X\text{-}RC)$, $\mathcal{L}(X\text{-}pC)$, and $\mathcal{L}(X\text{-}fC)$, respectively.

4.5 Ordered Grammars

An *ordered grammar G_O of type X* is a construct

$$(G_s, \prec, \Longrightarrow_{G_O})$$

where

- $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
- $\prec$ is a partial order relation on the rules in P;
- $\Longrightarrow_{G_O}$ is the derivation relation assigned to G_O such that for any $x, y \in O$, $x \Longrightarrow_{G_O} y$ if and only if for some rule $q \in P$ $x \Longrightarrow_q y$ and, moreover, no rule p from P with $q \prec p$ is applicable to x.

$L(G_O) = \{v \in O_T \mid w \Longrightarrow_{G_O}^{*} v\}$ is the language generated by G_O. The family of languages generated by ordered grammars of type X is denoted by $\mathcal{L}(X\text{-}O)$.

4.6 General Results

We now recall the main results and proofs established for the control mechanisms introduced so far in [21].

Theorem 1. *For any arbitrary type X,*

$$\mathcal{L}\left(X\text{-}MAT_{ac}\right) \subseteq \mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) \subseteq \mathcal{L}\left(X\text{-}GC_{ac}\right) \text{ and}$$
$$\mathcal{L}\left(X\text{-}MAT\right) \subseteq \mathcal{L}\left(X\text{-}GC^{allfinal}\right) \subseteq \mathcal{L}\left(X\text{-}GC\right).$$

Proof (see [21]*).* Let $G_M = (G_s, M, F, \Longrightarrow_{G_M})$ be a matrix grammar with $G_s = (O, O_T, w, P, \Longrightarrow_{G_s})$ being a grammar of type X; let

$$M = \{(p_{i,1}, \dots, p_{i,n_i}) \mid 1 \leq i \leq n\}$$

with $p_{i,j} \in P$, $1 \leq j \leq n_i$, $1 \leq i \leq n$. Then we construct the graph-controlled grammar $G_{GC} = (G_s, g, H_i, H_f, \Longrightarrow_{GC})$ with $g = (H, E, K)$, $H = \{(i,j) \mid 1 \leq j \leq n_i, 1 \leq i \leq n\}$, $K((i,j)) = \{p_{i,j}\}$, $1 \leq j \leq n_i$, $1 \leq i \leq n$, and

$$\begin{aligned} E = {} & \{((i,j), Y, (i,j+1)) \mid 1 \leq j < n_i, 1 \leq i \leq n\} \\ & \cup \{((i,j), N, (i,j+1)) \mid 1 \leq j < n_i, 1 \leq i \leq n, p_{i,j} \in F\} \\ & \cup \{((i,n_i), Y, (j,1)) \mid 1 \leq j \leq n, 1 \leq i \leq n\} \\ & \cup \{((i,n_i), N, (j,1)) \mid 1 \leq j \leq n, 1 \leq i \leq n, p_{i,j} \in F\} \end{aligned}$$

as well as $H_i = \{(i,1) \mid 1 \leq i \leq n\}$. As we have assumed that the sequential application of the rules of the chosen matrix may stop at any moment, we have to take $H_f = H$. By this construction it is guaranteed that G_{GC} simulates a derivation in G_M correctly by choosing a matrix to be simulated in a non-deterministic way and then applying the rules from this matrix in the desired sequence; the application of a rule $p_{i,j}$ may be skipped if and only if $p_{i,j} \in F$; hence, G_{GC} is without applicability checking if and only if G_M is without applicability checking, which observation completes the proof. □

The following theorem shows that forbidden contexts can simulate a partial order relation on the rules:

Theorem 2. *For any arbitrary type X, $\mathcal{L}(X\text{-}O) \subseteq \mathcal{L}(X\text{-}fC)$.*

Proof (see [21]*).* Let $G_s = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X. Consider the ordered grammar $G_O = (G_s, \prec, \Longrightarrow_{G_O})$ of type X and the corresponding grammar with forbidden contexts $G_{fC} = \left(G_s, P_{fC}, \Longrightarrow_{G_{fC}}\right)$ of type X where

$$\begin{aligned} P_{fC} &= \{(p, \emptyset, Q(p)) \mid p \in P\} \text{ with} \\ Q(p) &= \{q \mid q \in P, p \prec q\}. \end{aligned}$$

It is easy to see that $L(G_{fC}) = L(G_O)$, because a rule $p \in P$ can be applied in G_{fC} if and only if no rule from $Q(p)$ is applicable which is the same condition as for the applicability of p in G_O. □

Yet also the reverse inclusion holds, provided the type X allows for trap rules:

Theorem 3. *For any type X with trap rules, $\mathcal{L}(X\text{-}fC) \subseteq \mathcal{L}(X\text{-}O)$.*

Proof. Let $G_s = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X and consider the grammar with forbidden contexts $G_{fC} = \left(G_s, P_{fC}, \Longrightarrow_{G_{fC}}\right)$ of type X with

$$P_{fC} = \{(p, \emptyset, Q(p)) \mid p \in P\}.$$

We now extend the underlying grammar G_s by the trap rules p^- for all rules p in P, thus obtaining the grammar

$$G'_s = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'_s}\right)$$

where, according to the definition of grammars with trap rules,

- $P^{(-)} = \left\{p^{(-)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

As X is a type with trap rules, G'_s again is of type X.

We now define the ordered grammar

$$G_O = (G'_s, \prec, \Longrightarrow_{G_O})$$

which by definition again is of type X, with the partial order $\prec$ on the rules in $P \cup P^{(-)}$ as follows:

$$\text{for any } p \in P, p \prec q^- \text{ for all } q \in Q(p).$$

This guarantees that $L(G_{fC}) = L(G_O)$, as a rule $p \in P$ can be applied in G_O if and only if no rule from $Q(p)$ is applicable which is the same condition as for the applicability of p in G_{fC}. On the other hand, the application of a rule in $P^{(-)}$ can never lead to a terminal result. $\square$

The following result is an immediate consequence of the two previous theorems:

Corollary 1. *For any type X with trap rules, $\mathcal{L}(X\text{-}fC) = \mathcal{L}(X\text{-}O)$.*

As all the types defined for array grammars on Cayley grids in this paper are at least types with trap rules as argued above, see Lemma 3 and Remark 4, we obtain:

Corollary 2. *$\mathcal{L}(X\text{-}fC) = \mathcal{L}(X\text{-}O)$ for all types $C(G)\text{-}Y$ with $Y \in \{ARBA, \#\text{-}CFA, CFA, SCFA\}$.*

Matrix grammars (with applicability checking) can simulate random context grammars for any arbitrary type X with unit rules and trap rules:

Theorem 4. *For any arbitrary type X with unit rules and trap rules,*

$$\mathcal{L}\left(X\text{-}RC\right) \subseteq \mathcal{L}\left(X\text{-}MAT_{ac}\right).$$

Proof. Consider a *random-context grammar* $G_{RC} = (G_s, P_{RC}, \Longrightarrow_{G_{RC}})$ where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of a type X with unit rules and trap rules; then we define the *matrix grammar* with appearance checking $G_M = (G'_s, M, F, \Longrightarrow_M)$ of type X as follows: for each rule $(p, R, Q) \in P_{RC}$, $R = \{r_i \mid 1 \leq i \leq m\}$, $Q = \{q_j \mid 1 \leq j \leq n\}$, $m, n \geq 0$, we take the matrix $\left(r_1^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)$ into M. In that way we obtain $G'_s = \left(O, O_T, w, P', \Longrightarrow_{G'_s}\right)$ where

$$P' = P \cup \left\{ r^{(+)}, q^{(-)} \mid r \in R, q \in Q \text{ for some } (p, R, Q) \in P_{RC} \right\}$$

and $F = \left\{ q^{(-)} \mid q \in Q \text{ for some } (p, R, Q) \in P_{RC} \right\}$. As X is a type with unit rules and trap rules, all the elements of G_M are well defined. Obviously, for all $x, y \in O$ we have $x \Longrightarrow_{(p,R,Q)} y$ if and only if $x \Longrightarrow_{\left(r_i^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)} y$, which implies $L\left(G_M\right) = L\left(G_{RC}\right)$.

As a technical detail we mention that when the application of rules in the sequence of the matrix $\left(r_i^{(+)}, \ldots, r_m^{(+)}, q_1^{(-)}, \ldots, q_n^{(-)}, p\right)$ stops before having reached the end with applying p, either the underlying object has not yet changed as long as only the unit rules have been applied or else has already been trapped by the application of one of the trap rules, hence, no additional terminal results can arise from such situations. □

Omitting the forbidden rules and applicability checking, respectively, from the (proof of the) preceding theorem we immediately obtain the following result:

Corollary 3. *For any arbitrary type X with unit rules,*

$$\mathcal{L}\left(X\text{-}pC\right) \subseteq \mathcal{L}\left(X\text{-}MAT\right).$$

The main results elaborated for the relations between the specific regulating mechanisms in [21] and in this paper are depicted in the following diagram; most of these relations even hold for arbitrary types X.

Theorem 5. *The inclusions indicated by vectors as depicted in Fig. 1 hold, the additionally needed features of having unit and/or trap rules indicated by u and t, respectively, aside the vector:*

$$\begin{tikzcd}
 & & \mathcal{L}(X\text{-}GC_{ac}) & & & \\
 & & \mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) \arrow[u] & & & \\
\mathcal{L}(X\text{-}P_{ac}) \arrow[urr] & & & & \mathcal{L}(X\text{-}MAT_{ac}) \arrow[ull] & \\
 & & \mathcal{L}(X\text{-}GC^{allfinal}) \arrow[uu] & & & \mathcal{L}(X\text{-}RC) \arrow[ul, "{u,t}"'] \\
\mathcal{L}(X\text{-}P) \arrow[uu] \arrow[urr] & & \mathcal{L}(X\text{-}MAT) \arrow[u] \arrow[uurr] & & & \mathcal{L}(X\text{-}fC) \arrow[u] \arrow[d, shift left, "t"] \\
 & & & & \mathcal{L}(X\text{-}pC) \arrow[ull, "u"'] \arrow[uur] & \mathcal{L}(X\text{-}O) \arrow[u, shift left] \\
 & & \mathcal{L}(X) \arrow[uull] \arrow[uu] \arrow[urr] \arrow[urrr] & & &
\end{tikzcd}$$

Fig. 1. Hierarchy of control mechanisms for grammars of type X.

5 Grammars with Activation and Blocking of Rules

We now recall the definition of sequential grammars with activation and blocking of rules in a similar way as introduced in [3–5].

A *grammar with activation and blocking of rules* (an *AB-grammar* for short) of type X is a construct

$$G_{AB} = (G_s, L, f_L, A, B, L_0, \Longrightarrow_{G_{AB}})$$

where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, A, B are finite subsets of $L \times L \times \mathbb{N}$, and L_0 is a finite set of tuples of the form $\left(q, Q, \bar{Q}\right)$, $q \in L$, with the elements of $Q, \bar{Q}$ being of the form (l, t), where $l \in L$ and $t \in \mathbb{N}$, $t > 1$.

A derivation in G_{AB} starts with one element $\left(q, Q, \bar{Q}\right)$ from L_0 which means that the rule labeled by q has to be applied to the initial object w in the first step and for the following derivation steps the conditions given by Q as activations of rules and $\bar{Q}$ as blockings of rules have to be taken into account in addition to the activations and blockings coming along with the application of the rule labeled by q. The role of L_0 is to get a derivation started by activating some rule for the first step(s) although no rule has been applied so far, but probably also providing additional activations and blockings for further derivation steps.

A configuration of G_{AB} in general can be described by the object derived so far and the activations Q and blockings $\bar{Q}$ for the next steps. In that sense, the starting tuple $\left(q, Q, \bar{Q}\right)$ can be interpreted as $\left(\{(q,1)\} \cup Q, \bar{Q}\right)$, and we may also simply write $\left(Q', \bar{Q}\right)$ with $Q' = \{(q,1)\} \cup Q$. We mostly will assume Q and $\bar{Q}$ to be non-conflicting, i.e., $Q \cap \bar{Q} = \emptyset$; otherwise, we interpret $\left(Q', \bar{Q}\right)$ as $\left(Q' \setminus \bar{Q}, \bar{Q}\right)$.

Given a configuration $\left(u, Q, \bar{Q}\right)$, in one step we can derive $\left(v, R, \bar{R}\right)$, and we also write

$$\left(u, Q, \bar{Q}\right) \Longrightarrow_{G_{AB}} \left(v, R, \bar{R}\right),$$

if and only if

- $u \Longrightarrow_G v$ using the rule r such that $(q,1) \in Q$ and $(q,r) \in f_L$, i.e., we apply the rule labeled by q activated for this next derivation step to u; the new sets of activations and blockings are defined by

$$\begin{aligned} \bar{R} &= \left\{(x,i) \mid (x,i+1) \in \bar{Q},\ i > 0\right\} \cup \{(x,i) \mid (q,x,i) \in B\}, \\ R &= (\{(x,i) \mid (x,i+1) \in Q,\ i > 0\} \cup \{(x,i) \mid (q,x,i) \in A\}) \\ &\quad \setminus \left\{(x,i) \mid (x,i) \in \bar{R}\right\} \end{aligned}$$

  (observe that R and $\bar{R}$ are made non-conflicting by eliminating rule labels which are activated and blocked at the same time);
  or
- no rule r is activated to be applied in the next derivation step; in this case we take $v = u$ and continue with $\left(v, R, \bar{R}\right)$ constructed as before provided R is not empty, i.e., there are rules activated in some further derivation steps; otherwise the derivation stops.

The language generated by G_{AB} is defined by

$$L(G_{AB}) = \left\{v \in O_T \mid \left(w, Q, \bar{Q}\right) \Longrightarrow^*_{G_{AB}} \left(v, R, \bar{R}\right) \text{ for some } \left(Q, \bar{Q}\right) \in L_0\right\}.$$

The family of languages generated by AB-grammars of type X is denoted by $\mathcal{L}\,(X\text{-}AB)$. If the set B of blocking relations is empty, then the grammar is said to be a *grammar with activation of rules* (an *A-grammar* for short) of type X; the corresponding family of languages is denoted by $\mathcal{L}\,(X\text{-}A)$.

5.1 AB-Grammars and Graph-Controlled Grammars

Already in [21] graph-controlled grammars have been shown to be the most powerful control mechanism, and they can also simulate AB-grammars with the underlying grammar being of any arbitrary type X.

Theorem 6. *For any type X, $\mathcal{L}\,(X\text{-}AB) \subseteq \mathcal{L}\,(X\text{-}GC_{ac})$.*

Proof. Let $G_{AB} = (G, L, f_L, A, B, L_0, \Longrightarrow_{G_A})$ be an AB-grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of any type X. Then we construct a graph-controlled grammar

$$G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$$

with the same underlying grammar G. The simulation power is captured by the structure of the control graph $g = (H, E, K)$. The node labels in H, identifying the nodes of the graph in a one-to-one manner, are obtained from G_{AB} as all possible triples of the forms $\left(q, Q, \bar{Q}\right)$ or $\left(\bar{q}, Q, \bar{Q}\right)$ with $q \in L$ and the elements of $Q, \bar{Q}$ being of the form (r, t), $r \in L$ and $t \in \mathbb{N}$ such that t does not exceed the maximum time occurring in the relations in A and B, hence this in total is a bounded number. We also need a special node labeled $\emptyset$, where a computation in G_{GC} ends in any case when this node is reached.

All nodes can be chosen to be final, i.e., $H_f = H$. $H_i = L_0$ is the set of initial labels, i.e., we start with one of the initial conditions as in the AB-grammar.

The idea behind the node $\left(q, Q, \bar{Q}\right)$ is to describe the situation of a configuration derived in the AB-grammar where q is the label of the rule to be applied and $Q, \bar{Q}$ describe the activated and blocked rules for the further derivation steps in the AB-grammar. Hence, as already in the definition of an AB-grammar, we therefore assume $Q \cap \bar{Q} = \emptyset$.

Now let $g(l)$ denote the rule r assigned to label l, i.e., $(l, r) \in f_L$. Then, the set of rules assigned to $\left(q, Q, \bar{Q}\right)$ is taken to be $\{g(q)\}$. The set of rules assigned to $\emptyset$ is taken to be $\emptyset$.

As it will become clear later in the proof why, the nodes $\left(\bar{q}, Q, \bar{Q}\right)$ are assigned the set of rules $\{g(l) \mid (l, 1) \in Q,\ l \neq q\}$; we only take those nodes where this set is not empty.

When being in node $\left(q, Q, \bar{Q}\right)$, we have to distinguish between two possibilities:

– If $g(q)$ is applicable to the object derived so far, a Y-edge has to go to every node which describes a situation corresponding to what would have been the next configuration in the AB-grammar. We then compute

$$\begin{aligned}
\bar{R} &= \left\{(x, i) \mid (x, i+1) \in \bar{Q},\ i > 0\right\} \cup \{(x, i) \mid (q, x, i) \in B\}\,, \\
R &= (\{(x, i) \mid (x, i+1) \in Q,\ i > 0\} \cup \{(x, i) \mid (q, x, i) \in A\}) \\
&\quad \setminus\ \left\{(x, i) \mid (x, i) \in \bar{R}\right\}
\end{aligned}$$

(observe that R and $\bar{R}$ are made non-conflicting) as well as – if it exists – $t_0 := min\{t \mid (x, t) \in R\}$, i.e., the next time step when the derivation in the AB-grammar could continue. Hence, we take a Y-edge to every node $\left(p, P, \bar{P}\right)$ where $p \in \{x \mid (x, t_0) \in R\}$ and

$$\begin{aligned}
\bar{P} &= \left\{(x, i) \mid (x, i + t_0 - 1) \in \bar{R},\ i > 0\right\}, \\
P &= \{(x, i) \mid (x, i + t_0 - 1) \in R\}\,.
\end{aligned}$$

If $t_0 := min\{t \mid (x, t) \in R\}$ does not exist, this means that R is empty and we have to make a Y-edge to the node $\emptyset$.

– If $g(q)$ is not applicable to the object derived so far, we first have to check that none of the other rules activated at this step could have been applied, i.e., we check for the applicability of the rules in the set of rules

$$\bar{U} := \{g(l) \mid (l,1) \in Q,\ l \neq q\}$$

by going to the node $\left(\bar{q}, Q, \bar{Q}\right)$ with a N-edge; from there no Y-edge leaves, as this would indicate the unwanted case of the applicability of one of the rules in $\bar{U}$, but with a N-edge we continue the computation in any node $\left(p, P, \bar{P}\right)$ with p, P, $\bar{P}$ computed as above in the first case. We observe that in case $\bar{R}$ is empty, we can omit the path through the node $\left(\bar{q}, Q, \bar{Q}\right)$ and directly go to the nodes $\left(p, P, \bar{P}\right)$ which are obtained as follows: we first check whether $t_0 := min\{t \mid (x,t) \in Q,\ t > 1\}$ exists or not; if not, then the computation has to end with a N-edge to node $\emptyset$. Otherwise, a N-edge goes to every node $\left(p, P, \bar{P}\right)$ with $p \in \{x \mid (x,t_0) \in Q\}$ and

$$\begin{aligned}\bar{P} &= \left\{(x,i) \mid (x, i+t_0-1) \in \bar{Q},\ i > 0\right\},\\ P &= \left\{(x,i) \mid (x, i+t_0-1) \in Q\right\}.\end{aligned}$$

where the simulation may continue.

In this way, every computation in the AB-grammar can be simulated by the graph-controlled grammar with taking a correct path through the control graph and finally ending in node $\emptyset$; due to this fact, we could also choose the node $\emptyset$ to be the only final node, i.e., $H_f = \{\emptyset\}$. On the other hand, if we have made a wrong choice and wanted to apply a rule which is not applicable, although another rule activated at the same moment would have been applicable, we get stuck, but the derivation simulated in this way still is a valid one in the AB-grammar, although in most standard types X, which usually are strictly extended ones, such a derivation does not yield a terminal object. Having taken $H_f = \{\emptyset\}$, such paths would not even lead to successful computations in G_{GC}.

In any case, we conclude that the graph-controlled grammar G_{GC} generates the same language as the AB-grammar G_{AB}, which observation concludes the proof. □

We remark that in the construction of the graph-controlled grammar given in the preceding proof, all labels could be chosen to be final.

In the case of graph-controlled grammars with all labels being final, for any strictly extended type X with trap rules, we can show an exciting result exhibiting that the power of rule activation is really strong and that the additional power of blocking is not needed.

Theorem 7. *For any strictly extended type X with trap rules,*

$$\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) \subseteq \mathcal{L}\left(X\text{-}A\right).$$

Proof. Let

$$G_{GC} = (G_s, g, H_i, H_f, \Longrightarrow_{GC})$$

be a graph-controlled grammar where $G_s = (O, O_T, w, P, \Longrightarrow_G)$ is a strictly extended grammar of type X with trap rules; $g = (H, E, K)$, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and H_f is the set of final labels coinciding with the whole set H, i.e., $H_f = H$.

Then we construct an equivalent A-grammar

$$G_A = (G'_s, L, f_L, A, L_0, \Longrightarrow_{G_A})$$

as follows:

The underlying grammar G'_s is obtained from G_s by adding all trap rules, i.e., $G'_s = \left(O, O_T, w, P', \Longrightarrow_{G'_s}\right)$ with $P' = P \cup \{p^- \mid p \in P\}$. G'_s again is strictly extended and $w \notin O_T$, hence, also in G_A rules have to be applied before terminal objects are obtained. For any node in g labeled by l with the assigned set of rules P_l we assume it to be described by $P_l = \{p_{l,i} \mid 1 \leq i \leq n_l\}$. For all $q \in P$ we take the labels l_{q^-} into L as well as (l_{q^-}, q^-) into f_L.

We now sketch how the transitions from a node in g labeled by l with the assigned set of rules P_l can be simulated. The assumption that all nodes are final is crucial for this construction. Arriving in some node, one of the following situations is given:

1. the underlying object is terminal and therefore no rule from P is applicable any more, as X is a strictly extendable type; hence, we may stop in this node and extract the underlying object as a terminal result of the derivation, as all nodes are final;
2. the underlying object is not terminal, but no rule from $\bigcup_{i \in H} P_i$ is applicable any more; hence, even when continuing the derivation following a path through the control graph only using N-edges, the derivation cannot yield a terminal object any more; therefore, in such a case, we need not continue the derivation;
3. the underlying object is not terminal, no rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, is applicable, but there is still some node k reachable from node l following a path through the control graph only using N-edges that contains an applicable rule;
4. the underlying object is not terminal, but there is some rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, which is applicable.

For the simulation of these situations by the A-grammar, we therefore can restrict ourselves to the cases where when applying a rule we follow a path starting with a Y-edge and continuing with only N-edges until we reach a node containing a probably applicable rule; observe that such a path can only consist of the Y-edge, too.

In order to simulate a rule $p_{l,i}$ in P_l, $1 \leq i \leq n_l$, we take all activations into A which allow us to simulate the application of $p_{l,i}$ and to guess with which $p_{k,j}$ probably to continue afterwards. Hence, we consider all paths without loops $h_0 = l - h_1 - \cdots - h_n = k$ in the control graph g which start with a Y-edge and continue with only N-edges. For any such path we introduce labels

$((l,i),h_1,\ldots,(k,j))$ in L and $((l,i),h_1,\ldots,(k,j)):p_{l,i}$ in f_L; the set of all labels describing such paths from node l to any node k is denoted by $L_{l,i}$. Moreover, we use the following activations in A:

- $((l,i),h_1,\ldots,(k,j)),\{l_{q^-} \mid q \in \bigcup_{1\leq i\leq n-1} P_{h_i}\},1)$ is used to check in the next step that no rule along the path from node l to node k is applicable; observe that for $n=1$ the set $\bigcup_{1\leq i\leq n-1} P_{h_i}$ is empty and the whole activation can be omitted;
- in the second next step only the designated rule $p_{k,j}$ can be applied, i.e., we take $((l,i),h_1,\ldots,(k,j)),L_{k,j},2)$ into A; as with every label in $L_{k,j}$ the rule $p_{k,j}$ is assigned, the intended continuation is prepared.

How can a derivation in the A-grammar be started? As $w \notin O_T$, at least one rule must be applied to obtain a terminal object; hence, we check all possibilities that a rule in an initial node in H_i or along a path in g following only N-edges from such an initial node can be applied (observe that there are only finitely many paths without loops of that kind through the control graph); for each such rule $p_{l,i}$ in node l we take all labels from $L_{l,i}$ into L_0. As by construction $p_{l,i}$ is applicable it is guaranteed that any continuation of the computation will follow a Y-edge in g and thus the simulation in G_{A_C} will follow the simulation of an applicable rule as described above.

In total, the construction given above guarantees that the simulation of a computation in G_{GC} by a computation in G_A starts correctly and continues until no rule can be applied any more. As we have assumed all nodes in g to be final and X to be a strictly extended type, i.e., no rules can be applied to a terminal object any more, the only condition to get a result is to obtain a terminal object at the end of a computation. This observation completes our proof. □

As programmed grammars are just a special case of graph-controlled grammars with all labels being final, we immediately infer the following result:

Corollary 4. *For any strictly extended type X with trap rules,*

$$\mathcal{L}\left(X\text{-}P_{ac}\right) \subseteq \mathcal{L}\left(X\text{-}A\right).$$

Combining Theorems 6 and 7, we infer the following equality:

Corollary 5. *For any strictly extended type X with trap rules,*

$$\mathcal{L}\left(X\text{-}GC_{ac}^{allfinal}\right) = \mathcal{L}\left(X\text{-}A\right).$$

6 Results for Array Grammars on Cayley Grids

In many papers on control mechanisms for string grammars, the proof for showing that when using arbitrary productions any new control mechanism can be simulated is omitted, often simply citing the Church-Turing thesis, which usually is a legitimate claim as any formal proof would be tedious although bringing no new insights.

In case of array grammars on Cayley graphs the situation is more delicate: as long as the underlying group presentation is computable, one might still easily argue with the Church-Turing thesis as long as – for infinite groups – there is also an infinite path in the Cayley graph, which is obvious if there is a group element of infinite order – see the examples in Subsect. 2.2 and Example 10 as well as Remark 2. Yet even if there is no such element (for examples of such group presentations we refer to [23]), in a nondeterministic way, we can find lines of arbitrary length for the necessary computations, as by definition the out-degree of every node is bounded, hence, by König's infinity lemma such a path must exist; it is important to observe that these paths need not be computable in the general case. Therefore, in the general case of Cayley grids we need an algorithm that works directly with the power inherent to arbitrary array productions.

Theorem 8. *For any control mechanism Y,*

$$Y \in \{O, fC, pC, RC, P, P_{ac}, MAT, MAT_{ac}, GC, GC_{ac}, A, AB\},$$

$$\mathcal{L}\left(C\left(G\right)\text{-}ARBA\text{-}Y\right) \subseteq \mathcal{L}\left(C\left(G\right)\text{-}ARBA\right).$$

Proof (sketch). Given any array grammar with the control mechanism Y and with the underlying sequential array grammar being of type $ARBA$, we can construct an equivalent sequential array grammar of type $ARBA$ as follows:

The simulation of the application of an array production is obvious. The main difficulty which usually arises is to check that a specific array production is not applicable at any position in the array derived so far. In order to accomplish this task, from the beginning of the derivation we mark all positions ever visited by an array production with non-blank symbols which store the parent-children relation, i.e., as a child the information from where the underlying position has been affected is stored, and as a parent the information which children have been "born" is stored. Then, whenever we want to check that a specific array production is not applicable at any position in the Cayley grid, we send out a *checking signal* which propagates from the start node along the parent-children relations; whenever a node in the Cayley graph has no children any more and the array production under question is not applicable from that node, a *No*-signal is back-propagated along the children-parent relations, i.e., when all children have answered *No* and the rule under consideration is also not applicable at this current position in the Cayley graph, a signal *No* can be sent back from this parent node to its own parent. This algorithm ends in a successful way if the start node has received all *No*-answers from its children and the rule under consideration is also not applicable from the start node. Such information then can be moved along in the Cayley grid to a node where an array production is to be applied under the condition that specific other productions are not applicable in the whole current array.

This idea not only works for forbidding rules or for array grammars with a partial ordering on the rules, but also for simulating the passing from one node to another one in the control graph of a graph-controlled grammar along an N-edge as well as for checking that no activated rule is applicable.

At the end of the simulation, the intermediate non-terminal symbols have to be erased or to be changed into their corresponding terminal symbols including the blank symbol; we here also refer to the algorithm described in the proof of Lemma 2. □

Already an order relation on the rules is sufficient as a control mechanism to obtain $\mathcal{L}(C(G)\text{-}ARBA)$:

Theorem 9. $\mathcal{L}(C(G)\text{-}ARBA) \subseteq \mathcal{L}(C(G)\text{-}\#\text{-}CFA\text{-}O)$.

Proof (sketch). Let the array language be given by an array grammar

$$G_A = (C(G), N, T, \#, P, \{(v_0, S)\}, \Longrightarrow_{G_A})$$

on $C(G)$ in marked normal form, see Lemma 1. The underlying finitely presented group is $G = \langle B \mid R \rangle$, G' denotes the set of group elements.

We now sketch how to construct an equivalent array grammar

$$G_O = (G'_s, \prec, \Longrightarrow_{G_O}).$$

simulating the derivations in G_A.

The main idea is to first generate a workspace of non-terminal symbols $X_\#$ representing the blank symbol; such symbols $X_\#$ still occurring in the derived array at the end of a simulation of a derivation in G_A finally will be erased as to be described later in the proof. Moreover, at the very beginning we generate a control symbol at some place, chosen in a non-deterministic way, not interfering with the workspace needed for the simulations of the application of rules in G_A. In the general case, another construction is needed for that than the one exhibited in [16] for 1- and 2-dimensional array grammars. The main task then is to show how a marked array production $\bar{A}vB \to C\bar{D}$, where $A, B, C, D \in N'$, can be simulated by using a suitable order relation on the rules in G_O.

We first sketch how to obtain the control symbol and the workspace: Instead of starting with $\{(v_0, S)\}$ we use a new start symbol S' and the new initial array $\{(v_0, S')\}$. Using one of the rules $S'v\# \to S''H_A$ and then the rules $H_Av\# \to \#H_A$ for any $v \in B$, the initial control symbol H_A can move to any position (node) in the Cayley graph. At some moment we use the rule $H_A \to H_0$, which ends this travel and then allows the rule $S' \to \tilde{S}$ to be applied; this rule is "dominated" by the rules in $H^- \setminus \{H_0 \to F\}$, i.e., $S' \to \tilde{S} \prec p$ for all $p \in H^- \setminus \{H_0 \to F\}$, where $H^- = \{X \to F \mid X \in V_H\}$ and V_H denotes the set of all variants of the control variable H like H_A at the beginning.

Notation: In the following, the set of rules "dominating" a rule p will be written as $P(p \prec)$, i.e., $P(p \prec) = \{q \mid p \prec q\}$.

In general, the idea with the variants of the control variable H is to guide the application of another rule p by, instead of checking for the presence of the specific variant H_v of H, ensuring the absence of all other variants of H, using the rule relations $p \prec q$ for all $q \in \{X \to F \mid X \in V_H \setminus \{H_v\}\}$; hence, we also write $P(p \prec) = \{X \to F \mid X \in V_H \setminus \{H_v\}\}$.

The next task is to generate sufficient workspace of symbols $X_\#$ surrounded by a layer of symbols $\tilde{X}_\#$ on the border to the remaining environment of blank symbols:

We start with

$$\begin{aligned} p_0 &= \{(e, \tilde{S}\} \cup \{(v, \# \mid v \in B\} \to \{(e, \tilde{S}\} \cup \{(v, \tilde{X}_\# \mid v \in B\} \\ P(p_0 \prec) &= \{X \to F \mid X \in V_H \setminus \{H_0\}\}. \end{aligned}$$

Iteratively, now a new "layer" of symbols $X_\#$ is added by first generating symbols $\hat{X}_\#$ from the symbols $\tilde{X}_\#$, then renaming the symbols $\tilde{X}_\#$ to $X_\#$ and finally renaming the symbols $\hat{X}_\#$ to $\tilde{X}_\#$, which is accomplished by the following rules p and the corresponding "dominating" set of rules $P(p \prec)$:

1. $H_0 \to H_1$, $P(H_0 \to H_1 \prec) = \{\tilde{S} \to F\}$;
2. for all $v \in B$,

$$\begin{aligned} p_v^1 &= \{(e, \tilde{X}_\#), (v, \#)\} \to \{(e, \tilde{X}_\#), (v, \hat{X}_\#)\}, \\ P(p_v^1 \prec) &= \{X \to F \mid X \in V_H \setminus \{H_1\}\}, \end{aligned}$$

   $H_1 \to H_2$, $P(H_1 \to H_2 \prec) = \{p_v^{1^-} \mid v \in B\}$,
   where $p_v^{1^-}$ is the trap rule corresponding to the rule p_v^1, i.e.,

$$p_v^{1^-} = \{(e, \tilde{X}_\#), (v, \#)\} \to \{(e, F), (v, F)\};$$

3. for all $v \in B$,

$$\begin{aligned} p_v^2 &= \tilde{X}_\# \to X_\#, \\ P(p_v^2 \prec) &= \{X \to F \mid X \in V_H \setminus \{H_2\}\}, \end{aligned}$$

   $H_2 \to H_3$, $P(H_2 \to H_3 \prec) = \{p_v^{2^-} \mid v \in B\}$;
4. for all $v \in B$,

$$\begin{aligned} p_v^3 &= \hat{X}_\# \to \tilde{X}_\#, \\ P(p_v^3 \prec) &= \{X \to F \mid X \in V_H \setminus \{H_3\}\}, \end{aligned}$$

   $H_3 \to H_1$, $P(H_3 \to H_1 \prec) = \{p_v^{3^-} \mid v \in B\}$;
   the iteration can start again with 2.
5. In order to stop the iteration, instead of $H_3 \to H_1$ we use the rule
   $H_3 \to H$, $P(H_3 \to H \prec) = \{p_v^{3^-} \mid v \in B\}$.

For the simulation in G_O we assume the marked array productions in G_A to be labeled, i.e., we write $p : \bar{A}_p v_p B_p \to C_p \bar{D}_p$.

1. We start the simulation of the application of $p : \bar{A}_p v_p B_p \to C_p \bar{D}_p$ with indicating the intention to do that by the rule $H \to H_p^1$ for the control symbol;

2. we continue with marking exactly one symbol B_p as B'_p by

$$p_1 = B_p \rightarrow B'_p,$$
$$P(p_1 \prec) = \{X \rightarrow F \mid X \in (V_H \setminus \{H^1_p\}) \cup \{B'_p\}\},$$

   $H^1_p \rightarrow H^2_p$, $P(H^1_p \rightarrow H^2_p \prec) = P_F$,
   $P_F = \{\{(e, X), (v, \#)\} \rightarrow FF \mid X \in N \cup \{X_\#\}\}\}$,
   i.e., no blank symbol inside the workspace is allowed yet;
3. we now make a "#-hole" inside the workspace in such a way that the only non-terminal symbol having "access" to this blank position should be $\bar{A}_p$ by

$$p_2 = B'_p \rightarrow \#,$$
$$P(p_2 \prec) = \{X \rightarrow F \mid X \in (V_H \setminus \{H^2_p\})\},$$

   $H^2_p \rightarrow H^3_p$, $P(H^2_p \rightarrow H^3_p \prec) = P_F \setminus \{\bar{A}_p v_p \# \rightarrow FF\}$;
4. the "#-hole" made in the previous step now is filled correctly by

$$p_3 = \bar{A}_p v_p \# \rightarrow C_p \bar{D}_p,$$
$$P(p_3 \prec) = \{X \rightarrow F \mid X \in (V_H \setminus \{H^3_p\})\},$$

   $H^3_p \rightarrow H$, $P(H^3_p \rightarrow H \prec) = P_F$.

Using the sequence of rules as described above, we finally have simulated the application of the rule $p : \bar{A}_p v_p B_p \rightarrow C_p \bar{D}_p$ and reached the control symbol H again, which allows us to continue with simulating the next rule. At some moment we have to guess whether we can switch to the terminal procedure eliminating all non-terminal symbols:

1. We start with $H \rightarrow H_t$; the only symbols allowed in the current array in order to obtain a terminal array are terminal symbols, the workspace symbols $X_\#$ and $\tilde{X}_\#$ as well as (one) symbol $\bar{X}_\#$ indicating that in the simulated array grammar G_A in marked normal form the final rule $\bar{X}_\# \rightarrow \#$ could be applied; hence, we take

$$P(H \rightarrow H_t \prec) = \{X \rightarrow F \mid X \in (V \setminus (T \cup \{X_\#, \tilde{X}_\#, \bar{X}_\#\}))\};$$

2. for all $X \in \{X_\#, \tilde{X}_\#, \bar{X}_\#\}$, we take

$$p_X = \{X \rightarrow \#\},$$
$$P(p_X \prec) = \{X \rightarrow F \mid X \in (V_H \setminus \{H_t\})\};$$

3. if all other non-terminal symbols have been erased, finally the control symbol H_t can be erased, too, using the rule $H \rightarrow \#$, with

$$P(H \rightarrow \# \prec) = \{X \rightarrow F \mid X \in (V \setminus \{H_t\})\};$$

According to the construction of G_O and the explanation given above we conclude that $L(G_O) = L(G_A)$. □

Looking at the general results collected in Theorem 5 we see that a partial order on the rules is the weakest control mechanism in the inclusion line of the control mechanisms

$$O - fC - RC - MAT_{ac} - GC_{ac}^{allfinal} - GC_{ac}$$

and therefore we immediately infer the following result:

Corollary 6. *For any control mechanism Y,*

$Y \in \{O, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}\}$,

$$\mathcal{L}\left(C\left(G\right)\text{-}ARBA\right) \subseteq \mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}Y\right).$$

A similar result can be shown for programmed array grammars by proving the following equality:

Lemma 4.

$$\mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}PC_{ac}\right) = \mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}GC_{ac}^{allfinal}\right).$$

Proof. It is sufficient to show

$$\mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}PC_{ac}\right) \supseteq \mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}GC_{ac}^{allfinal}\right),$$

which can be proved using standard arguments already used for proving similar results for strings in [8] and for 1- and 2-dimensional arrays in [16]:

Given a graph-controlled array grammar with all nodes being final, we take a new non-terminal symbol S' as the new start symbol, i.e., instead of starting with $\{(v_0, S)\}$ we use new initial array $\{(v_0, S')\}$, and add one additional node to the control graph, to which we assign the new array production $S' \to S$; from this new node, Y-edges lead to every initial node in the original control graph.

As the new set of initial nodes we now can take every node in the new control graph, as the only array production applicable to the new initial array $\{(v_0, S')\}$ is the new array production $S' \to S$ assigned to the new node (which in fact could be the only initial node). Having all nodes being initial and final ones, the constructed new graph-controlled array grammar is a programmed one, too. □

Combining all the general results elaborated in this section, we obtain the main theorem of this paper for sequential array grammars on Cayley graphs with control mechanisms:

Theorem 10. *For any control mechanism Y,*

$Y \in \{O, fC, RC, P_{ac}, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}, A, AB\}$,

$$\mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}Y\right) = \mathcal{L}\left(C\left(G\right)\text{-}ARBA\right).$$

Proof. For $Y \in \left\{O, fC, RC, MAT_{ac}, GC_{ac}^{allfinal}, GC_{ac}\right\}$, the result follows from Corollary 6 and Theorem 8.

For $Y = P_{ac}$, we apply the result stated in Lemma 4, i.e.,

$$\mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}P_{ac}\right) = \mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}GC_{ac}^{allfinal}\right).$$

For $Y \in \{A, AB\}$, we can use the general result stated in Corollary 5, i.e.,

$$\mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}GC_{ac}^{allfinal}\right) = \mathcal{L}\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}A\right).$$

Moreover, even using activation *and blocking* of rules does not add additional computational power beyond $\mathcal{L}\left(C\left(G\right)\text{-}ARBA\right)$, as has been shown in Theorem 8. □

Based on Lemma 2, we obtain similar results for languages of k-connected arrays; the corresponding families of languages of k-connected arrays are marked with subscript $_k$, i.e., we write $\mathcal{L}_k$ instead of $\mathcal{L}$:

Theorem 11. *For any control mechanism Y,*
$Y \in \{O, fC, RC, P_{ac}, MAT_{ac}, GC_{ac}, A, AB\}$,

$$\mathcal{L}_k\left(C\left(G\right)\text{-}\#\text{-}CFA\text{-}Y\right) = \mathcal{L}_k\left(C\left(G\right)\text{-}ARBA\right).$$

7 Summary and Future Research

The notion of arrays as well as the concept of array grammars can be extended from the d-dimensional grid $\mathbb{Z}^d$ to arrays defined on Cayley graphs of finitely presented groups. We have investigated arrays defined on Cayley graphs of finitely presented groups and shown that the families of languages of such arrays generated by arbitrary array grammars coincide with those generated by #-context-free array grammars equipped with one out of various control mechanisms – control graphs, matrices, permitting and forbidden rules, or activation and blocking of rules. These results only need a few direct proof constructions, yet most of them directly follow from general results obtained for the relation between these control mechanisms for sequential grammars of arbitrary type.

Besides #-context-free array productions there are other types of rules to be considered in this framework of arrays defined on Cayley graphs of finitely presented groups together with these control mechanisms. For example, we are going to investigate whether similar results can be obtained when using insertion and deletion rules on arrays, for example, see [14,20]. Theorem 8 still remains valid when using array insertion and deletion rules together with the control mechanisms considered in this paper, yet showing that array insertion and deletion rules together with different control mechanisms reach the computational power of arbitrary array grammars needs careful proofs again.

There are also other control mechanisms to be considered, for example, using the structural power of tissue P systems, i.e., in a network of cells different rules are applicable in different cells, and the application of a rule sends the current array to another cell, for example, see [14,20].

Another interesting topic is to consider accepting array grammars with control mechanisms, an investigation already having been started two decades ago, see [10]: a given input array is accepted if it can be reduced to the initial array (in the accepting case better called the *goal* array). The type of an accepting array production $(W, \mathcal{A}_2, \mathcal{A}_1)$ is defined as the type of the corresponding generating array production $(W, \mathcal{A}_1, \mathcal{A}_2)$. In [10] specific results for d-dimensional accepting array grammars together with the control mechanisms of having an order relation on the rules or control graphs were established.

Acknowledgements. The current paper has been inspired by the discussions and the exchange of ideas with many of my friends and co-authors, yet I can only mention some of them being most influential for my work: I already became interested in the concepts of regulated rewriting during my studies even a decade before the book of Jürgen Dassow and Gheorghe Păun *Regulated Rewriting in Formal Language Theory* appeared in 1989, see [8], and collected the main results on control mechanisms for the string case in a comprehensive way.

Yet since the beginning, I have always tried to also apply these control mechanisms to grammars dealing with other objects as graphs, for instance, see [19], or d-dimensional arrays, for example, see [24], the first paper with Gheorghe, as well as [16]. Already more than twenty-five years ago, together with Jürgen I started the investigation of a general framework, yet it took a long time before the initial ideas condensed in *A General Framework for Regulated Rewriting Based on the Applicability of Rules*, see [21].

Array grammars have also been an interesting topic in picture generation and acceptance, and the first paper together with Jürgen and Gheorghe on *Cooperating Array Grammar Systems* was exploring some possibilities of how to use controlled array grammars for picture generation. Cooperating systems (of grammars) are another interesting way of controlling the application of rules by allowing different sets of rules to be applied according to a given strategy as, for example, to work as long as possible or a certain number of steps on the underlying object (an array in this case), also see [17].

Variants of array grammars were investigated to be applied for character recognition, for example, see [9], which paper started a very fruitful cooperation with Henning Fernau. The topic of character recognition was continued together with Markus Holzer, see [11–13]. Recently the cooperation with Henning (and his team in Trier) has been resumed with several papers dealing with control mechanisms for several variants of array grammars, for example, see [2,14,15].

With the concept of membrane systems introduced by Gheorghe Păun, see [27,28], the Handbook of Membrane Computing, a new model emerged, controlling the applicability of rules by using the hierarchical membrane structure where together with the application of a rule the current array can be sent to the outer or an inner membrane, which makes other sets of rules applicable, for example, see [18], the first paper on that topic for the sequential derivation mode with arrays as underlying objects, as well as [14,20], the contributions to MCU 2013.

From the beginning of this century, Marion Oswald, my colleague and friend working with me at the TU Wien for a very long period, had become my favorite co-author, especially in the area of P systems, but she also co-authored several papers on regulated rewriting, especially [21] on *A General Framework for Regulated Rewriting*

Based on the Applicability of Rules, and on variants of array grammars, especially the main papers introducing *Array Grammars and Automata on Cayley Grids*, see [22,23].

More recently, my friends from Moldova Artiom Alhazov and Sergiu Ivanov have become the motivating force to continue the research on new control mechanisms, not only in the area of P systems, but also with respect to new control mechanisms, especially also for array grammars, for instance, see [2,14,20]. The new concept of activation and blocking of rules in its final form has been developed during the *Brainstorming Week on Membrane Computing* this year in Sevilla mainly together with Sergiu, and different facets of this new control mechanism are presented in several papers, see [3–5], including this paper.

Both the list of references and the list of colleagues who contributed to my research on control mechanisms and/or on array grammars are far from being complete. I want to express my deep respect and my gratitude to all my colleagues and friends who helped me to develop new ideas and concepts, some of them presented in this paper.

References

1. Aizawa, K., Nakamura, A.: Grammars on the hexagonal array. In: Wang, P.S.P. (ed.) Array Grammars, Patterns and Recognizers. Series in Computer Science, vol. 18, pp. 144–152. World Scientific, River Edge (1989)
2. Alhazov, A., Fernau, H., Freund, R., Ivanov, S., Siromoney, R., Subramanian, K.G.: Contextual array grammars with matrix control, regular control languages, and tissue P systems control. Theor. Comput. Sci. **682**, 5–21 (2017). https://doi.org/10.1016/j.tcs.2017.03.012
3. Alhazov, A., Freund, R., Ivanov, S.: Introducing the concept of activation and blocking of rules in the general framework for regulated rewriting in sequential grammars. In: Proceedings of BWMC 2018 (2018)
4. Alhazov, A., Freund, R., Ivanov, S.: P systems with activation and blocking of rules. In: Verlan, S. (ed.) Proceedings of UCNC 2018. Lecture Notes in Computer Science. Springer (2018)
5. Alhazov, A., Freund, R., Ivanov, S.: Sequential grammars with activation and blocking of rules. In: Durand-Lose, J., Verlan, S. (eds.) MCU 2018. LNCS, vol. 10881, pp. 51–68. Springer, Cham (2018)
6. Cook, C.R., Wang, P.S.P.: A Chomsky hierarchy of isotonic array grammars and languages. Comput. Graph. Image Process. **8**, 144–152 (1978)
7. Csuhaj-Varjú, E., Mitrana, V.: Array grammars on Cayley grids, private communication
8. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. EATCS Monographs in Theoretical Computer Science, vol. 18. Springer, Heidelberg (1989)
9. Fernau, H., Freund, R.: Bounded parallelism in array grammars used for character recognition. In: Perner, P., Wang, P., Rosenfeld, A. (eds.) SSPR 1996. LNCS, vol. 1121, pp. 40–49. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61577-6_5
10. Fernau, H., Freund, R.: Accepting array grammars with control mechanisms. In: Păun, Gh., Salomaa, A. (eds.) New Trends in Formal Languages. LNCS, vol. 1218, pp. 95–118. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-62844-4_7
11. Fernau, H., Freund, R., Holzer, M.: Character recognition with k-head finite array automata. In: Amin, A., Dori, D., Pudil, P., Freeman, H. (eds.) SSPR /SPR 1998. LNCS, vol. 1451, pp. 282–291. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0033246

12. Fernau, H., Freund, R., Holzer, M.: Regulated array grammars of finite index. Part I: theoretical investigations. In: Păun, Gh., Salomaa, A. (eds.) Grammatical Models of Multi-Agent Systems. Topics in Computer Mathematics, vol. 8, pp. 157–181. Gordon and Breach Science Publishers, London (1999)
13. Fernau, H., Freund, R., Holzer, M.: Regulated array grammars of finite index. Part II: syntactic pattern recognition. In: Păun, Gh., Salomaa, A. (eds.) Grammatical Models of Multi-Agent Systems. Topics in Computer Mathematics, vol. 8, pp. 284–296. Gordon and Breach Science Publishers, London (1999)
14. Fernau, H., Freund, R., Ivanov, S., Schmid, M.L., Subramanian, K.G.: Array insertion and deletion P systems. In: Mauri, G., Dennunzio, A., Manzoni, L., Porreca, A.E. (eds.) UCNC 2013. LNCS, vol. 7956, pp. 67–78. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39074-6_8
15. Fernau, H., Freund, R., Siromoney, R., Subramanian, K.G.: Non-isometric contextual array grammars and the role of regular control and local selectors. Fundam. Inform. **155**(1–2), 209–232 (2017). https://doi.org/10.3233/FI-2017-1582
16. Freund, R.: Control mechanisms on #-context-free array grammars. In: Păun, Gh. (ed.) Mathematical Aspects of Natural and Formal Languages, pp. 97–137. World Scientific Publ., Singapore (1994)
17. Freund, R.: Array grammar systems. J. Autom. Lang. Comb. **5**(1), 13–30 (2000)
18. Freund, R.: P systems working in the sequential mode on arrays and strings. In: Calude, C.S., Calude, E., Dinneen, M.J. (eds.) DLT 2004. LNCS, vol. 3340, pp. 188–199. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30550-7_16
19. Freund, R., Haberstroh, B.: Attributed elementary programmed graph grammars. In: Schmidt, G., Berghammer, R. (eds.) WG 1991. LNCS, vol. 570, pp. 75–84. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55121-2_7
20. Freund, R., Ivanov, S., Oswald, M., Subramanian, K.G.: One-dimensional array grammars and P systems with array insertion and deletion rules. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, 9–11 September 2013. EPTCS, vol. 128, pp. 62–75 (2013). https://doi.org/10.4204/EPTCS.128
21. Freund, R., Kogler, M., Oswald, M.: A general framework for regulated rewriting based on the applicability of rules. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life. LNCS, vol. 6610, pp. 35–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20000-7_5
22. Freund, R., Oswald, M.: Array automata on Cayley grids. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013 Zürich, Switzerland, 9–11 September 2013. EPTCS, vol. 128, pp. 27–28 (2013). https://doi.org/10.4204/EPTCS.128
23. Freund, R., Oswald, M.: Array grammars and automata on Cayley grids. J. Autom. Lang. Comb. **19**(1–4), 67–80 (2014). https://doi.org/10.25596/jalc-2014-067
24. Freund, R., Păun, Gh.: One-dimensional matrix array grammars. Elektronische Informationsverarbeitung und Kybernetik **29**(6), 357–374 (1993)
25. Holt, D.F., Eick, B., O'Brien, E.A.: Handbook of Computational Group Theory. CRC Press, Hoboken (2005)
26. Krithivasan, K., Siromoney, R.: Array automata and operations on array languages. Int. J. Comput. Math. **4**(1), 3–30 (1974). https://doi.org/10.1080/00207167408803078
27. Păun, Gh.: Computing with Membranes. J. Comput. Syst. Sci. **61**, 108–143 (1998)
28. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press Inc., New York (2010)

29. Rosenfeld, A.: Picture Languages. Academic Press, Reading (1979)
30. Rosenfeld, A., Siromoney, R.: Picture languages - a survey. Lang. Des. **1**, 229–245 (1993)
31. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages: Volume 3 Beyond Words. Springer, Heidelberg (1997). https://doi.org/10.1007/978-3-642-59126-6
32. Salomaa, A.: Formal Languages. Academic Press, New York (1973)
33. Wang, P.S.P.: An application of array grammars to clustering analysis for syntactic patterns. Pattern Recogn. **17**, 441–451 (1984)

A Pleasant Stroll Through the Land of Distributed Machines, Computation, and Universality

Michel Raynal[1,2](✉) and Jiannong Cao[2]

[1] Institut Universitaire de France and Univ Rennes, IRISA CNRS INRIA, Rennes, France
raynal@irisa.fr

[2] Department of Computing, Polytechnic University, Hung Hom, Hong Kong

Abstract. Not only the world is distributed, but more and more applications are distributed. Hence, a fundamental question is the following one: What can be computed in a distributed system? The answer to this question depends on the environment in which evolves the considered distributed system, i.e., on the assumptions the system relies on. This environment is very often left implicit and nearly always not formulated in terms of precise underlying requirements. In the extreme case where the environment is such that there is no synchrony assumption and the computing entities may commit failures, some problems become impossible to solve. Given a distributed computing problem, it is consequently important to know the weakest assumptions (lower bounds) that give the limits beyond which the considered distributed problem cannot be solved. This paper is a short introduction to this kind of issues. It is made up of short sections, each addressing an important point of the theory of distributed computing. Its style is voluntarily informal.

Keywords: Agreement · Asynchronous system · Atomicity Concurrency · Consensus object · Consensus number · Crash failure Distributed complexity · Distributed computability Distributed computing · Environment · Fault-tolerance Impossibility result · Indistinguishability · Message adversary Message-passing system · Non-blocking · Obstruction-freedom Progress condition · Read/write system · Synchronous system · Task Universal construction · Wait-freedom

What is Distributed Computing?

An Informal Definition. "*Distributed computing* arises when one has to solve a problem in terms of distributed entities (usually called processors, nodes, processes, actors, agents, sensors, peers, etc.) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved. [...] Distributed computing can be characterized by the term *uncertainty*. This uncertainty is created by asynchrony, multiplicity of control flows, absence of shared memory and global time, failure, dynamicity, mobility, etc. Mastering one form or another of uncertainty is pervasive in all distributed computing problems. A main difficulty in designing distributed algorithms comes from the fact that each entity cooperating in the achievement of a common goal cannot instantaneously obtain the current state of the other entities, it can only know their past local states" which are not necessarily mutually consistent" (Extract from the preface of [44]).

On a more humorous side, there is Lamport's famous quotation defining a distributed system, namely,

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."

The previous sentences state that a distributed system/algorithm runs on a distributed machine (i.e., a machine composed of several computing devices), and the way these devices interact combined with their possible failures can have a strong impact on what can be computed.

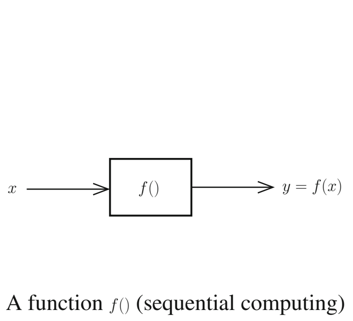
Birth Certificates. Since Lamport's seminal article "*Time, clocks, and the ordering of events in a distributed system*" [28], and other articles such as, to cite only two more among many other articles, namely, "*Impossibility of distributed consensus with one faulty process*" By Fischer et al. [16], and "*Wait-free synchronization*" by Herlihy [19][1], distributed computing is no longer a set of tricks or recipes, but a domain of *Informatics* with its own concepts, sane foundations, methods, and applications.

The Basic Unit of Distributed Computing

From a function ... While the basic unit of sequential computing is the notion of a *function*, the basic unit of distributed computing is the notion of a *task*, which was formalized in several papers (e.g., see [20,22,23] to cite a few). Intuitively, a task is a distributed function, which takes into account the fact that the inputs of the problem we want solve are distributed.

... to a task. A task $T()$ is made up of n processes $p_1, ..., p_n$ (computing entities), such that each process has its own input (in_i denoting the input of p_i) and must compute its own output (out_i denoting the output of p_i). Let $I = [in_1, \cdots, in_n]$ be an input vector (let us notice that a process knows only its local input, it does not know the whole input vector). Let $OUT = [out_1, \cdots, out_n]$ be an output vector (similarly, even if a process is required to cooperate with the

[1] These articles were awarded the "Dijkstra Prize" (in 2000, 2001, and 2003, respectively). As stated in https://en.wikipedia.org/wiki/Dijkstra_Prize, this prize "is given for outstanding papers on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade".



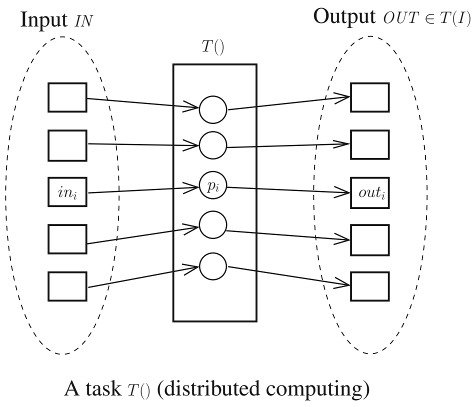


Fig. 1. Function vs (distributed) task

other processes, it has to compute its local output out_i, and not the whole output vector). A task T is defined by a set $\mathcal{I}$ of input vectors, a set $\mathcal{O}$ of output vectors, and a mapping T from $\mathcal{I}$ to $\mathcal{O}$, such that, given any input vector $in \in \mathcal{I}$, the output vector OUT (cooperatively computed by processes) is such that $OUT \in T(IN)$. The case $n = 1$ corresponds to sequential computing (see Fig. 1), and, in this case, a task boils down to a function.

Distributed Computing $\neq$ Parallel Computing

This section is inspired from [42], where fundamental differences between distributed computing and parallel computing are investigated in more details. These differences rest in the fact that a task is distributed by its very definition. This means that the processes, each with its own inputs, are geographically distributed and need to communicate to compute their outputs. The geographical *distribution of the computing entities is a not a design choice*, but is an input of the problem which gives its name to *distributed computing.*

Parallel Computing. In parallel computing, the inputs are, by essence, centralized. When considering the left part of Fig. 1, a function $f()$, and an input parameter x, parallel computing addresses concepts, methods, and strategies which allow to benefit from parallelism when one has to implement $f(x)$. The input x is given, and (if any) its initial scattering on distinct processors is not a priori imposed, but is a design choice aiming at obtaining efficient implementations of $f()$. The *essence* of parallel computing is to look for (or create) *independent parts of a computation* and, thanks to the independence of these computation units, manage them in an appropriate way in order to reduce as much a possible the computation time.

Distributed Computing. Differently, the *essence* of distributed computing is not on looking for efficiency but on *coordination in the presence of "adversaries"* (globally called *environment*) such as asynchrony, failures, locality, mobility, heterogeneity, limited bandwidth, restricted energy, etc. From the local point of view of each computing entity, these adversaries create uncertainty generating non-determinism, which (when possible) has to be solved by an appropriate algorithm.

A Synoptic View. In a few words, parallel computing is on the decomposition of a problem in independent parts (to benefit from the existence of many processors), while distributed computing is on the cooperation of pre-existing entities (in a given environment). Parallel computing is an *extension* of sequential computing in the sense that any problem addressed by a parallel algorithm can be solved –maybe very inefficiently– by a sequential algorithm. Differently, there are distributed computing problems that have neither a counterpart, nor a meaning, in sequential computing (e.g., the distributed termination detection problem in failure-free distributed systems [41]), and the consensus agreement abstraction in failure-prone distributed systems [44]).

Round-Based Synchronous Message-Passing Systems

Computing Model. From a structural point of view, this model can be represented by a connected graph where each vertex is a computing entity (process), and each edge corresponds to a bi-directional communication channel. The processes execute a sequence of rounds, and every round is made up of three phases:

- First phase: each process sends a message to all or a subset of its neighbors.
- Second phase: each process receives a message from each of its neighbors,
- Third phase: each process execute a local computation.

The fundamental synchrony property, which characterizes this model, is the fact that a message sent during a round r is received during the very same round r. This means that, if processes may crash, there is a bound on messages transfer delays. Actually the rounds are provided for free by the model, which ensures their progress [5,41]. An example of a synchronous run is depicted on the left of Fig. 2.

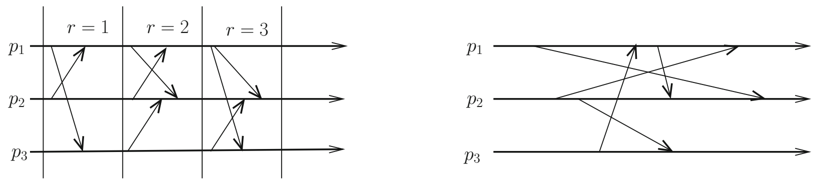


Fig. 2. Synchronous vs asynchronous computing model

The Notion of Locality. Let us consider a reliable synchronous message-passing system. If at the first round ($r = 1$) each process sends its local input to its neighbors, and then, at every round $r > 1$, sends them what it learned in the previous round $(r - 1)$, after a number of rounds equal to the diameter of the communication graph, any process knows the whole input vector.

The notion of a *local* algorithm was introduced in [29][2] and the associated model denoted $\mathcal{LOCAL}$ has been investigated in [37]. The locality notion has first been used to study complexity issues of distributed algorithms on graphs (such as vertex coloring, minimum independent set, minimum vertex cover, etc.), and has then addressed more general decision problems. In a local algorithm, a process is restricted to collect data from other processes which are at distance at most x (i.e., in at most x rounds), where x is smaller than the network diameter, or even a constant.

The main question is then: "Given a distributed graph problem, is it possible to solve it with a local algorithm?" This question is fundamental from a scalability point of view. As a simple example, the optimal vertex coloring problem is not local, while verifying if an arbitrary vertex coloring is such that no two neighbor vertices have the same color can be solved in one round.

The Notion of a Message Adversary. Let us now consider that there is a daemon (called *message adversary*), which, at every round, can suppress messages. Given a message adversary, characterized by its evil power (which is known by the processes), which problems can be solved in the corresponding failure-prone distributed synchronous system? These notions where introduced in [45,46] (with a different vocabulary).

As a simple example, it is shown in [27] that, even if processes do not known the graph diameter, it is possible for each process to compute any computable function on the input vector, if, at every round, the message adversary can suppress all messages on all channels except on the channels defining a spanning tree unknown to the processes. Moreover, the message adversary can define a different (unknown) spanning tree at at every round.

Communication in Failure-Prone Asynchronous Systems: Read/Write vs Message-Passing

On Failures. A distributed system may suffer different kind of failures. A process may crash (unexpected definitive halting), or commits a more severe failure by behaving in an arbitrary way (i.e., maliciously or not, the process executes a code different from the one specified by its algorithm). Such a process is called Byzantine process. Byzantine failures were introduced in [31][3].

If communication is by message-passing, channels can be unreliable by dropping messages (message alteration can be solved with error-detecting codes). Byzantine fault-tolerance and messages losses are not addressed here (the interested reader is referred to [44]).

[2] This paper received the Dijkstra Prize in 2013.
[3] This paper received the Dijkstra Prize in 2005.

In the following, n denote the number of processes, and t an upper bound on the number of processes that a model allows to crash.

Read/Write Model and Message-Passing Model. Let us consider a distributed system composed of n processes, denoted p_1, ..., p_n, where a *process adversary* may crash any subset of t of them. Each process is asynchronous, which means that it progresses at its own speed, which can vary arbitrarily, and is never known by the other processes. We use the following notations where CA stands for "Crash Asynchronous".

- $CARW_{n,t}[\emptyset]$ denotes the previous distributed model where the processes communicate by reading and writing atomic registers.
- $CAMP_{n,t}[\emptyset]$ denotes the previous distributed model where the processes communicate by sending and receiving messages through channels. It is assumed that any pair of processes is connected by a bi-directional channel. Example an execution in this model is described on the right side of Fig. 2.

Let us notice that, while the notion of a round is given for free in the synchronous system model, it is not in an asynchronous system model. However it is possible for the processes to build a sequence of rounds. Considering the model $CAMP_{n,t}[\emptyset]$, and assuming each message carries the round number in which it was sent, during a round r, a process can wait round r messages from at most $(n-t)$ processes (including itself), before locally proceeding to the next round. It follows that the processes execute a sequence of *asynchronous rounds* (which means that two processes are not necessarily at the same round at the same time). Moreover, a process discards the round r messages arriving after it stops waiting for round r messages.

Read/Write on Top of Message-Passing. A simple, but important, question concerns the equivalence of these distributed computing models. Simulate $CAMP_{n,t}[\emptyset]$ on top of $CARW_{n,t}[\emptyset]$ is easy: A channel can be implemented with two queues of atomic read/write registers, one for each direction of the channel.

As far the other direction is concerned, it was proved in [2][4] the following

> Theorem: It is impossible to build an atomic read/write register on top of $CAMP_{n,t}[\emptyset]$ if $t \geq n/2$.

The intuition that explains the impossibility to build an atomic read/write register in $CAMP_{n,t}[\emptyset]$ relies on the fact that, when $t \geq n/2$, half or more processes may crash in a run. More precisely, as any number of processes may crash, it is possible to construct executions in which the system "partitions" in two set of processes such that, while there is no crash, the messages between the two partitions take – in both directions– an arbitrarily long time, making the processes in each partition believe that the processes in the other partition have crashed. The system can then progress as two disconnected subsystems.

[4] This paper received the Dijkstra Prize in 2011.

Let $CAMP_{n,t}[t < n/2]$ denote $CAMP_{n,t}[\emptyset]$ restricted by the environment assumption $t < n/2$. Several algorithms (each with its own features) building atomic read/write registers in $CAMP_{n,t}[t < n/2]$ have been proposed (e.g., [2,35] to cite a few; see [44] for a pedagogical presentation).

A Fundamental Notion: Indistinguishability

A fundamental notion associated with impossibility results in distributed computing is the notion of *indistinguishability* [4,39]. This is related to the fact distributed computing has to cope with the *uncertainty* created by the adversaries managing the environment, namely, here the net effect of asynchrony and failures.

Assuming an algorithm that solves a given problem, the indistinguishability-based strategy to prove its impossibility (or a lower bound) consists in playing with asynchrony and failures to produce two executions $E1$ and $E2$ such that there is a process p_i that should output a result $out1$ in $E1$, and a result $out2 \neq out1$ in $E2$, but p_i cannot distinguish $E1$ and $E2$. In both $E1$ and $E2$, it executed the same sequence of steps, and produced the same sequence of local states. The fact that both $E1$ and $E2$ can be produced by the assumed algorithm entails a contradiction from which follows the impossibility (or a lower bound).

Concurrent Objects and Progress Conditions

This section is inspired from [43].

Concurrent Objects. A *concurrent object* is an object that can be accessed (possibly simultaneously) by several processes. From both practical and theoretical point of views, a fundamental problem of concurrent distributed programming consists in implementing high level concurrent objects, where "high level" means that the object provides the processes with an abstraction level higher than the atomic hardware-provided instructions. While this is well-known and well-mastered since a long time in the context of failure-free systems [7], it is far from being trivial in failure-prone distributed systems (e.g., see textbooks such as [40,49] for the read/write model, and such as [44] for the message-passing model).

Progress Conditions. Deadlock-freedom and starvation-freedom are the two progress conditions encountered in failure-free asynchronous systems. As their implementation is based on lock mechanisms, they are not suited to asynchronous crash-prone systems. This is due to the fact that, as it is impossible to distinguish a crashed process from a slow process, a process that acquires a lock and crashes before releasing it can entail the blocking of the entire system.

Hence, new progress conditions for concurrent objects suited to crash-prone asynchronous systems have been proposed. Given an object, we have the following.

– The strongest progress condition is *wait-freedom* (WF) [19]. It states that, any operation (on the object that is built), issued by a process that does not crash, terminates. This means that it terminates whatever the behavior of the other processes. This can be seen as the equivalent of the starvation-freedom progress condition encountered in failure-free systems.
– The *non-blocking* progress condition (NB) states that at least one of the processes, that do not crash, returns from all its object operations [25]. This progress condition is also called *lock-freedom*. It can be seen as the equivalent of deadlock-freedom in failure-free systems.
– The *obstruction-freedom* progress condition (OB) states that a process that does not crash will be able to terminate its operation if all the other processes hold still long enough [21]. This is the weakest progress condition.

While *wait-freedom* and *non-blocking* are independent of the concurrency and failure pattern, *obstruction-freedom* is dependent from it. Asymmetric progress conditions have been introduced in [26]. The computational structure of progress conditions is investigated in [50].

Notion of a Universal Construction

Universal Construction in Distributed Computing. The notion of a *universal construction* for concurrent objects built on top of message-passing distributed systems was first introduced by Lamport [28][5] under the notion of *state machine replication*. (See also [8,44,47,48] for surveys on fault-tolerant state machine replication.)

It was then addressed by Herlihy [19] in the context of concurrent objects built on top of a read/write asynchronous crash-prone distributed system ($CARW_{n,t}[\emptyset]$). The concurrent objects that are considered are the ones (a) defined by a sequential specification and (b) with total operations (i.e., any object operation returns a result; as an example, pop() on an empty stack returns the default value $\perp$).

Let PC be a progress condition. A *PC-compliant universal construction* is a algorithm that, given the sequential specification of an object O (or a sequential implementation of it), provides a concurrent implementation of O satisfying the progress condition PC (Fig. 3).

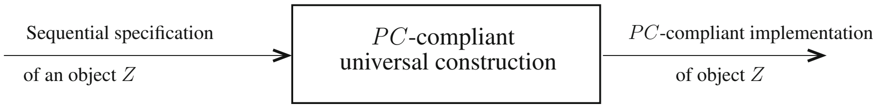


Fig. 3. PC-compliant universal construction

Universal Concurrent Objects. Let us consider in the following the strongest progress condition, namely wait-freedom. It is shown in [19][6] that a WF-

[5] This paper received the Dijkstra Prize in 2000
[6] This paper received the Dijkstra Prize in 2003

compliant universal construction can be built from atomic read registers AND consensus objects (Fig. 4). This is why consensus is called a *universal* object.

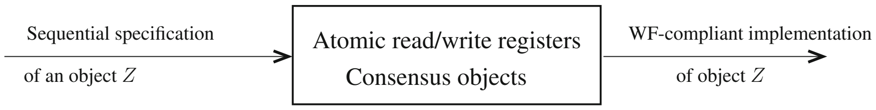


Fig. 4. WF-compliant universal construction

Consensus objects are used, in a WF-compliant universal construction, to ensure that, despite asynchrony and process crashes, the processes agree on the sequential order in which operations on the object that is built are applied to its internal representation. (This internal representation can be stored in the shared memory or in the local memory of each process in the model $CARW_{n,t}[\emptyset]$, or in the local memories in of the processes in the model $CAMP_{n,t}[\emptyset]$.) The important point (as explained below) is that read/write registers alone are not sufficient to design a WF-compliant universal construction. A tutorial-oriented survey on such universal constructions can be found in [43].

Consensus Object. A consensus object is a one-shot concurrent object that has a single operation, denoted propose() (one-shot means that a process invokes propose() at most once). When a process p_i invokes propose(v), we say "p_i proposes value v". When its invocation terminates, p_i obtains a value w, and we say "p_i decides w". The consensus object is defined by the following properties.

– Validity. If a process decides a value, this value was proposed by a process.
– Agreement. No two processes decide different values.
– Termination. The invocation of propose() by a process that does not crash terminates.

Validity and Agreement are safety properties. Validity relates the outputs to the inputs, and Agreement gives its meaning to the output. Termination is a liveness property, stating that the progress condition of a consensus object is wait-freedom.

Consensus as a Task. The task notion was presented in Fig. 1. For ease of exposition, let us consider binary consensus, where only the values 0 and 1 can be proposed by the processes. We show here that consensus is a task. To this end, we have to define the set $\mathcal{I}$ of input vectors, the set $\mathcal{O}$ of the possible output vectors, and the mapping $T()$ from $\mathcal{I}$ to $\mathcal{O}$. We have the following.

– $\mathcal{I} = \{$ all vectors of 0 and 1 $\}$.
– $\mathcal{O} = \{[0, \ldots, 0], [1, \ldots, 1]\}$.
– Let $X_0 = [0, \ldots, 0]$, and $X_1 = [1, \ldots, 1]$.

- $T(X_0) = X_0$, and $T(X_1) = X_1$.
- $T(\text{any vector} \neq X_0, X_1) \in \mathcal{O}$.

Relations between concurrent objects and tasks are investigated in [10].

A Fundamental Impossibility Result: FLP85

The Impossibility Result. This is (one of) the most important impossibility result(s) of distributed computing. It was first established by Fischer, Lynch, and Paterson in 1985 (hence its name) in the context of asynchronous message-passing systems[7] (system model $CAMP_{n,t}[\emptyset]$). It was then extended to the case of asynchronous read/write systems [30] (system model $CARW_{n,t}[\emptyset]$).

> Theorem: There is no deterministic algorithm implementing a consensus object in the system $CAMP_{n,t}[\emptyset]$ or $CARW_{n,t}[\emptyset]$.

It is important to notice that this impossibility holds even for $t = 1$, is not related to a specific communication medium, and is independent of the domain of the values that can be proposed. More, as expressed below, neither it is related to the local computability power of the computing entities.

The Nature of the Impossibility Result. The following citation (from [22]) captures what makes different the nature of sequential computability (as described for example in [18]), and the nature of distributed computability.

> "In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine.
>
> In distributed systems, where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants."

Universal Objects

Read/Write Registers. These objects constitute the cells of the tape of a Turing machine, and are consequently universal in sequential computing, and more generally, in failure-free parallel computing (as any problem that can be solved "in parallel" can be solved sequentially). As a simple example, mutual exclusion can solved in $CARW_{n,t}[t = 0]$ (read/write-based mutex algorithms are described and analyzed in several textbooks such as [40,49]).

[7] [16] This paper received the Dijkstra Prize in 2002

Consensus Objects. As we have just seen, read/write registers are no longer universal in $CARW_{n,t}[\emptyset]$. Differently, in the presence of asynchrony and crash failures, consensus objects are universal. This means that, while not all concurrent object defined by a sequential specification can be implemented on top of $CARW_{n,t}[\emptyset]$ and $CAMP_{n,t}[t < n/2]$, they can in the system models $CARW_{n,t}[\text{CONS}]$ and $CAMP_{n,t}[t < n/2, \text{CONS}]$, where CONS states that the corresponding model is enriched with consensus objects.

Consensus Numbers in Enriched Read/Write Systems

Hardware-Provided Synchronization Operations. A lot of machines (e.g., multiprocessors, multicore) provide processes with specialized hardware operations, whose aim is to help programmers manage synchronization issues. Such operations are encountered under the names Test&Set(), Compare&Swap(), LL/SC, Fetch&Add(), etc. (For more information on these operations, the interested reader can consult textbooks such as [24,40,49]).

XX being any of the previous synchronization-oriented operation, let $CARW_{n,t}[\mathsf{XX}]$ denote the system model $CARW_{n,t}[\emptyset]$ enriched with the operation XX. An important question is then the following one: from a computability point of view, is $CARW_{n,t}[\mathsf{XX}]$ stronger than $CARW_{n,t}[\emptyset]$? Given XX1 and XX2 $\neq$ XX1, have $CARW_{n,t}[\mathsf{XX1}]$ and $CARW_{n,t}[\mathsf{XX2}]$ the same computability power, or is one of them stronger than the other?

Consensus Numbers and the Consensus Hierarchy. The previous question was posed and answered by M. Herlihy, who introduced the notion of a consensus number of an object [19].

Let us consider an object type T. The *consensus number* of the objects of type T (denoted $CN(T)$), is the greatest integer x, such that it is possible to implement consensus in a system of x processes with any number of read/write registers and objects of type T. If there is no largest x, $CN(T) = +\infty$.

From a universal construction point of view, the objects of type $T1$ are stronger than the objects of type $T2$, if $CN(T1) > CN(T2)$. This defines a hierarchy on the “universality power” of objects in the presence of asynchrony an crash failures. More precisely we have the following, where memory locations are considered as objects accessed by one of the previous synchronization operations.

– The consensus number of read/write registers is 1. It follows that all the objects that can be implemented in $CARW_{n,t}[\emptyset]$ have consensus number 1. This the case of the (non-trivial) snapshot object introduced in [1], and the renaming object [3,9].
– The consensus number of Test&Set(), Fetch&Add(), a stack, or a queue is 2.
– Let a k-window read/write register be an object that stores only the k last written values, and whose read operation returns this sequence of at most k values. The consensus number of this object is exactly k [33].

– The consensus number of Compare&Swap(), LL/SC, and a few others, is $+\infty$.

This establishes a hierarchy on the computability power of each synchronization operation –taken individually– provided by some machines, in the presence of asynchrony and process crash-failures. This is important as soon as one wants to be able to cope with failures in multicore machines.

Circumventing Consensus Impossibility in Asynchronous Systems

The basic communication object in $CAMP_{n,t}[\emptyset]$ is the uni-directional channel, where only one process send messages, and only one other process receives message. This object has consensus number 1. Moreover, it is not possible to enrich a message-passing system the model with an additional operation such as the previous hardware-provided synchronization operations. This means that implement consensus in $CAMP_{n,t}[t < n/2]$ requires additional assumptions of a different nature. We examine in the following four approaches that have been proposed to solve this issue.

Synchrony Assumptions. As consensus can be solved in a synchronous message-passing system, a first approach consists in enriching the system model $CAMP_{n,t}[\emptyset]$ with a synchrony assumption as proposed in [13,14][8]. The most common of these assumptions is called *eventual synchrony*. It assumes that there is an unknown, but finite, time τ after which the system behaves as a synchronous system.

Failure Detectors. Failure detectors have been introduced in [11,12][9]. A failure detector is an oracle that provides each process with information on failures. Several types of failure detector have been proposed, each with precisely defined quality of service, formulated as a set of properties (the important point is that, a failure detector is not defined from specific features on a distributed machine).

A failure detector can be unreliable in the sense that it can give erroneous information on failures. Their interest lies in the fact that, given an object O impossible to implement in $CARW_{n,t}[\emptyset]$, or $CAMP_{n,t}[\emptyset]$, they allow to state the *weakest information on failures* that, when added to $CARW_{n,t}[\emptyset]$, or $CAMP_{n,t}[\emptyset]$, allow to implement the O.

As an example the weakest failure detector that allow to implement consensus in $CARW_{n,t}[\emptyset]$, or $CAMP_{n,t}[t < n/2]$, is denoted Ω (hence, from a notation point of view, consensus can be implemented in $CARW_{n,t}[\Omega]$ and $CAMP_{n,t}[t < n/2, \Omega]$, Ω provides each process p_i with a read-only local variable $leader_i$ satisfying the following properties:

– Validity. For any process p_i and at any time, $leader_i$ contains a process identity (which can change with time).

[8] The second of these papers received the Dijkstra Prize in 2007.
[9] These papers received the Dijkstra Prize in 2010.

– Eventual leadership. There a finite, but unknown, time τ after which the local variables $leader_i$ of all the processes contain forever the same process identity, and this identity is the one of a process that does not crash.

Said differently, there is a finite anarchy period during the variables $leader_i$ can contain arbitrary values (e.g., each process is its own leader), after which these local read-only variables stabilize to the same identity of a non-crashed process.

Distributed algorithms implementing Ω in message-passing systems have been proposed (a chapter of [44] is devoted to such algorithms). Each of these algorithms considers that the underlying system satisfies some specific behavioral assumptions. For the model $CARW_{n,t}[\emptyset]$, an algorithm implementing Ω from very weak synchrony additional assumptions, is described in [15].

Conditions. The *condition-based* approach to solve asynchronous consensus consists in identifying subsets of input vectors, such that if the actual input vector belongs to the considered subset, consensus can be solved. Such a subset is called a *t-legal condition.* If adding another vector to the condition makes it not t-legal, the condition is *maximal.*

Let $\mathsf{dist}(I1, I2)$ be the Hamming distance between the input vectors $I1$ and $I2$ (number of entries in which they differ), and $\mathsf{equal}(a, I)$ be the number of entries of I whose value is a. It is shown in [34] that a subset of input vectors C is a t-legal condition if there is a function $h() : C \mapsto \mathcal{V}$ (where $\mathcal{V}$ is the set values that can be proposed, in our case $\mathcal{V} = \{0, 1\}$) such that

– $\forall I \in C :\ \mathsf{equal}(h(I), I) > t$,
– $\forall I1, I2 \in C :\ \big(h(I1) \neq h(I2)\big) \Rightarrow \big(\mathsf{dist}(I1, I2) > t\big)$.

The intuition that underlies the condition-based approach is the following. Given a condition C, each of its input vectors allows a proposed value to be selected as the decided value. This value is extracted from the input vector with the function $h()$, i.e., $h(I)$ is the value decided from I. When looking at the definition of t-legality, the first item states that, to be decided, a value must be “present enough” in the input vector, while the second item states that input vectors from which different values are decided must be “enough far apart” to prevent ambiguity. A relation linking the consensus condition-based approach and error-correcting code was established in [17].

An example of a condition is the one that favors the most present value in the input vector. Given an input vector I, let $\mathsf{first}(I)$ be its most frequent value, $\mathsf{second}(I)$ its second most frequent value, and let $\mathsf{equal}(\mathsf{second}(I), I) = 0$ when I contains a single value. This condition C is defined as follows.

$$C = \{I \text{ such that } \big(\mathsf{equal}(\mathsf{first}(I), I) - \mathsf{equal}(\mathsf{second}(I), I)\big) > t\}.$$

Randomization. The use of random oracles to build a consensus object was introduced in the early eighties in [6, 38][10]. Considering binary consensus, each

[10] These papers received the Dijkstra Prize in 2015.

process can invoke a function denoted random() which returns each of 0 and 1 with probability 1/2.

The termination property of consensus has to be slightly modified to take into account randomization. The corresponding algorithms are round-based, and the termination property becomes: the probability that a process that does not crash decides by round r tends to 1 when r tends to $+\infty$. So, from a randomized algorithm point of view, these consensus algorithms are Las Vegas algorithms.

A randomized binary consensus algorithm has been recently proposed for Byzantine message-parsing systems in [32], which is optimal in the number of messages exchanged at every round ($O(n^2)$), the expected number of rounds ($O(n^2)$), and t-resilience ($t < n/3$). This algorithm also tolerates an adversary which can read the content of the messages, and controls their delivery order according to their values. (In a Byzantine system, a process not only may crash, but may also behave arbitrarily). A reduction of multivalued consensus to binary consensus, suited to Byzantine message-passing systems has been recently proposed in [36], which requires $t < n/3$, a constant number of communication steps (each requiring n^2 messages). Let us notice that, due the fact that its non-determinism cannot be anticipated, randomization seems an appropriate approach to cope with Byzantine processes.

Complexity vs Computability

Computability and complexity are the two lenses that allows us to understand and master computing. The following table presents the main issues encountered in distributed computing, when considering these lenses. (The interested reader will find more developments in distributed computing theory-oriented textbooks such as [4,20,37,40,49]).

	Synchronous	Asynchronous
Failure-free	Complexity	Complexity
Failure-prone	Complexity	Computability

A special thanks to J. Durand-Lose for his careful reading of a draft of this article.

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM **40**(4), 873–890 (1993)
2. Attiya, H., Bar-Noy, A., Dolev, D.: Sharing memory robustly in message-passing systems. J. ACM **42**(1), 121–132 (1995)
3. Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D., Reischuk, R.: Renaming in an asynchronous environment. J. ACM **37**(3), 524–548 (1990)

4. Attiya, H., Ellen, F.: Impossibility Results in Distributed Computing. Synthesis Lectures on Distributed Computing Theory, 146 pages. Morgan & Claypool, San Rafael (2014)
5. Awerbuch, B.: Complexity of network synchronization. J. ACM **4**, 804–823 (1985)
6. Ben-Or, M.: Another advantage of free choice: completely asynchronous agreement protocols. In: Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing, pp. 27–30. ACM Press (1983)
7. Brinch Hansen, P.: The Origin of Concurrent Programming, 534 pages. Springer, New York (2002). https://doi.org/10.1007/978-1-4757-3472-0
8. Cachin, C.: State machine replication with Byzantine faults. In: Charron-Bost, B., Pedone, F., Schiper, A. (eds.) Replication. LNCS, vol. 5959, pp. 169–184. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11294-2_9
9. Castañeda, A., Rajsbaum, S., Raynal, M.: The renaming problem in shared memory systems: an introduction. Elsevier Comput. Sci. Rev. **5**, 229–251 (2011)
10. Castañeda, A., Rajsbaum, S., Raynal, M.: Specifying concurrent problems: beyond linearizability and up to tasks. In: Moses, Y. (ed.) DISC 2015. LNCS, vol. 9363, pp. 420–435. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48653-5_28
11. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
12. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM **43**(2), 225–267 (1996)
13. Dolev, D., Dwork, C., Stockmeyer, L.: On the minimal synchronism needed for distributed consensus. J. ACM **34**(1), 77–97 (1987)
14. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988)
15. Fernández, A., Jiménez, E., Raynal, M., Trédan, G.: A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared-memory systems. Algorithmica **56**(4), 550–576 (2010)
16. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
17. Friedman, R., Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Asynchronous agreement and its relation with error-correcting codes. IEEE Trans. Comput. **56**(7), 865–875 (2007)
18. Harel D., Feldman, Y.: Algorithmics: The Spirit of Computing, 572 pages. Springer, Heidelberg (2012)
19. Herlihy, M.P.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991)
20. Herlihy, M.P., Kozlov, D., Rajsbaum, S.: Distributed Computing Through Combinatorial Topology, 336 pages. Morgan Kaufmann/Elsevier (2014). ISBN 9780124045781
21. Herlihy, M.P., Luchangco, V., Moir, M.: Obstruction-free synchronization: double-ended queues as an example. In: Proceedings of the 23th International IEEE Conference on Distributed Computing Systems (ICDCS 2003), pp. 522–529. IEEE Press (2003)
22. Herlihy, M.P., Rajsbaum, S., Raynal, M.: Power and limits of distributed computing shared memory models. Theoret. Comput. Sci. **509**, 3–24 (2013)
23. Herlihy, M., Shavit, N.: The topological structure of asynchronous computability. J. ACM **46**(6), 858–923 (1999)
24. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, 508 pages. Morgan Kaufmann (2008). ISBN 978-0-12-370591-4

25. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
26. Imbs, D., Raynal, M., Taubenfeld, G.: On asymmetric progress conditions. In: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing (PODC 2010), pp. 55–64. ACM Press (2010)
27. Kuhn, F., Lynch, N.A., Oshman, R.: Distributed computation in dynamic networks. In: Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC 2010), pp. 513–522. ACM Press (2010)
28. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
29. Linial, N.: Locality in distributed graph algorithms. SIAM J. Comput. **21**(1), 193–201 (1992)
30. Loui, M., Abu-Amara, H.: Memory requirements for agreement among unreliable asynchronous processes. Adv. Comput. Res. **4**, 163–183 (1987)
31. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**, 228–234 (1980)
32. Mostéfaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous binary Byzantine consensus with $t<n/3$, $O(n^2)$ messages, and $O(1)$ expected time. J. ACM **62**(4), Article 31, 21 pages (2015)
33. Mostéfaoui, A., Perrin, M., Raynal, M.: A simple object that spans the whole Consensus hierarchy. Technical report, IRISA, Université de Rennes 1 (France), 6 pages (2017)
34. Mostéfaoui, A., Rajsbaum, S., Raynal, M.: Conditions on input vectors for consensus solvability in asynchronous distributed systems. J. ACM **50**(6), 922–954 (2003)
35. Mostéfaoui, A., Raynal, M.: Two-bit messages are sufficient to implement atomic read/write registers in crash-prone systems. In: Proceedings 35th ACM Symposium on Principles of Distributed Computing (PODC 2016), pp. 381–390. ACM Press (2016)
36. Mostéfaoui, A., Raynal, M.: Signature-free asynchronous Byzantine systems: from multivalued to binary consensus with $t<n/3$, $O(n^2)$ messages, and constant time. Acta Informatica **54**(5), 501–520 (2017)
37. Peleg, D.: Distributed Computing, A Locally-sensitive Approach. SIAM Monographs on Discrete Mathematics and Applications, 343 pages (2000)
38. Rabin M.O.: Randomized Byzantine generals. In: Proceedings of the 24th Annual Symposium on Foundations of Computer Science (FOCS 1983), pp. 403–409. IEEE Preess (1983)
39. Rajsbaum, S.: Indistinguishability. In: Talk at Yoram Moses Birthday Celebration, DISC (2017)
40. Raynal M., Concurrent Programming: Algorithms, Principles, and Foundations, 530 pages. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-32027-9. ISBN 978-3-642-32026-2
41. Raynal M., Distributed Algorithms for Message-Passing Systems, 515 pages. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38123-2. ISBN 978-3-642-38122-5
42. Raynal, M.: Parallel computing vs. distributed computing: a great confusion? (position paper). In: Hunold, S., et al. (eds.) Euro-Par 2015. LNCS, vol. 9523, pp. 41–53. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-27308-2_4
43. Raynal, M.: Distributed universal constructions: a guided tour. Bull. Eur. Assoc. Theoret. Comput. Sci. (EATCS) **121**(1), 64–96 (2017)

44. Raynal, M.: Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach, 560 pages. Springer (to appear, 2018)
45. Santoro, N., Widmayer, P.: Time is not a healer. In: Monien, B., Cori, R. (eds.) STACS 1989. LNCS, vol. 349, pp. 304–313. Springer, Heidelberg (1989). https://doi.org/10.1007/BFb0028994
46. Santoro, N., Widmayer, P.: Agreement in synchronous networks with ubiquitous faults. Theoret. Comput. Sci. **384**(2–3), 232–249 (2007)
47. Schneider, F.B.: Implementing fault-tolerant services using the state machine approach. ACM Comput. Surv. **22**(4), 299–319 (1990)
48. Schneider, F.B., Zhou, L.: Implementing trustworthy services using replicated state machines. In: Charron-Bost, B., Pedone, F., Schiper, A. (eds.) Replication. LNCS, vol. 5959, pp. 151–167. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-642-11294-2_8
49. Taubenfeld, G.: Synchronization Algorithms and Concurrent Programming, 423 pages. Pearson Education/Prentice Hall, Upper Saddle River (2006). ISBN 0-131-97259-6
50. Taubenfeld, G.: The computational structure of progress conditions. In: Lynch, N.A., Shvartsman, A.A. (eds.) DISC 2010. LNCS, vol. 6343, pp. 221–235. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15763-9_23

Sequential Grammars with Activation and Blocking of Rules

Artiom Alhazov[1], Rudolf Freund[2](✉), and Sergiu Ivanov[3]

[1] Institute of Mathematics and Computer Science, Academiei 5, 2028 Chişinău, Moldova
artiom@math.md

[2] Faculty of Informatics, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
rudi@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay, 23 Boulevard de France, 91025 Évry, France
sergiu.ivanov@univ-evry.fr

Abstract. We introduce new possibilities to control the application of rules based on the preceding application of rules which can be defined for a general model of sequential grammars and we show some similarities with other control mechanisms such as graph-controlled grammars and matrix grammars with and without appearance checking, as well as grammars with random context conditions. Using both activation and blocking of rules, in the string and in the multiset case we can show computational completeness of context-free grammars equipped with the control mechanism of activation and blocking of rules even when using only two nonterminal symbols. With one- and two-dimensional #-context-free array grammars, computational completeness can already be obtained by only using activation of rules.

1 Introduction

Nearly thirty years ago, the monograph on regulated rewriting by Dassow and Păun [2] already gave a first comprehensive overview on many concepts of regulated rewriting, especially for the string case. Yet as it turned out later, many of the mechanisms considered there for guiding the application of productions/rules can also be applied to other objects than strings, e.g., to n-dimensional arrays [4]. Even in the field of P systems [9,13] where mostly multisets are considered, such regulating mechanisms were used [1]. As exhibited in [5], for comparing the generating power of grammars working in the sequential derivation mode, many relations between various regulating mechanisms can be established in a very general setting without any reference to the underlying objects the rules are

The work is supported by National Natural Science Foundation of China (61320106005, 61033003, and 61772214) and the Innovation Scientists and Technicians Troop Construction Projects of Henan Province (154200510012).

J. Durand-Lose and S. Verlan (Eds.): MCU 2018, LNCS 10881, pp. 51–68, 2018.
https://doi.org/10.1007/978-3-319-92402-1_3

working on, using a general model for graph-controlled, programmed, random-context, and ordered grammars of arbitrary type based on the applicability of rules.

In the following section, we recall some notions from formal language theory as well as the main definitions of the general framework for sequential grammars elaborated in [5]. Then we define the new concept of activation and blocking of rules based on the applicability of rules within this general framework for regulated rewriting. In Sect. 3 some general results for sequential grammars using the control mechanism of activation or activation and blocking of rules are established. Specific results on computational completeness for strings, multisets, and arrays as underlying objects then are shown in Sect. 4. In Sect. 5 we establish our main results for strings and multisets showing that context-free (string and multiset) grammars with activation and blocking of rules are computationally complete even when only two non-terminal symbols are used, which establishes a sharp border as one non-terminal symbol is not sufficient. Finally, a summary of the shown results and some future research topics extending the notions and results established in this paper are given in Sect. 6.

2 Definitions

After some preliminaries from formal language theory, we define our general model for sequential grammars and recall some notions for string, array, and multiset grammars in the general setting of this paper. Then we formulate the models of graph-controlled, programmed, matrix grammars with and without appearance checking, as well as random-context grammars, based on the applicability of rules.

2.1 Preliminaries

The set of integers is denoted by $\mathbb{Z}$, the set of non-negative integers by $\mathbb{N}_0$, and the set of positive integers (natural numbers) by $\mathbb{N}$. An *alphabet* V is a finite non-empty set of abstract *symbols*. Given V, the free monoid generated by V under the operation of concatenation is denoted by V^*; the elements of V^* are called strings, and the *empty string* is denoted by λ; $V^* \setminus \{\lambda\}$ is denoted by V^+. Let $\{a_1, ..., a_n\}$ be an arbitrary alphabet; the number of occurrences of a symbol a_i in x is denoted by $|x|_{a_i}$; the *Parikh vector* associated with x with respect to $a_1, ..., a_n$ is $\left(|x|_{a_1}, ..., |x|_{a_n}\right)$. The *Parikh image* of a language L over $\{a_1, ..., a_n\}$ is the set of all Parikh vectors of strings in L, and we denote it by $Ps(L)$. For a family of languages FL, the family of Parikh images of languages in FL is denoted by $PsFL$.

A finite multiset over the finite alphabet V, $V = \{a_1, ..., a_n\}$, is a mapping $f : V \longrightarrow \mathbb{N}_0$ and represented by $\langle f(a_1), a_1\rangle ... \langle f(a_n), a_n\rangle$ or by any string x the Parikh vector of which with respect to $a_1, ..., a_n$ is $(f(a_1), ..., f(a_n))$. In the following we will not distinguish between a vector $(m_1, ..., m_n)$, its representation by a multiset $\langle m_1, a_1\rangle ... \langle m_n, a_n\rangle$ or its representation by a string x

having the Parikh vector $\left(|x|_{a_1}, ..., |x|_{a_n}\right) = (m_1, ..., m_n)$. Fixing the sequence of symbols $a_1, ..., a_n$ in the alphabet V in advance, the representation of the multiset $\langle m_1, a_1\rangle ... \langle m_n, a_n\rangle$ by the string $a_1^{m_1} ... a_n^{m_n}$ is unique. The set of all finite multisets over an alphabet V is denoted by V°.

For more details of formal language theory the reader is referred to the monographs and handbooks in this area [2,11].

2.2 A General Model for Sequential Grammars

We first recall the main definitions of the general model for sequential grammars as established in [5], grammars generating a set of terminal objects by derivations where in each derivation step exactly one rule is applied to exactly one object. This does not cover rules involving more than one object – as, for example, splicing rules – or other derivation modes – as, for example, the maximally parallel mode considered in many variants of P systems [9].

A *(sequential) grammar* G is a construct $(O, O_T, w, P, \Longrightarrow_G)$ where

- O is a set of *objects* (often an infinite set);
- $O_T \subseteq O$ is a set of *terminal objects*;
- $w \in O$ is the *axiom (start object)*;
- P is a finite set of *rules*;
- $\Longrightarrow_G \subseteq O \times O$ is the *derivation relation* of G.
  We assume that each of the rules $p \in P$ induces a relation $\Longrightarrow_p \subseteq O \times O$ with respect to $\Longrightarrow_G$ fulfilling at least the following conditions: (i) for each object $x \in O$, $(x, y) \in \Longrightarrow_p$ for only finitely many objects $y \in O$; (ii) there exists a finitely described mechanism as, for example, a Turing machine, which, given an object $x \in O$, computes all objects $y \in O$ such that $(x, y) \in \Longrightarrow_p$. A rule $p \in P$ is called *applicable* to an object $x \in O$ if and only if there exists at least one object $y \in O$ such that $(x, y) \in \Longrightarrow_p$; we also write $x \Longrightarrow_p y$. The derivation relation $\Longrightarrow_G$ is the union of all $\Longrightarrow_p$, i.e., $\Longrightarrow_G = \cup_{p\in P} \Longrightarrow_p$. The reflexive and transitive closure of $\Longrightarrow_G$ is denoted by $\overset{*}{\Longrightarrow}_G$.

In the following we shall consider different types of grammars depending on the components of G (where the set of objects O is infinite, e.g., V^*, the set of strings over the alphabet V), especially with respect to different types of rules (e.g., context-free string rules). Some specific conditions on the elements of G, especially on the rules in P, may define a special type X of grammars which then will be called *grammars of type* X. The *language generated by* G is the set of all terminal objects (we also assume $v \in O_T$ to be decidable for every $v \in O$) derivable from the axiom, i.e.,

$$L(G) = \left\{ v \in O_T \mid w \overset{*}{\Longrightarrow}_G v \right\}.$$

The family of languages generated by grammars of type X is denoted by $\mathcal{L}(X)$.

Let $G = (O, O_T, w, P, \Longrightarrow_G)$ be a grammar of type X. If for every G of type X we have $O_T = O$, then X is called a *pure* type, otherwise it is called *extended*;

X is called *strictly extended* if for any grammar G of type X, $w \notin O_T$ and for all $x \in O_T$, no rule from P can be applied to x.

In many cases, the type X of the grammar allows for one or even both of the following features:

A type X of grammars is called a *type with unit rules* if for every grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X a grammar $G' = \left(O, O_T, w, P \cup P^{(+)}, \Longrightarrow_{G'}\right)$ of type X exists such that $\Longrightarrow_G \;\subseteq\; \Longrightarrow_{G'}$ and

- $P^{(+)} = \left\{p^{(+)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(+)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(+)}$ is applicable to x, the application of $p^{(+)}$ to x yields x back again.

A type X of grammars is called a *type with trap rules* if for every grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ of type X a grammar $G' = \left(O, O_T, w, P \cup P^{(-)}, \Longrightarrow_{G'}\right)$ of type X exists such that $\Longrightarrow_G \;\subseteq\; \Longrightarrow_{G'}$ and

- $P^{(-)} = \left\{p^{(-)} \mid p \in P\right\}$,
- for all $x \in O$, $p^{(-)}$ is applicable to x if and only if p is applicable to x, and
- for all $x \in O$, if $p^{(-)}$ is applicable to x, the application of $p^{(-)}$ to x yields an object y from which no terminal object can be derived anymore.

2.3 Specific Types of Objects

String Grammars. In the general notion as defined above, a *string grammar* G_S is represented as

$$\left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$$

where N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, $w \in (N \cup T)^+$, P is a finite set of *rules* of the form $u \rightarrow v$ with $u \in V^*$ (for generating grammars, $u \in V^+$) and $v \in V^*$ (for accepting grammars, $v \in V^+$), with $V := N \cup T$; the derivation relation for $u \rightarrow v \in P$ is defined by $xuy \Longrightarrow_{u \rightarrow v} xvy$ for all $x, y \in V^*$, thus yielding the well-known derivation relation $\Longrightarrow_{G_S}$ for the string grammar G_S. In the following, we shall also use the common notation $G_S = (N, T, w, P)$ instead, too. We remark that, usually, the axiom w is supposed to be a non-terminal symbol, i.e., $w \in V \setminus T$, and is called the *start symbol*.

As special types of string grammars we consider string grammars with arbitrary rules and context-free rules of the form $A \rightarrow v$ with $A \in N$ and $v \in V^*$. The corresponding types of grammars are denoted by ARB an CF, thus yielding the families of languages $\mathcal{L}(ARB)$, i.e., the family of recursively enumerable languages (also denoted by RE), as well as $\mathcal{L}(CF)$, i.e., the family of context-free languages, respectively. Observe that the types ARB and CF are types with unit rules and trap rules (for $p = w \rightarrow v \in P$, we can take $p^{(+)} = w \rightarrow w$ and $p^{(-)} = w \rightarrow F$ where $F \notin T$ is a new symbol – the trap symbol).

We refer to [5] where some examples for string grammars of specific types illustrating the expressive power of this general framework are given.

Array Grammars. We now introduce the basic notions for n-dimensional arrays and array grammars, for example, see [4,10,12].

Let $d \in \mathbb{N}$. Then a *d-dimensional array* $\mathcal{A}$ over an alphabet V is a function $\mathcal{A} : \mathbb{Z}^d \to V \cup \{\#\}$, where *shape* $(\mathcal{A}) = \left\{v \in \mathbb{Z}^d \mid \mathcal{A}(v) \neq \#\right\}$ is finite and $\# \notin V$ is called the *background* or *blank symbol*. We usually write $\mathcal{A} = \{(v, \mathcal{A}(v)) \mid v \in \mathit{shape}(\mathcal{A})\}$.

The set of all d-dimensional arrays over V is denoted by V^{*d}. The *empty array* in V^{*d} with empty shape is denoted by Λ_d. Moreover, we define $V^{+d} = V^{*d} \setminus \{\Lambda_d\}$.

Let $v \in \mathbb{Z}^d$, $v = (v_1, \ldots, v_d)$. The *translation* $\tau_v : \mathbb{Z}^d \to \mathbb{Z}^d$ is defined by $\tau_v(w) = w + v$ for all $w \in \mathbb{Z}^d$, and for any array $\mathcal{A} \in V^{*d}$ we define $\tau_v(\mathcal{A})$, the corresponding d-dimensional array translated by v, by $(\tau_v(\mathcal{A}))(w) = \mathcal{A}(w - v)$ for all $w \in \mathbb{Z}^d$. The vector $(0, \ldots, 0) \in \mathbb{Z}^d$ is denoted by Ω_d.

A *d-dimensional array rule* p over V is a triple $(W, \mathcal{A}_1, \mathcal{A}_2)$, where $W \subseteq \mathbb{Z}^d$ is a finite set and $\mathcal{A}_1$ and $\mathcal{A}_2$ are mappings from W to $V \cup \{\#\}$ such that *shape* $(\mathcal{A}_1) \neq \emptyset$. We say that the array $\mathcal{B}_2 \in V^{*d}$ is *directly derivable* from the array $\mathcal{B}_1 \in V^{*d}$ by the d-dimensional array rule $(W, \mathcal{A}_1, \mathcal{A}_2)$, i.e., $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$, if and only if there exists a vector $v \in \mathbb{Z}^d$ such that $\mathcal{B}_1(w) = \mathcal{B}_2(w)$ for all $w \in \mathbb{Z}^d \setminus \tau_v(W)$ as well as $\mathcal{B}_1(w) = \mathcal{A}_1(\tau_{-v}(w))$ and $\mathcal{B}_2(w) = \mathcal{A}_2(\tau_{-v}(w))$ for all $w \in \tau_v(W)$, i.e., the subarray of $\mathcal{B}_1$ corresponding to $\mathcal{A}_1$ is replaced by $\mathcal{A}_2$, thus yielding $\mathcal{B}_2$. In the following, we shall also write $\mathcal{A}_1 \to \mathcal{A}_2$, because W is implicitly given by the finite arrays $\mathcal{A}_1, \mathcal{A}_2$.

A *d-dimensional array grammar* G_A is represented as

$$\left((N \cup T)^{*d}, T^{*d}, \{(v_0, S)\}, P, \Longrightarrow_{G_A}\right) \text{ where}$$

- N is the alphabet of *non-terminal symbols*;
- T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$;
- $\{(v_0, S)\}$ is the *start array (axiom)* with $S \in N$ and $v_0 \in \mathbb{Z}^d$*;*
- P is a finite set of *d-dimensional array rules* over V, $V := N \cup T$;
- $\Longrightarrow_{G_A}$ is the derivation relation induced by the array rules in P according to the explanations given above, i.e., for arbitrary $\mathcal{B}_1, \mathcal{B}_2 \in V^{*d}$, $\mathcal{B}_1 \Longrightarrow_{G_A} \mathcal{B}_2$ if and only if there exists a d-dimensional array rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P such that $\mathcal{B}_1 \Longrightarrow_p \mathcal{B}_2$.

A d-dimensional array rule $p = (W, \mathcal{A}_1, \mathcal{A}_2)$ in P is called *#-context-free*, if *shape* $(\mathcal{A}_1) = \{\Omega_d\}$ and $\mathcal{A}_1(\Omega_d) \in N$. A d-dimensional array grammar is said to be of type *d-ARBA*, *d-#-CFA* if every array rule in P is of the corresponding type, i.e., an arbitrary and #-context-free d-dimensional array rule, respectively. The corresponding families of d-dimensional array languages of type X are denoted by $\mathcal{L}(X)$, i.e., $\mathcal{L}(d\text{-}ARBA)$ and $\mathcal{L}(d\text{-}\#\text{-}CFA)$ are the families of recursively enumerable and #-context-free d-dimensional array languages, respectively.

Observe that the types *d-ARBA* and *d-#-CFA* are types with unit rules and trap rules – for $p = (W, \mathcal{A}_1, \mathcal{A}_2)$, we can take $p^{(+)} = (W, \mathcal{A}_1, \mathcal{A}_1)$ and $p^{(-)} = (W, \mathcal{A}_1, \mathcal{A}_F)$ with $\mathcal{A}_F(v) = F$ for $v \in W$, where F is a new non-terminal symbol – the trap symbol.

Multiset Grammars. $G_m = \left((N \cup T)^\circ, T^\circ, w, P, \Longrightarrow_{G_m}\right)$ is called a *multiset grammar*; N is the alphabet of *non-terminal symbols*, T is the alphabet of *terminal symbols*, $N \cap T = \emptyset$, w is a non-empty multiset over V, $V := N \cup T$, and P is a finite set of multiset rules yielding a derivation relation $\Longrightarrow_{G_m}$ on the multisets over V; the application of the rule $u \to v$ to a multiset x has the effect of replacing the multiset u contained in x by the multiset v. For the multiset grammar G_m we also write $(N, T, w, P, \Longrightarrow_{G_m})$.

As special types of multiset grammars we consider multiset grammars with *arbitrary* rules as well as *context-free* (*non-cooperative*) rules of the form $A \to v$ with $A \in N$ and $v \in V^\circ$; the corresponding types X of multiset grammars are denoted by $mARB$ and mCF, thus yielding the families of multiset languages $\mathcal{L}(X)$. Observe that $mARB$ and mCF are types with unit rules and trap rules (for $p = w \to v \in P$, we can take $p^{(+)} = w \to w$ and $p^{(-)} = w \to F$ where F is a new symbol – the trap symbol). Even with arbitrary multiset rules, it is not possible to get $Ps(\mathcal{L}(ARB))$ [7]:

$$\mathcal{L}(mCF) = Ps(\mathcal{L}(CF)) \subsetneq \mathcal{L}(mARB) \subsetneq Ps(\mathcal{L}(ARB)).$$

2.4 Register Machines

As a computationally complete model able to generate/accept all sets in $PsRE = Ps(\mathcal{L}(ARB))$ we use register machines/deterministic register machines:

A *register machine* is a construct $M = (n, L_M, R_M, p_0, h)$ where n, $n \geq 1$, is the number of registers, L_M is the set of instruction labels, p_0 is the start label, h is the halting label (only used for the `HALT` instruction), and R_M is a set of (labeled) instructions being of one of the following forms:

- $p : (\texttt{ADD}(r), q, s)$ increments the value in register r and continues with the instruction labeled by q or s,
- $p : (\texttt{SUB}(r), q, s)$ decrements the value in register r and continues the computation with the instruction labeled by q if the register was non-empty, otherwise it continues with the instruction labeled by s;
- $h : \texttt{HALT}$ halts the machine.

M is called deterministic if in all ADD-instructions $p : (\texttt{ADD}(r), q, s)$ $q = s$; in this case we write $p : (\texttt{ADD}(r), q)$. Deterministic register machines can accept all recursively enumerable sets of vectors of natural numbers with k components using exactly $k+2$ registers, for instance, see [8].

2.5 Graph-Controlled and Programmed Grammars

A *graph-controlled grammar* (with applicability checking) of type X is a construct

$$G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X; $g = (H, E, K)$ is a labeled graph where H is the set of node labels identifying the nodes of the graph in a one-to-one manner, $E \subseteq H \times \{Y, N\} \times H$ is the set of edges labeled by Y or N, $K : H \to 2^P$ is a function assigning a subset of P to each node of g; $H_i \subseteq H$ is the set of initial labels, and $H_f \subseteq H$ is the set of final labels. The derivation relation $\Longrightarrow_{GC}$ is defined based on $\Longrightarrow_G$ and the control graph g as follows: For any $i, j \in H$ and any $u, v \in O$, $(u, i) \Longrightarrow_{GC} (v, j)$ if and only if

- $u \Longrightarrow_p v$ by some rule $p \in K(i)$ and $(i, Y, j) \in E$ *(success case)*, **or**
- $u = v$, no $p \in K(i)$ is applicable to u, and $(i, N, j) \in E$ *(failure case)*.

The language generated by G_{GC} is defined by

$$L(G_{GC}) = \left\{ v \in O_T \mid (w, i) \Longrightarrow^*_{G_{GC}} (v, j),\ i \in H_i, j \in H_f \right\}.$$

If $H_i = H_f = H$, then G_{GC} is called a *programmed grammar*. The families of languages generated by graph-controlled and programmed grammars of type X are denoted by $\mathcal{L}(X\text{-}GC_{ac})$ and $\mathcal{L}(X\text{-}P_{ac})$, respectively. If the set E contains no edges of the form (i, N, j), then the graph-controlled grammar is said to be *without applicability checking*; the corresponding families of languages are denoted by $\mathcal{L}(X\text{-}GC)$ and $\mathcal{L}(X\text{-}P)$, respectively.

The notions and concepts *with/without applicability checking* were introduced as *with/without appearance checking* in the original definition for string grammars because the appearance of the non-terminal symbol on the left-hand side of a context-free rule was checked, which coincides with checking for the applicability of this rule in our general model; in both cases – applicability checking and appearance checking – we can use the abbreviation *ac*.

2.6 Matrix Grammars

A *matrix grammar* (with applicability checking) of type X is a construct

$$G_M = (G, M, F, \Longrightarrow_{G_M})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, M is a finite set of sequences of the form $(p_1, \ldots, p_n)$, $n \geq 1$, of rules in P, and $F \subseteq P$. For $w, z \in O$ we write $w \Longrightarrow_{G_M} z$ if there are a matrix $(p_1, \ldots, p_n)$ in M and objects $w_i \in O$, $1 \leq i \leq n + 1$, such that $w = w_1$, $z = w_{n+1}$, and, for all $1 \leq i \leq n$, either

- $w_i \Longrightarrow_G w_{i+1}$ or
- $w_i = w_{i+1}$, p_i is not applicable to w_i, and $p_i \in F$.

$L(G_M) = \left\{ v \in O_T \mid w \Longrightarrow^*_{G_M} v \right\}$ is the language generated by G_M. The family of languages generated by matrix grammars of type X is denoted by $\mathcal{L}(X\text{-}MAT_{ac})$. If the set F is empty, then the grammar is said to be *without applicability checking*; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}MAT)$.

Remark 1. We mention that in this paper we choose the definition where the sequential application of the rules of the final matrix may stop at any moment.

2.7 Random-Context Grammars

The following general notion of a random context-grammar had already been introduced in [1,6] in a similar way before it was formulated in [5].

A *random-context grammar* G_{RC} *of type* X is a construct $(G, P', \Longrightarrow_{G_{RC}})$ where

– $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X;
– P' is a set of rules of the form (q, R, Q) where $q \in P$, $R \cup Q \subseteq P$;
– $\Longrightarrow_{G_{RC}}$ is the derivation relation assigned to G_{RC} such that for any $x, y \in O$, $x \Longrightarrow_{G_{RC}} y$ if and only if for some rule $(q, R, Q) \in P'$, $x \Longrightarrow_q y$ and, moreover, all rules from R are applicable to x as well as no rule from Q is applicable to x.

A random-context grammar $G_{RC} = (G, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with permitting contexts of type* X if for all rules (q, R, Q) in P' we have $Q = \emptyset$, i.e., we only check for the applicability of the rules in R. A random-context grammar $G_{RC} = (G, P', \Longrightarrow_{G_{RC}})$ of type X is called a *grammar with forbidden contexts of type* X if for all rules (q, R, Q) in P' we have $R = \emptyset$, i.e., we only check for the non-applicability of the rules in Q.

$L(G_{RC}) = \left\{v \in O_T \mid w \Longrightarrow^*_{G_{RC}} v\right\}$ is the language generated by G_{RC}. The families of languages generated by random context grammars, grammars with permitting contexts, and grammars with forbidden contexts of type X are denoted by $\mathcal{L}(X\text{-}RC)$, $\mathcal{L}(X\text{-}pC)$, and $\mathcal{L}(X\text{-}fC)$, respectively.

2.8 Grammars with Activation and Blocking of Rules

We now define our new concept of regulating the application of rules at a specific moment by activation and blocking relations.

A *grammar with activation and blocking of rules* (an *AB-grammar* for short) of type X is a construct

$$G_{AB} = (G, L, f_L, A, B, L_0, \Longrightarrow_{G_{AB}})$$

where $G = (O, O_T, w, P, \Longrightarrow_G)$ is a grammar of type X, L is a finite set of labels with each label having assigned one rule from P by the function f_L, A, B are finite subsets of $L \times L \times \mathbb{N}$, and L_0 is a finite set of tuples of the form $\left(q, Q, \bar{Q}\right)$, $q \in L$, with the elements of $Q, \bar{Q}$ being of the form (l, t), where $l \in L$ and $t \in \mathbb{N}$, $t > 1$.

A derivation in G_{AB} starts with one element $\left(q, Q, \bar{Q}\right)$ from L_0 which means that the rule labeled by q has to be applied to the initial object w in the first step and for the following derivation steps the conditions given by Q as activations of rules and $\bar{Q}$ as blockings of rules have to be taken into account in addition to the activations and blockings coming along with the application of the rule labeled by q. The role of L_0 is to get a derivation started by activating some rule for the first step although no rule has been applied so far, but probably also providing additional activations and blockings for further derivation steps.

A configuration of G_{AB} in general can be described by the object derived so far and the activations Q and blockings $\bar{Q}$ for the next steps. In that sense, the starting tuple $\left(q, Q, \bar{Q}\right)$ can be interpreted as $\left(\{(q,1)\} \cup Q, \bar{Q}\right)$, and we may also simply write $\left(Q', \bar{Q}\right)$ with $Q' = \{(q,1)\} \cup Q$. We will always assume Q and $\bar{Q}$ to be non-conflicting, i.e., $Q \cap \bar{Q} = \emptyset$. Given such a configuration $\left(u, Q, \bar{Q}\right)$, in one step we can derive $\left(v, R, \bar{R}\right)$, and we also write

$$\left(u, Q, \bar{Q}\right) \Longrightarrow_{G_{AB}} \left(v, R, \bar{R}\right) \text{ if and only if}$$

– $u \Longrightarrow_G v$ using the rule r such that $(q,1) \in Q$ and $(q,r) \in f_L$, i.e., we apply the rule labeled by q activated for this next derivation step to u; the new sets of activations and blockings are defined by

$$\begin{aligned} \bar{R} &= \left\{(x,i) \mid (x,i+1) \in \bar{Q},\ i > 0\right\} \cup \left\{(x,i) \mid (q,x,i) \in B\right\}, \\ R &= \left(\{(x,i) \mid (x,i+1) \in Q,\ i > 0\} \cup \{(x,i) \mid (q,x,i) \in A\}\right) \\ &\quad \setminus \left\{(x,i) \mid (x,i) \in \bar{R}\right\} \end{aligned}$$

(observe that R and $\bar{R}$ are made non-conflicting by eliminating rule labels which are activated and blocked at the same time);
or

– no rule r is activated to be applied in the next derivation step; in this case we take $v = u$ and continue with $\left(v, R, \bar{R}\right)$ constructed as before provided R is not empty, i.e., there are rules activated in some further derivation steps; otherwise the derivation stops.

The language generated by G_{AB} is defined by

$$L(G_{AB}) = \left\{v \in O_T \mid \left(w, Q, \bar{Q}\right) \Longrightarrow_{G_{AB}}^{*} \left(v, R, \bar{R}\right) \text{ for some } \left(Q, \bar{Q}\right) \in L_0\right\}.$$

The family of languages generated by AB-grammars of type X is denoted by $\mathcal{L}(X\text{-}AB)$. If the set B of blocking relations is empty, then the grammar is said to be a *grammar with activation of rules* (an *A-grammar* for short) of type X; the corresponding family of languages is denoted by $\mathcal{L}(X\text{-}A)$. Moreover, an A-grammar is called an *A*1*-grammar* if for all $(p,q,t) \in A$ we have $t = 1$, which means that only the rule applied in one derivation step activates the rules which can be applied in the next step; in this case we may only write (p,q) instead of $(p,q,1)$. Moreover, in L_0 we may simply list the labels of the rules to be applied in the first step.

Example 1. Consider the string grammar $G_S = \left((N \cup T)^*, T^*, w, P, \Longrightarrow_P\right)$ with $N = \{A, B, C\}$, $T = \{a, b, c\}$, $w = ABC$, and the set of rules
$P = \{A \to aA, B \to bB, C \to cC, A \to \lambda, B \to \lambda, C \to \lambda\}$,
as well as the A1-grammar
$G_A = (G, L, f_L, A, L_0, \Longrightarrow_{G_A})$ with
$L = \{p_a, p_b, p_c, p_A, p_B, p_C\}$, and, writing $p : r$ for the pairs (p,r) in f_L,
$f_L = \{p_a : A \to aA, p_b : B \to bB, p_c : C \to cC\}$
$\quad \cup \ \{p_A : A \to \lambda, p_B : B \to \lambda, p_C : C \to \lambda\}$
$A = \{(p_a, p_b), (p_b, p_c), (p_c, p_a), (p_c, p_A), (p_A, p_B), (p_B, p_C)\}$, and
$P_0 = \{p_a, p_A\}$

The underlying string grammar generates the regular set $\{a\}^* \{b\}^* \{c\}^*$, whereas the A1-grammar G_A generates $\{a^n b^n c^n \mid n \in \mathbb{N}_0\}$: starting with the rule labeled by p_a from L_0, the rules corresponding to the sequence of labels $p_a p_b p_c$ is applied $n \geq 1$ times, and finally we switch to the sequence of rules given by $p_A p_B p_C$ whereafter no rule can be applied any more. Starting with p_A yields the empty string.

3 General Results

In this section, we elaborate some general results holding true for many types of grammars, some even holding for any type X, whereas some of them rely on specific conditions on X.

3.1 Matrix Grammars and A1-Grammars

Our first results show a close connection between matrix grammars without appearance checking and A1-grammars:

Theorem 1. *For any type X, $\mathcal{L}(X\text{-}MAT) \subseteq \mathcal{L}(X\text{-}A1)$.*

Proof. Let $G_M = (G, M, F, \Longrightarrow_{G_M})$ be a matrix grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being a grammar of type X; let $M = \{(p_{i,1}, \ldots, p_{i,n_i}) \mid 1 \leq i \leq n\}$ with $p_{i,j} \in P$, $1 \leq j \leq n_i$, $1 \leq i \leq n$.

We construct the equivalent A1-grammar

$$\begin{aligned}
G_A &= (G, L, f_L, A, L_0, \Longrightarrow_{G_A}), \\
L &= \{l_{i,j} \mid 1 \leq j \leq n_i,\ 1 \leq i \leq n\}, \\
f_L &= \{(l_{i,j}, p_{i,j}) \mid 1 \leq j \leq n_i,\ 1 \leq i \leq n\}, \\
A &= \{(l_{i,j}, l_{i,j+1}) \mid 1 \leq j < n_i, 1 \leq i \leq n\} \\
&\quad \cup \{(l_{i,n_i}, l_{j,1}) \mid 1 \leq j \leq n, 1 \leq i \leq n\}, \\
L_0 &= \{l_{i,1} \mid 1 \leq i \leq n\}.
\end{aligned}$$

We mention that according to our definitions the sequential application of the rules of the chosen matrix may stop at any moment if the next rule cannot be applied, in which case also the simulation in the A1-grammar stops. □

For the special cases of strings, multisets, and arrays as underlying objects, also the reverse inclusion holds:

Theorem 2. *For $X \in \{CF, mCF\} \cup \{d\text{-}\#\text{-}CFA \mid d \in \mathbb{N}\}$,*

$$\mathcal{L}(X\text{-}A1) \subseteq \mathcal{L}(X\text{-}MAT).$$

Proof. The rules in the set of rules P of the underlying sequential grammar G in the given A1-grammar

$$G_A = (G, L, f_L, A, L_0, \Longrightarrow_{G_A})$$

are labeled by elements from the set L of labels. We extend the set of non-terminal symbols N in G to N' by allowing the additional symbols to store the information which rules can be applied in the next step:

$$N' = N \cup \{\langle X, M\rangle \mid X \in N, M \subseteq L\}.$$

From G_A we then construct the equivalent matrix grammar

$$G_M = (G', M, \Longrightarrow_{G_M})$$

as follows: G' contains N' instead of N as set of non-terminal symbols, and the set of rules R' in G' includes all rules from G and in addition all the rules used in the matrices described below.

– As starting matrix we use $(S \to \langle S, L_0\rangle)$, where S is the start symbol (in G as well as in G'). From now on, every object derived in G' contains exactly one symbol from $\{\langle X, M\rangle \mid X \in N, M \subseteq L\}$ until in the last step the terminal object is derived.
– A derivation step in G by applying the rule labeled by p with the non-terminal symbol X on its left-hand side now is simulated in G' by any of the matrices

$$(\langle X, M\rangle \to X, r(p), Y \to \langle Y, K\rangle).$$

with $p \in M$ and $K = \{q \mid q \in L, (p, q) \in A\}$ is the set of all rules enabled by the application of the rule labeled by p in the A1-grammar G_A.

In the first step, we regain the non-terminal symbol X from $\langle X, M\rangle$, then the rule $r(p)$, i.e., the rule labeled by p, is applied, and finally we non-deterministically choose any non-terminal symbol Y still occurring in the derived object to carry the information which rules can be applied in the next step.

We finally observe that only when the application of the last rule yields a terminal object, the final rule $Y \to \langle Y, K\rangle$ cannot be applied any more, thus also ending the derivation in the matrix grammar G_M in a correct way. □

Corollary 1. *For $X \in \{CF, mCF\} \cup \{d\text{-}\#\text{-}CFA \mid d \in \mathbb{N}\}$,*

$$\mathcal{L}(X\text{-}MAT) = \mathcal{L}(X\text{-}A1).$$

3.2 Random Context Grammars and AB-Grammars

For any type X with unit rules, random context grammars of type X can be simulated by AB-grammars of type X.

Remark 2. In order to keep proofs shorter, in the following, instead of specifying the set of rules P, the set of labels L, and the function f_L assigning rules to the labels separately, we will only specify the corresponding labeled rules of the form $l : r$ with $l \in L$, $r \in P$, and $(l,r) \in f_L$. Moreover, for $X \in \{A,B\}$, instead of $(p,q,t) \in X$, we write $(p,q,t)_X$.

Theorem 3. *For any type X with unit rules, $\mathcal{L}(X\text{-}RC) \subseteq \mathcal{L}(X\text{-}AB)$.*

Proof. Let $(G, R, \Longrightarrow_{G_{RC}})$ be a random context grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of a type X with unit rules, where

$$\begin{aligned} R &= \{(r_i, P_i, Q_i) \mid 1 \le i \le n\}, r_i \in P,\ 1 \le i \le n, \\ P_i &= \{p_{i,j} \mid 1 \le j \le m_i,\ 1 \le i \le n\}, m_i \ge 0,\ 1 \le i \le n, \\ Q_i &= \{q_{i,j} \mid 1 \le j \le n_i,\ 1 \le i \le n\}, n_i \ge 0,\ 1 \le i \le n. \end{aligned}$$

Then we construct an AB-grammar G_{AB} of type X as follows:

$$\begin{aligned} G_{AB} &= (G', L, f_L, A, B, L_0, \Longrightarrow_{G_A}), \\ G' &= (O, O_T, w, P', \Longrightarrow_{G'}), \\ P' &= P \cup \{r^+ \mid r \in P\}; \\ L_0 &= \{l_{r_i} \mid 1 \le i \le n\}; \end{aligned}$$

the application of a random context rule (r_i, P_i, Q_i) is simulated by the following sequence of labeled rules together with suitable activations and blockings of rules:

- $l_{r_i} : {r_i}^+$, $(l_{r_i}, l_{r_i,1})_A$, $(l_{r_i}, \bar{l}_{r_i,j}, m_i + j)_A$, $1 \le j \le n_i$; at the beginning, the checking of all rules which should not be applicable is initiated, and the sequence of applicability checkings for the rules in P_i is started;
- $l_{r_i,j} : {p_{i,j}}^+$, $(l_{r_i,j}, l_{r_i,j+1})_A$, $1 \le j < m_i$;
- $l_{r_i,m_i} : {p_{i,m_i}}^+$, $(l_{r_i,m_i}, \hat{l}_{r_i}, n_i + 1)_A$; when all rules in P_i have been checked to be applicable, the application of rule r_i after further n_i steps is activated; yet if any of the rules in Q_i is applicable, then this application of rule r_i is blocked;
- $\bar{l}_{r_i,j} : {q_{i,j}}^+$, $(\bar{l}_{r_i,j}, \hat{l}_{r_i}, n_i - j + 1)_B$, $1 \le j \le n_i$;
- $\hat{l}_{r_i} : r_i$, $(\hat{l}_{r_i}, l_{r_k})$, $1 \le k \le n$; after the successful application of rule r we may continue with trying to apply any random context rule from R.

We finally observe that only unit rules and no trap rules as in other simulations known from [5] are needed to obtain this result. □

3.3 AB-Grammars and Graph-Controlled Grammars

Already in [5] graph-controlled grammars have been shown to be the most powerful control mechanism, and they can also simulate AB-grammars with the underlying grammar being of any arbitrary type X.

Theorem 4. *For any type X, $\mathcal{L}(X\text{-}AB) \subseteq \mathcal{L}(X\text{-}GC_{ac})$.*

Proof. Let $G_{AB} = (G', L, f_L, A, B, L_0, \Longrightarrow_{G_{AB}})$ be an AB-grammar with the underlying grammar $G = (O, O_T, w, P, \Longrightarrow_G)$ being of any type X. Then we construct a graph-controlled grammar $G_{GC} = (G, g, H_i, H_f, \Longrightarrow_{GC})$ with the same underlying grammar G. The simulation power is captured by the structure of the control graph $g = (H, E, K)$. The node labels in H, identifying the nodes of the graph in a one-to-one manner, are obtained from G_{AB} as all possible triples of the forms $\left(q, Q, \bar{Q}\right)$ or $\left(\bar{q}, Q, \bar{Q}\right)$ with $q \in L$ and the elements of $Q, \bar{Q}$ being of the form (r, t), $r \in L$ and $t \in \mathbb{N}$ such that t does not exceed the maximum time occurring in the relations in A and B, hence, this in total is a bounded number. We also need a special node labeled $\emptyset$, where a computation in G_{GC} ends in any case when this node is reached.

All nodes can be chosen to be final, i.e., $H_f = H$. $H_i = L_0$ is the set of initial labels, i.e., we start with one of the initial conditions as in the AB-grammar.

The idea behind the node $\left(q, Q, \bar{Q}\right)$ is to describe the situation of a configuration derived in the AB-grammar where q is the label of the rule to be applied and $Q, \bar{Q}$ describe the activated and blocked rules for the further derivation steps in the AB-grammar. Hence, as already in the definition of an AB-grammar, we assume $Q \cap \bar{Q} = \emptyset$.

Now let $g(l)$ denote the rule r assigned to label l, i.e., $(l, r) \in f_L$. Then, the set of rules assigned to $\left(q, Q, \bar{Q}\right)$ is taken to be $\{g(q)\}$. The set of rules assigned to $\emptyset$ is taken to be $\emptyset$. As it will become clear later in the proof why, the nodes $\left(\bar{q}, Q, \bar{Q}\right)$ are assigned the set of rules $\{g(l) \mid (l, 1) \in Q,\ l \neq q\}$; we only take those nodes where this set is not empty.

In node $\left(q, Q, \bar{Q}\right)$, we have to distinguish between two possibilities:

– If $g(q)$ is applicable to the object derived so far, a Y-edge has to go to every node which describes a situation corresponding to what would have been the next configuration in the AB-grammar. We then compute

$$\begin{aligned} \bar{R} &= \left\{(x, i) \mid (x, i+1) \in \bar{Q},\ i > 0\right\} \cup \{(x, i) \mid (q, x, i) \in B\}, \\ R &= (\{(x, i) \mid (x, i+1) \in Q,\ i > 0\} \cup \{(x, i) \mid (q, x, i) \in A\}) \\ &\quad \setminus\ \left\{(x, i) \mid (x, i) \in \bar{R}\right\} \end{aligned}$$

(observe that R and $\bar{R}$ are made non-conflicting) as well as – if it exists – $t_0 := min\{t \mid (x, t) \in R\}$, i.e., the next time step when the derivation in the AB-grammar could continue. Hence, we take a Y-edge to every node $\left(p, P, \bar{P}\right)$ where $p \in \{x \mid (x, t_0) \in R\}$ and

$$\begin{aligned} \bar{P} &= \left\{(x, i) \mid (x, i + t_0 - 1) \in \bar{R},\ i > 0\right\}, \\ P &= \{(x, i) \mid (x, i + t_0 - 1) \in R\}. \end{aligned}$$

If $t_0 := min\{t \mid (x, t) \in R\}$ does not exist, this means that R is empty and we have to make a Y-edge to the node $\emptyset$.

– If $g(q)$ is not applicable to the object derived so far, we first have to check that none of the other rules activated at this step could have been applied, i.e., we check for the applicability of the rules in the set of rules

$$\bar{U} := \{g(l) \mid (l, 1) \in Q,\ l \neq q\}$$

by going to the node $\left(\bar{q}, Q, \bar{Q}\right)$ with a N-edge; from there no Y-edge leaves, as this would indicate the unwanted case of the applicability of one of the rules in $\bar{U}$, but with a N-edge we continue the computation in any node $\left(p, P, \bar{P}\right)$ with p, P, $\bar{P}$ computed as above in the first case. We observe that in case $\bar{R}$ is empty, we can omit the path through the node $\left(\bar{q}, Q, \bar{Q}\right)$ and directly go to the nodes $\left(p, P, \bar{P}\right)$ which are obtained as follows: we first check whether $t_0 := min\{t \mid (x,t) \in Q,\ t > 1\}$ exists or not; if not, then the computation has to end with a N-edge to node $\emptyset$. Otherwise, a N-edge goes to every node $\left(p, P, \bar{P}\right)$ with $p \in \{x \mid (x, t_0) \in Q\}$ and

$$\begin{aligned} \bar{P} &= \left\{(x,i) \mid (x, i + t_0 - 1) \in \bar{Q},\ i > 0\right\}, \\ P &= \left\{(x,i) \mid (x, i + t_0 - 1) \in Q\right\}. \end{aligned}$$

where the simulation may continue.

In this way, every computation in the AB-grammar can be simulated by the graph-controlled grammar with taking a correct path through the control graph and finally ending in node $\emptyset$; due to this fact, we could also choose the node $\emptyset$ to be the only final node, i.e., $H_f = \{\emptyset\}$. On the other hand, if we have made a wrong choice and wanted to apply a rule which is not applicable, although another rule activated at the same moment would have been applicable, we get stuck, but the derivation simulated in this way still is a valid one in the AB-grammar, although in most standard types X, which usually are strictly extended ones, such a derivation does not yield a terminal object. Having taken $H_f = \{\emptyset\}$, such paths would not even lead to successful computations in G_{GC}.

In any case, we conclude that the graph-controlled grammar G_{GC} generates the same language as the AB-grammar G_{AB}. □

4 Special Results for Specific Objects

4.1 Special Results for Arrays

In both the one- and the two-dimensional case, it has been shown, see [4], that even matrix grammars without *ac* are sufficient to generate any recursively enumerable array language, i.e., for $d \in \{1,2\}$, $\mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}MAT\right) = \mathcal{L}\left(d\text{-}ARBA\right)$ (the main reason for such a result is the "#-sensing" ability of the rules of type $d\text{-}\#\text{-}CFA$). Based on Theorem 1, we immediately infer the following result:

Theorem 5. *For $d \in \{1,2\}$,*

$$\mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}A1\right) = \mathcal{L}\left(d\text{-}\#\text{-}CFA\text{-}MAT\right) = \mathcal{L}\left(d\text{-}ARBA\right).$$

4.2 Special Results for Strings

It is well-known, for example see [2], that $\mathcal{L}\left(CF\text{-}RC\right) = \mathcal{L}\left(ARB\right)$. Based on Theorem 3, we immediately infer the following result:

Theorem 6. *$\mathcal{L}\left(CF\text{-}AB\right) = \mathcal{L}\left(CF\text{-}RC\right) = \mathcal{L}\left(ARB\right) = RE$.*

4.3 Special Results for Multisets

As in the case of multisets the structural information contained in the sequence of symbols cannot be used, arbitrary multiset rules are not sufficient for obtaining all sets in $Ps\left(\mathcal{L}\left(ARB\right)\right)$. Yet we can easily show that with AB-grammars we obtain the following:

Theorem 7. *$PsRE = Ps\left(\mathcal{L}\left(ARB\right)\right) = \mathcal{L}\left(mARB\text{-}AB\right)$.*

Proof. It is folklore, for example see [7] and [5], that

$$PsRE = Ps\left(\mathcal{L}\left(ARB\right)\right) = \mathcal{L}\left(mARB\text{-}fC\right) = \mathcal{L}\left(mARB\text{-}RC\right),$$

hence, by Theorem 3, we also obtain $PsRE = \mathcal{L}\left(mARB\text{-}AB\right)$. □

5 Computational Completeness for Context-Free AB-Grammars with Two Non-terminal Symbols

In this section, we state our main results for context-free string and multiset grammars showing that computational completeness can already be obtained with two non-terminal symbols, which result is optimal with respect to the number of non-terminal symbols.

Theorem 8. *Any recursively enumerable set of strings can be generated by a context-free AB-grammar using only two non-terminal symbols.*

Proof. (Sketch) The main technical details of how to use only two non-terminal symbols A and B for generating a given recursively enumerable language follow the construction given in [5] for graph-controlled grammars. The most important to be shown here is how to simulate the ADD- and SUB-instructions of a deterministic register machine with the contents of the two working registers being given by the number of symbols A and B; only at the end, both numbers are zero, whereas in between, during the whole computation, at least one symbol A or B is present. The initial string is A, and one A is also the last symbol to be erased at the end in order to obtain a terminal string.

In the following, we use X to specify one of the two non-terminal symbols A and B, and Y then stands for the other one. For any label p of the register machine we use two labels p and p'. The simulations in the AB-grammar work as follows:

- $p : (ADD(X), q)$ is simulated by $p : X \rightarrow XX$ and $p' : Y \rightarrow YX$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', q, 1)_A$, $(p', q', 2)_A$;
- $p : (SUB(X), q, s)$ is simulated by $p : X \rightarrow \lambda$ and $p' : Y \rightarrow Y$ with $(p, p', 1)_B$ as well as $(p, q, 2)_A$, $(p, q', 3)_A$, and $(p', s, 1)_A$, $(p', s', 2)_A$;

in both cases, the application of the rule labeled by p blocks the rule labeled by p'; in any case, for the next rule labeled r to be simulated, both r and r' are activated, again r' following r one step later.

For the halting label h, only the labeled rule $h : A \to \lambda$ is to be activated. □

This result is optimal with respect to the number of non-terminal symbols: as it has been shown in [3], even for graph-controlled context-free grammars one non-terminal symbol is not enough, hence, the statement immediately follows from Theorem 4.

We now show a similar result for multiset grammars.

Theorem 9. *Any recursively enumerable set of multisets can be generated by an AB-grammar using context-free multiset rules and only two non-terminal symbols.*

Proof. Given a recursively enumerable set of multisets L over the terminal alphabet $T = \{a_1, \ldots, a_k\}$, we can construct a register machine M_L generating L in the following way: instead of speaking of a number n in register r we use the notation ${a_r}^n$, i.e., a configuration of M_L is represented as a string over the alphabet $V = T \cup \{a_{k+1}, a_{k+2}\}$ with the two non-terminal symbols a_{k+1}, a_{k+2}.

We start with one a_{k+1} and first generate an arbitrary multiset over T step by step adding one element a_m from T and at the same time multiply the number of symbols a_{k+1} by p_m, where p_m is the m-th prime number. At the end of this procedure, for the multiset ${a_1}^{n_1} \ldots {a_k}^{n_k}$ we have obtained ${a_m}^{n_m}$ in each register m, $1 \leq m \leq k$, and ${a_{k+1}}^{{p_1}^{n_1} \ldots {p_k}^{n_k}}$ in register $k+1$. As for example, already shown in [8], only using registers $k+1$ and $k+2$, a deterministic register machine M_L' simulating any number of registers by this prime number encoding can compute starting with ${a_{k+1}}^{{p_1}^{n_1} \ldots {p_k}^{n_k}}$ and halt if and only if ${a_1}^{n_1} \ldots {a_k}^{n_k} \in L$. Only with halting, all registers of M_L' are cleared to zero, i.e., we end up with only one a_{k+1} in M_L when this deterministic register machine M_L' has reached its halting label h. So the last step of M_L before halting is just to eliminate this last a_{k+1}. During the whole computation of M_L, the sum of symbols a_{k+1} and a_{k+2} is greater than zero. Hence, it only remains to show how to simulate the instructions of a register machine, which is done in a similar way as in the preceding proof; we use X to specify one of the two non-terminal symbols a_{k+1} and a_{k+2}, and Y then stands for the other one, i.e., $X, Y \in \{a_{k+1}, a_{k+2}\}$. For any label p of the register machine we use two labels p and p'. The simulations in the AB-grammar work as follows:

– a non-deterministic ADD-instruction $p : (ADD(X), q, s)$ is simulated by branching into two deterministic ADD-instructions even twice:
  $p : X \to X$ and $p' : Y \to Y$ with $(p, p', 1)_B$ as well as
  $(p, (p, X, q), 2)_A$, $(p, (p, X, s), 2)_A$, and $(p', (p, Y, q), 1)_A$, $(p', (p, Y, s), 1)_A$;
  in the third step of the simulation, we already know whether X is present or else we have to use Y; this now allows us to simulate the four deterministic ADD-instructions $(p, \alpha, \beta) : (ADD(X), \beta)$, $\alpha \in \{X, Y\}$, $\beta \in \{q, s\}$, in a simpler way by using the rules
  $(p, \alpha, \beta) : \alpha \to \alpha X$

and the activations
$((p,\alpha,\beta),\beta,1)_A$, $((p,\alpha,\beta),\beta',2)_A$;
- $p : (ADD(X), q)$ is simulated by $p : X \to XX$ and $p' : Y \to YX$ with $(p,p',1)_B$ as well as $(p,q,2)_A$, $(p,q',3)_A$, and $(p',q,1)_A$, $(p',q',2)_A$;
- $p : (SUB(X), q, s)$ is simulated by $p : X \to \lambda$ and $p' : Y \to Y$ with $(p,p',1)_B$ as well as $(p,q,2)_A$, $(p,q',3)_A$, and $(p',s,1)_A$, $(p',s',2)_A$;
in both cases, the application of the rule labeled by p blocks the rule labeled by p'; in any case, for the next rule labeled r to be simulated, both r and r' are activated, again r' following r one step later;
- for the halting label h, only the labeled rule $h : a_{r+1} \to \lambda$ is to be activated.

When the final rule $h : a_{r+1} \to \lambda$ is applied, no further rule is activated, thus the derivation ends yielding the multiset ${a_1}^{n_1} \dots {a_k}^{n_k} \in L$ as terminal result. $\square$

6 Conclusion

We have considered the concept of regulating the applicability of rules based on the application of rules in the preceding step(s) within a very general model for sequential grammars and compared the resulting computational power with various other control mechanisms based on the applicability of rules in the underlying grammar, in particular with graph-controlled and matrix grammars as well as random context grammars. Even only using the structural features of the sequences of applied rules, yet not taking into account the features of the underlying objects (e.g., strings, multisets, arrays), general simulation results are obtained. Then we also established some special computational completeness results: for one- and two-dimensional array grammars, only the activation of rules is needed when using #-context-free array rules; for strings and multisets, both activation and blocking of rules were needed when using only context-free rules. For computational completeness for strings or multisets with context-free rules, only two non-terminal symbols are necessary, which is a sharp result, as only one non-terminal symbol is not sufficient.

The concept of activation and blocking of rules can also be used when rules are applied in parallel, which is an attractive idea for the area of P systems where multiple variants of parallel derivation modes are common.

References

1. Cavaliere, M., Freund, R., Oswald, M., Sburlan, D.: Multiset random context grammars, checkers, and transducers. Theor. Comput. Sci. **372**(2–3), 136–151 (2007)
2. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. Springer, Heidelberg (1989)
3. Fernau, H., Freund, R., Oswald, M., Reinhardt, K.: Refining the nonterminal complexity of graph-controlled, programmed, and matrix grammars. J. Autom. Lang. Comb. **12**(1–2), 117–138 (2007)

4. Freund, R.: Control mechanisms on #-context-free array grammars. In: Păun, Gh. (ed.) Mathematical Aspects of Natural and Formal Languages, pp. 97–137. World Scientific Publishing, Singapore (1994)
5. Freund, R., Kogler, M., Oswald, M.: A general framework for regulated rewriting based on the applicability of rules. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life. LNCS, vol. 6610, pp. 35–53. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20000-7_5
6. Freund, R., Oswald, M.: Modelling grammar systems by tissue P systems working in the sequential mode. Fundamenta Informaticae **76**(3), 305–323 (2007)
7. Kudlek, M., Martín-Vide, C., Păun, Gh.: Toward a formal macroset theory. In: Calude, C.S., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) WMC 2000. LNCS, vol. 2235, pp. 123–133. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45523-X_7
8. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall Inc., Upper Saddle River (1967)
9. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press Inc., New York (2010)
10. Rosenfeld, A.: Picture Languages. Academic Press, Reading (1979)
11. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, vol. 1–3. Springer, New York (1997)
12. Wang, P.S.P. (ed.): Array Grammars, Patterns and Recognizers, World Scientific Series in Computer Science, vol. 18. World Scientific Publishing, Singapore (1989)
13. The P Systems Website. http://ppage.psystems.eu/

The Language (and Series) of Hammersley-Type Processes

Cosmin Bonchiş[1,2], Gabriel Istrate[1,2](✉), and Vlad Rochian[1]

[1] Department of Computer Science, West University of Timişoara, Bd. V. Pârvan 4, Timişoara, Romania
gabrielistrate@acm.org

[2] e-Austria Research Institute, Bd. V. Pârvan 4, cam. 045 B, Timişoara, Romania

Abstract. We study languages and formal power series associated to (variants of) the Hammersley process. We show that the ordinary Hammersley process yields a regular language and the Hammersley tree process yields deterministic context-free (but non-regular) languages. For the Hammersley interval process we show that there are *two* relevant variants of formal languages. One of them leads to the same language as the ordinary Hammersley tree process. The other one yields non-context-free languages.

The results are motivated by the problem of studying the analog of the famous Ulam-Hammersley problem for heapable sequences. Towards this goal we also give an algorithm for computing formal power series associated to the Hammersley process. We employ this algorithm to settle the nature of the scaling constant, conjectured in previous work to be the golden ratio. Our results provide experimental support to this conjecture.

1 Introduction

The Physics of Complex Systems and Theoretical Computing have a long and fruitful history of cooperation: for instance the celebrated Ising Model can be studied combinatorially, as some of its versions naturally relate to graph-theoretic concepts [20]. Methods from formal language theory have been employed (even in papers published by physicists, in physics venues) to the analysis of dynamical systems [13,21]. Sometimes the cross-fertilization goes in the opposite direction: concepts from the theory of *interacting particle systems* [12] (e.g. the voter model) have been useful in the analysis of gossiping protocols. A relative of the famous TASEP process, the so-called *Hammersley-Aldous-Diaconis* (HAD) process, has provided [1,2] the most illuminating solution to the famous *Ulam-Hammersley problem* [17] concerning the scaling behavior of the longest increasing subsequence of a random permutation.

This work was supported by a grant of Ministry of Research and Innovation, CNCS - UEFISCDI, project number PN-III-P4-ID-PCE-2016-0842, within PNCDI III.

J. Durand-Lose and S. Verlan (Eds.): MCU 2018, LNCS 10881, pp. 69–87, 2018.
https://doi.org/10.1007/978-3-319-92402-1_4

In this paper we contribute to the literature on investigating physical models with discrete techniques by bringing methods based on formal language theory (and, possibly, noncommutative formal power series) to the analysis of several variants of the HAD process: We define formal languages (and power series) encoding all possible trajectories of such processes, and completely determine (the complexity of) these languages.

The main process we are concerned with was defined in combinatorially in [9], and in more general form in [4], where it was dubbed *the Hammersley tree process*. It appeared naturally in [9] as a tool to investigate a version of the Ulam-Hammersley problem that employs the concept (due to Byers et al. [7]) of *heapable sequence*, an interesting variation on the concept of increasing sequence. Informally, a sequence of integers is heapable if it can be successively inserted into the leaves of a (not necessarily complete) binary tree satisfying the heap property. The Ulam-Hammesley problem for heapable sequences is open, the scaling behavior being the subject of an intriguing conjecture (see Conjecture 19 below) involving the golden ratio [9]. Methods based on formal power series can conceivably rigorously establish the true value of this constant. We also study a (second) version of the Hammersley tree process, motivated by the analogue of the Ulam-Hammersley problem for random intervals [3] (see Conjecture 20 below).

The outline of the paper is the following: In Sect. 2 we precisely specify the systems we are interested in, and outline the results we obtain. In Sect. 3 we discuss the combinatorial and probability-theoretic motivations of the problems we are interested in. This section is not needed to understand the technical details of our proofs. In Sect. 4 we prove our main result: we precisely identify the Hammersley language for every $k \geq 1$. The language turns out to be regular for $k = 1$ and deterministic context-free but non-regular for $k \geq 2$. The result is then extended to (the analog of) the Hammersley process *for intervals*. In this case, it turns that there are *two* natural ways to define the associated formal language. The "effective" version yields the same language as in the case of permutations. The "more useful" one yields (as we show) non-context-free languages that can be explicitly characterized. We then proceed by presenting (Sect. 9) algorithms for computing the power series associated to these systems. They are applied to the problem of determining true value of scaling constant (believed to be equal to the golden ratio) in the Ulam-Hammersley problem for heapable sequences. In a nutshell, **the experimental results tend to confirm the identity of this constant to the golden ratio**; however the convergence is slow, as the estimates based on the formal power series computations we undertake (based on small values of n) seem quite far from the true value. The paper concludes (Sect. 11) with several discussions and open problems.

2 Main Definitions and Results

We are interested in the following variant of the process in [4], totally adequate for the purpose of describing the heapability of random permutations, defined as follows:

Definition 1. *In the process HAD_k, individuals appear at integer times $t \geq 1$. Each individual can be identified with a value $X_t \in \mathbf{R}$, and is initially endowed with k "lives". The appearance of a new individual X_{t+1} subtracts a life from the smallest individual $X_a > X_{t+1}$ (if any) still alive at moment t.*

We can describe combinatorially the evolution of process HAD_k in the following manner: each state of the system at a certain moment n can be encoded by a word of length n over the alphabet $\Sigma_k = \{0, 1, \ldots, k\}$ obtained by discarding the value information from particles and only record the number of lives. Thus particles are arranged in the increasing order of values, from the smallest to the largest.

Example 2. Consider the state s_X of the system HAD_2 after all particles with values $X = [5, 1, 4, 2, 3]$ have arrived (in this order). Then in the state s_X particle 5 has 0 lives, particle 1 has two lives, particle 4 has 0 lives left, particle 2 has two lives, particle 3 has two lives left. Consequently, the word w_X encoding s_X is 22200.

Given this encoding, the dynamics of process HAD_k on random permutations can be described in a completely equivalent manner as a process on words: given word w_k encoding the state of the system at moment k, we choose a random position of w_k, inserting a k there and subtract one from the first nonzero digit to the right of the insertion place, thus obtaining the word w_{k+1}. This first nonzero digit need not be directly adjacent to the insertion place, but separated from it by a block of zeros. These zeros will not be affected in the word w_{k+1}. Figure 1 presents the snapshots of all possible trajectories of system HAD_2 at moments $t = 1, 2, 3$.

Example 3. If we run the process HAD_2 on sequence X from Example 2, the outcome is a multiset of particles $1, 2$ and 3, each with multiplicity 2, encoded by the word 22200.

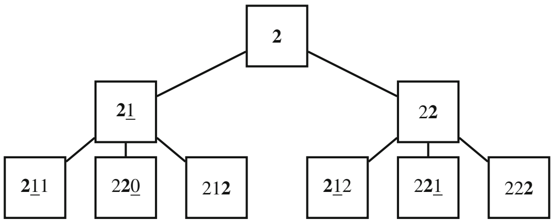


Fig. 1. Words in the Hammersley tree process ($k = 2$). Insertions are boldfaced. Positions that lost a life at the current stage are underlined.

We are interested in the following formal power series that encodes the large-scale evolution of process HAD_k, and the associated formal language:

Definition 4. *Given $k \geq 1$,* the Hammersley power series of order k *is the formal power series $F_k \in \mathbf{N}(< \Sigma_k >)$ defined as follows: given word $w \in \Sigma_k^*$, define $F_k(w)$ to be the multiplicity of word w in the process HAD_k.*

The Hammersley language of order k, *L_H^k, is defined as the support of F_k, i.e. the set of words in Σ_k^* s.t. there exists a* trajectory *of HAD_k that yields w.*

Example 5. $F_2(212) = 2, F_2(220) = 1$, hence $212, 220 \in L_H^2$. On the other hand $200 \notin L_H^2$, since $F_2(200) = 0$.

Definition 6. *For $w \in \Sigma_k^*$ and $a \in \Sigma_k$, denote by $|w|_a$ the number of copies of a in w. Given $k \geq 1$, word $w \in \Sigma_k$ is called k-*dominant *if the following inequality holds* **for every** *$z \in Pref(w)$: $|z|_k - \sum_{i=0}^{k-2}(k-i-1) \cdot |z|_i > 0$. We call the left-hand side term* ***the structural difference*** *of word z.*

Observation 1. *1-dominant words are precisely those that start with a 1. On the other hand, 2-dominant words are those that start with a 2 and have, in any prefix,* strictly *more twos than zeros.*

Our main result completely characterizes the Hammersley language of order k:

Theorem 7. *For every $k \geq 1$, $L_H^k = \{w \in \Sigma_k^* \,|\, w$ is k-dominant$\}$.*

Corollary 8. *Language L_H^1 is regular. For $k \geq 2$ languages L_H^k are deterministic one-counter languages but not regular.*

In [3] we considered the extension of heapability to partial orders, including intervals. We also noted that, just as in the case of random permutations, heapability of random intervals can be analyzed using the following version of the process HAD_k:

Definition 9. *The* interval Hammersley process with k lives *is the stochastic process defined as follows: The process starts with no particles. Particles arrive at integer moments; they have a value in the interval $(0, 1)$, and a number of lives. Given the state Z_{n-1} of the process after step $n-1$, to obtain Z_n we choose, independently, uniformly at random and with repetitions two random reals $X_n, Y_n \in (0, 1)$. Then we perform the following operations:*

- *First a new particle with k lives and value $min(X_n, Y_n)$ is inserted.*
- *Then the smallest (if any) live particle whose value is higher than $max(X_n, Y_n)$ loses one life, yielding state Z_n.*

The state *of the process at a certain moment n comprises a record of all the real numbers chosen along the trajectory:, $(X_0, Y_0, \ldots, X_{n-1}, Y_{n-1})$, even those that do not correspond to a particle. Each number is endowed (in case it represented a new particle) with an integer in the range $0 \ldots k$ representing the number of lives the given particle has left at moment n.*

Just as with process HAD_k, we can combinatorialize the previous definition as follows:

Definition 10. *Process $HAD_{k,INT}$ is the stochastic process on $(\Sigma_k \cup \{\diamond\})^*$ defined as follows: The process starts with the two-letter word $Z_1 = k\diamond$. Given the string representation Z_{n-1} of the process after step $n-1$, we choose, independently, uniformly at random and with repetitions two positions X_n, Y_n into string Z_{n-1}. X_n, Y_n may happen to be the same position, in which we also choose randomly an ordering of X_n, Y_n. Then we perform (see Fig. 2) the following operations:*

- *First, a k is inserted into Z_{n-1} at position $min(X_n, Y_n)$.*
- *Then a $\diamond$ is introduced in position $max(X_n, Y_n)$ (immediately after the newly introduced k, if $X_n = Y_n$).*
- *Then the smallest (if any) nonzero digit occurring* ***after the position of the newly inserted*** *$\diamond$ loses one unit. This yields string Z_n.*

It turns out (see the discussion at the end of Sect. 3) that there are *two* languages meaningfully associated to the process $HAD_{k,INT}$. The first of them has the following definition:

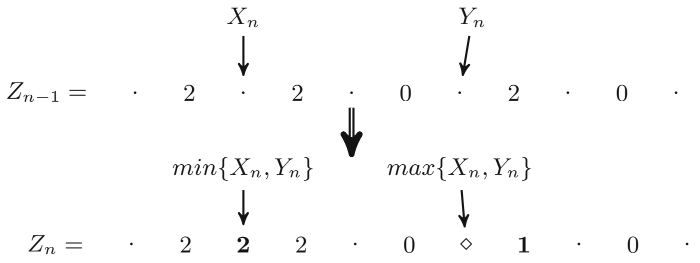

X_n
Y_n
$Z_{n-1} = \cdot\ 2\ \cdot\ 2\ \cdot\ 0\ \cdot\ 2\ \cdot\ 0\ \cdot$
$min\{X_n, Y_n\}$
$max\{X_n, Y_n\}$
$Z_n = \cdot\ 2\ \mathbf{2}\ 2\ \cdot\ 0\ \diamond\ \mathbf{1}\ \cdot\ 0\ \cdot$

Fig. 2. Insertion in process $HAD_{2,INT}$. Insertion positions are marked with a dot. Positions affected by the insertion are in bold.

Definition 11. *Denote by $L^k_{H,INT}$, called* the language of the interval Hammersley process*, the set of words (over alphabet $\Sigma_k \cup \{\diamond\}$) generated by the process $HAD_{k,INT}$.*

The second language associated to the interval Hammersley process is defined as follows:

Definition 12. *The* effective language of the interval Hammersley process*, $L^{k,eff}_{H,INT}$, is the set of strings in Σ_k^* obtained by deleting all diamonds from some string in $L^k_{H,INT}$.*

Despite the fact that the dynamics of process $HAD_{k,INT}$ is quite different from that of the ordinary process HAD_k (a fact that is reflected in the coefficients of the two power series), and the conjectured scaling behavior is not at all similar (for $k \geq 2$), our next result shows that this difference **is not visible** on the actual trajectories: the effective language of the Interval Hammersley process coincides with that of the "ordinary" Hammersley tree process. Indeed, we have:

Theorem 13. *For every $k \geq 1$, $L_{H,INT}^{k,eff} = L_H^k = \{w \in \Sigma_k^* \mid w$ is k-dominant$\}$.*

The previous result contrasts with our next theorem:

Theorem 14. *For $k \geq 1$ the language $L_{H,INT}^k$ is* **not** *context-free.*

In fact we can give a complete characterization of $L_{H,INT}^k$ similar in spirit to the one given for language L_H^k in Theorem 7:

Theorem 15. *Given $k \geq 1$, the language $L_{H,INT}^k$ is the set of words w over alphabet $\Sigma_k \cup \{\diamond\}$ that satisfy the following conditions:*

1. *$|w|_\diamond = |w|/2$. In particular $|w|$ must be even.*
2. *For every prefix p of w, (a) $|p|_\diamond \leq |p|/2$ and (b) $s(p) + (k+1)|p|_\diamond \geq k|p|$.*

Finally, we return to the power series perspective on the Ulam-Hammersley problem for heapable sequences. We outline a simple algorithm (based on dynamic programming) for computing the coefficients of the Hammersley power series F_k.

Theorem 16. *Algorithm ComputeMultiplicity (Fig. 3) correctly computes series F_k.*

We defer the presentation of the application of this result to Sect. 10.

```
Input: $k \geq 1, w \in \Sigma_k^*$
Output: $F_k(w)$
    $S := 0.\ w = w_1 w_2 \ldots w_n$
    if $w \notin L_H^k$
        return 0
    if $w ==$ ‘$k$‘
        return 1
    for $i$ in 1:n-1
        if $w_i == k$ and $w_{i+1} \neq k$
            let $r = min\{l \geq 1 : w_{i+l} \neq 0$ or $i + l = n + 1\}$
            for $j$ in 1:r-1
                let $z = w_1 \ldots w_{i-1} w_{i+1} \ldots w_{i+j-1} 1 w_{i+j+1} \ldots w_{i+r} \ldots w_n$
                $S := S + ComputeMultiplicity(k, z)$
            if $i + r \neq n + 1$ and $w_{i+r} \neq k$
                let $z = w_1 \ldots w_{i-1} w_{i+1} \ldots w_{i+r-1} (w_{i+r} + 1) w_{i+r+1} \ldots w_n$
                $S := S + ComputeMultiplicity(k, z)$
        if $w_n == k$
            let $Z = w_1 \ldots\ldots w_{n-1}$
            $S := S + ComputeMultiplicity(k, z)$
    return S
```

Fig. 3. Algorithm ComputeMultiplicity(k, w)

3 Motivation and Notations

Define $\Sigma_\infty = \cup_{k\geq 1}\Sigma_k$. Given x, y over Σ_∞ we use notation $x \sqsubseteq y$ to denote the fact that x is a prefix of y. The set of (non-empty) prefixes of x is be denoted by $Pref(x)$.

A *k-ary (max)-heap* is a k-ary tree, non necessary complete, whose nodes have labels $t[\cdot]$ respecting the min-heap condition $t[parent(x)] \geq t[x]$. Let $rgeq1$, and let $a_1, b_1, \ldots, a_r > 0$ and $b_r \geq 0$ be integers. We will use notation $[a_0, b_0, \ldots, a_t, b_t]$ as a shorthand for the word $1^{a_0}0^{b_0}\ldots 1^{a_r}0^{b_r} \in \Sigma_1^*$ (where $0^0 = \epsilon$, the null word).

The following combinatorial concept was introduced (for $k = 2$) in [7] and further studied in [3–5,9,10,15]:

Definition 17. *A sequence $X = X_0, \ldots, X_{n-1}$ is max k-heapable if there exists some k-ary tree T with nodes labeled by (exactly one of) the elements of X, such that for every non-root node X_i and parent X_j, $X_j \geq X_i$ and $j < i$. In particular a 2-heapable sequence will simply be called heapable [7]. Min heapability is defined similarly.*

Example 18. $X = [5, 1, 4, 2, 3]$ is max 2-heapable: A max 2-heap is displayed in Fig. 4. On the other $Y = [2, 4, 1, 3]$ is obviously **not** max 2-heapable, as 4 cannot be a descendant of 2.

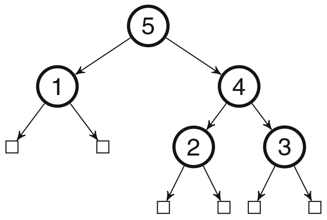


Fig. 4. Heap ordered tree for sequence X in Example 18.

Heapability can be viewed as a relaxation of the notion of decreasing sequence, thus it is natural to attempt to extend to heapable sequences the framework of the Ulam-Hammersley problem [17], concerning the scaling behavior of the longest increasing subsequence (LIS) of a random permutation. This extension can be performed in (at least) two ways, equivalent for LIS but no longer equivalent for heapable sequences: the first way, that of studying the length of the longest heapable subsequence, was dealt with in [7], and is reasonably simple: with high probability the length of the longest heapable subsequence of a random permutation is $n - o(n)$. On the other hand, by Dilworth's theorem [8] the length of the longest increasing subsequence of an arbitrary sequence is equal to the number of classes in a partition of the original sequence into *decreasing* subsequences. Thus it is natural to call *the Hammersley-Ulam problem for heapable sequences* the investigation of the scaling behavior of the number of classes of the partition of a random permutation into a minimal number of (max) heapable subsequences. This was the approach we took in [9]. Unlike the case of LIS, for heapable subsequences the relevant parameter (denoted in [9] by $MHS_k(\pi)$) scales logarithmically, and the following conjecture was proposed:

Conjecture 19. For every $k \geq 2$ there exists $\lambda_k > 0$ s.t., as $n \to \infty$, $\frac{E[MHS_k(\pi)]}{ln(n)}$ converges to λ_k. Moreover $\lambda_2 = \frac{1+\sqrt{5}}{2}$ **is the golden ratio.**

The problem was further investigated in [4,5], where the existence of the constant λ_k was proved. The equality of λ_2 to the golden ratio is less clear: authors of [4] claim it is slightly less than ϕ. Some non-rigorous, "physics-like" arguments, in favor of the identity $\lambda_2 = \phi$ was already outlined in [9], and is presented in [10], together with experimental evidence. Here we bring more convincing such evidence.

The intuition for Conjecture 19 relies on the extension from the LIS problem to heapable sequences of a correspondence between LIS and an interactive particle system [1] called the *Hammersley-Aldous-Diaconis (shortly, Hammersley or HAD) process.* The validity of correspondence was noted, for heapable sequences, in [9]. The generalized process was further investigated in [4], where it was called *the Hammersley tree process.*

To recover the connection with random permutations we will assume from now on that the X_i's in process HAD_k are independent random numbers in $(0, 1)$. The proposed value for λ_2 arises from a conjectural identification of the "hydrodynamic limit" of the Hammersley tree process (in the form of a compound Poisson process).

As $n \to \infty$ a "typical" sample word from the Hammersley process HAD_2 will have approximately $c_0 n$ zeros, $c_1 n$ ones and $\sim c_2 n$ twos, for some constants[1] $c_0, c_1, c_2 > 0$. Moreover, conditional on the number of zeros, ones, twos, in a typical word these digits are "uniformly mixed" throughout the sequence. Experimental evidence presented in [10] seems to confirm the accuracy of this heuristic description.

A proof of the existence of constants c_0, c_1, c_2 was attempted in [9] based on subadditivity (Fekete's lemma). However, part of the proof in [9] is incorrect. While it could perhaps be fixed using more sophisticated tools (e.g. the subadditive ergodic theorem [19]) than those in [9], an alternate approach involves analyzing the asymptotic behavior of process HAD_k using (noncommutative) power series ([6,18]).

Understanding and controlling the behavior of formal power series F_k may be the key to obtaining a rigorous analysis that confirms the picture sketched above. Though that we would very much want to accomplish this task, in this paper we resign ourselves to a simpler, language-theoretic, version of this problem, that of computing the associated formal language.

[1] Nonrigorous computations predict that $c_0 = c_2 = \frac{\sqrt{5}-1}{2}, c_1 = \frac{3+\sqrt{5}}{2}$..

The Ulam-Hammersley problem has also been studied [11] for sets of random intervals, generated as follows: to generate a new interval I_n first we sample (independently and uniformly) two random x, y from $(0, 1)$. Then we let I_n be the interval $[min(x, y), max(x, y)]$. In fact the problem was settled in [11], where the scaling of LIS for sets of random intervals was determined to be $\lim_{n\to\infty} \frac{E[LIS(I_1,\ldots,I_n)]}{\sqrt{n}} = \frac{2}{\sqrt{\pi}}$.

Several results on the heapability of partial orders were proved in [3]; in particular, the greedy algorithm for partitioning a permutation into a minimal number of heapable subsequences extends to interval orders. This justifies an extension of the Ulam-Hammersley problem from increasing to heapable sequences of intervals. Indeed, in [3] we conjectured the following scaling law:

Conjecture 20. For every $k \geq 2$ there exists $c_k > 0$ such that, if R_n is a sequence of n random intervals then $\lim_{n\to\infty} \frac{E[\#Heaps_k(R_n)]}{n} = c_k$. Moreover $c_k = \frac{1}{k+1}$.

Remarkably, it was already noted in [3] that the connection between the Ulam-Hammersley problem and particle systems extends to the interval setting as well. To prove a similar result for the interval Hammersley process we need to "combinatorialize" the process from Definition 9, that is, to replace that definition (which employs (random) real values in $(0, 1)$) with an equivalent stochastic process on words.

The combinatorialization process has some technical complications with respect to the case of permutations. Specifically, for permutations the state of the system could be preserved, with no real loss of information by a string representing only the number of lifelines of the given particles, but **not** their actual values. This enables (as we will see below in Sect. 9) an algorithm for computing the associated formal power series.

To accomplish a similar goal for random intervals we apparently need to take into account the fact that at each step we choose *two* random numbers in Definition 9, even though only one of them receives a particle, since the second one influences the state of the system. Thus, the proper discretization requires an extra symbol $\diamond$ (that marks the positions of real values that were generated but in which no particle was inserted), and is accomplished as described in Definition 10 and the language from Definition 11.

A result that was easy for the process HAD_k but deserves some discussion in the case of the interval process is the following:

Proposition 21. *Consider the string $w_n \in \Sigma_k^*$ obtained by taking a random state of the Hammersley interval process with k lifelines at stage n and then "forgetting" the particle value information (recording instead only the value in $\Sigma_k \cup \{\diamond\}$). Then w_n has the same distribution as a sample from process $HAD_{k,INT}$ at stage n.*

Proof. The crux of the proof is the following

Lemma 22. *The ordering of the values $X_0, Y_0, X_1, Y_1, \ldots, X_{n-1}Y_{n-1}$ inserted in the first n steps in the Hammersley interval process (disregarding their number of lifelines) is that of a random permutation with $2n$ elements.*

Proof. X_i, Y_i have the same distribution, both are random uniformly distributed variables in $(0, 1)$. Thus to simulate $HAD_{k,INT}$ for n steps one needs $2n$ random numbers in (0,1), which yields a random permutation of size $2n$.

This discussion motivates the language-theoretic study of trajectories of the interval Hammersley process $HAD_{k,INT}$ as well. In that respect Definition 12 seems better motivated than Definition 11. Indeed, due to the presence of diamonds, words in the Definition 12 are not "physical", as diamonds do not necessarily correspond to actual particles. On the other hand one can easily obtain an algorithm (similar to the ComputeMultiplicity algorithm presented above) that computes multiplicities for "extended words" in the process $HAD_{k,INT}$ such as those in the Definition 11. Hence the study of this second language is motivated on pragmatic grounds, as a first step to investigating $F_{k,INT}$, the formal power series of multiplicities in the interval Hammersley process. We defer this investigation to the journal version of the paper.

4 Proof of the Main Result

The proof of Theorem 7 proceeds by double inclusion. Inclusion "$\subseteq$" is proved with the help of several easy auxiliary results:

Lemma 23. *Every word in L_H^k starts with a k.*

Proof. Follows easily by appealing to the particle view of the Hammersley process: the particle with the smallest label x stays with k lives until the end of the process, as no other particle can arrive to its left.

Lemma 24. *L_H^k is closed under prefix.*

Proof. Again we resort to the particle view of the Hammersley process: let $w \in L_H^k$ be a word and $u = x_0 \ldots x_{n-1}$ be a trajectory in [0,1] yielding w. A nonempty prefix z of w corresponds to the restriction of u to some segment $[0, l]$, $0 < l < 1$. This restriction is a trajectory itself, that yields z.

Lemma 25. *Every word in L_H^k has a positive structural difference.*

Proof. Let $w \in L_H^k$ and let t be a corresponding trajectory in the particle process.

Let λ be the number of times a particle arrives as a local maximum (without subtracting a lifeline from anyone). For $i = 1, \ldots, k$ let λ_i be the number of time the newly arrived particle subtracts a lifeline from a particle currently holding exactly i lives. $\lambda, \lambda_1, \ldots, \lambda_k \geq 0$. Moreover, $\lambda > 0$, since the largest particle does not take any lifeline.

By counting the number of particles with i lives at the end of the process, we infer: $|z|_0 = \lambda_1, |z|_1 = \lambda_2 - \lambda_1, \cdots |z|_{k-1} = \lambda_k - \lambda_{k-1}$. Finally, $|z|_k = \lambda + \sum_{i=0}^{i-2} \lambda_i.(*)$

Simple computations yield $\lambda_{i+1} = |z|_0 + \ldots + |z|_i$, for $i = 0, \ldots, k-1$. Relation (*) and inequality $\lambda > 0$ yield the desired result.

Together, Claims 23, 24 and 25 establish the fact that any word from L_H^k is k-dominant, thus proving inclusion "$\subseteq$". To proceed with the opposite inclusion, for every k-dominant word w we must construct a trajectory of the process HAD_k that acts as a witness for $w \in L_H^k$.

We will further reduce the problem of constructing a trajectory T_z to the case when z further satisfies a certain simple property, explained below:

Definition 26. *k-dominant word u is called critical if* $|u|_k - \sum_{i=0}^{k-2}(k-1-i)\cdot|u|_i = 1$.

The above-mentioned reduction has the following statement:

Lemma 27. *Every k-dominant* **critical** *z is witnessed by some trajectory T_z.*

Proof. By induction on $|z|$. The base case, $|z| = 1$, is trivial, as in this case $z = k$.

Inductive step: Assume the claim is true for all the critical words of length strictly smaller than z's. We claim that w_1, the word obtained from z by deleting the last copy of k and increasing by 1 the value of the letter immediately to the right of the deleted letter, is critical.

Indeed, it is easy to see that the structural difference of w_1 is 1. Clearly the deleted letter could not have been the last one, otherwise deleting it would yield a prefix of z that has structural constant equal to zero. Also clearly, the letter whose value was modified in the previous constraint could not have been a k, by definition, and certainly is nonzero after modification. So w_1's construction is indeed correct. As $|w_1| = |z| - 1$, w_1 satisfies the conditions of the induction hypothesis.

By the induction hypothesis, w_1 can be witnessed by some trajectory T. We can construct a trajectory for z by simply following T and then inserting the last k of z into w_1 in its proper position (thus also making the next letter assume the correct value).

We now derive Theorem 7 from Lemma 27. The key observation is the following fact: every k-dominant word z is a prefix of a *critical word*, e.g. $z' = z(k-2)^{\lambda}$ where $\lambda = |z|_k - \sum_{i=0}^{k-2}(k-i-1)\cdot|z|_i - 1 \geq 0$.

By Lemma 27, z' has a witnessing trajectory $T_{z'}$. Since the existence of a trajectory is closed under taking prefixes, Theorem 7 follows.

5 Proof of Corollary 8

Proof. For $k = 1$ the result is trivial, as $L_H^1 = 1\Sigma_1^*$. The claim that L_H^k is a deterministic one-counter language for $k \geq 2$ follows from Theorem 7, as one can construct a one-counter pushdown automaton P_k for the language on k-dominant words.

The one-counter PDA has input alphabet $0, 1, 2, \ldots, k$. Its stack alphabet contains two special stack symbols, the bottom symbol Z and another "counting" symbol $*$. The transitions of P_k are informally defined as follows:

- P_k starts with the stack consisting of the symbol Z. If the first letter is not a k, P_k immediately rejects. Otherwise it pushes a $*$ on the stack.
- on reading any subsequent k, P_k pushes a $*$ symbol on stack.
- on reading any symbol $i \in 1 \ldots k-2$, P_k attempts to pop $k-i-1$ stars from the stack. If this ever becomes impossible (by reaching Z), P_k immediately rejects.
- P_k ignores all $k-1$ symbols, proceeding without changing the content of the stack.
- If, while reaching the end of the word, the stack still contains a star, P_k accepts.

To prove that L_H^k, $k \geq 2$, is not regular is a simple exercise in formal languages. It involves applying the pumping lemma for regular languages to words $w_{k,n} = k^{n(k-1)+1}0^n \in L_H^k$. We infer that for large enough n, $w_{k,n} = w_1w_2w_3$, with w_2 nonempty and consisting of k's only, such that for every $l \geq 0$, $w_1w_2^lw_3 \in L_H^k$. We obtain a contradiction by letting $l = 0$, thus obtaining a word z that cannot belong to L_H^k, since $|z|_k \leq (k-1)|z|_0$. □

6 Proof of Theorem 13

It is immediate that $L_H^k \subseteq L_{H,INT}^{k,\text{eff}}$. Indeed, every trajectory of the process HAD_k is a trajectory of the process $HAD_{k,INT}$ as well: simply restrict at every stage the two particles to choose the same slot.

For the opposite inclusion we prove, by induction on $|t|$, that the outcome w of every trajectory t of the interval Hammersley process belongs to L_H^k. The case $|t| = 0$ is trivial, since $w = k$.

Definition 28. *Given a word w over Σ_k, word z is a* left translate of w *if z can be obtained from z by moving a k in w towards the beginning of w (we allow "empty moves", i.e. $z = w$).*

Lemma 29. *L_H^k is closed under left translates. That is, if $w \in L_H^k$ and z is a left translate of w then $z \in L_H^k$.*

Proof. By moving forward a k the structural constants of all prefixes of w can only increase. Thus if these constants are positive for all prefixes of w then they are positive for all prefixes of z as well.

Now assume that the induction hypothesis is true for all trajectories of length less than n. Let t be a trajectory of length n, let t' be its prefix of length $n-1$, let w be the yield of t and z be the yield of t'. By the induction hypothesis $z \in L_H^k$. Let y be the word obtained by applying the Hammersley process to z, deleting a life from the same particle as the interval Hammersley process does to z to obtain w. It is immediate that w is a left translate of y (that is because in the interval Hammersley process we insert a particle to the left of the position where we would in HAD_k). Since $y \in L_H^k$, by the previous lemma $w \in L_H^k$.

7 Proof of Theorem 14

Define the language $S_k = L^k_{H,INT} \cap \{k\}^* \diamond^* \{k-1\}^* \diamond^*$.

Lemma 30. $S_k = \{k^{c+d+e} \diamond^{c+e} (k-1)^c \diamond^{c+d} | c, d, e \geq 0\}$.

Proof. The direct inclusion is fairly simple: let $w \in S_k$. define c to be the number of letters $(k-1)$ in w. Since there are no diamonds in between the $(k-1)$'s, all such letters must have been produced by removing one lifeline each by some k's. Thus the number of stars in between the k's and $(k-1)$'s is $c+e$, with e being the number of pairs $(k, \diamond)$ that did not kill any particle that will eventually become a $k-1$.

On the other hand the number of k's is obtained by tallying up c (for the c letters that become $k-1$, needing one copy of k each), e (for the pairs $(k, \diamond)$ where $\diamond$ belongs to the first set of diamonds) and d (for d pairs $(k, \diamond)$ with $\diamond$ in the second set of diamonds).

For the reverse implication we outline the following construction:
First we derive $k^e \diamond^e$. Then we repeat the following strategy c times:

- We insert a k at the beginning of the $k-1$ block (initially at the end of the word) and the corresponding $\diamond$ at the end of the word.
- With one pair $k, \diamond$ (with $\diamond$ inserted in the first block) we turn the k into a $k-1$.

Finally we insert k pairs $(k, \diamond)$, with $\diamond$ in the second block.

The theorem now follows from the following

Lemma 31. *S_k is not a context-free language.*

Proof. An easy application of Ogden's lemma: We take a string $s \in S_k$,

$$s = k^{c+d+e} \diamond^{c+e} (k-2)^c \diamond^{c+d}$$

with $c, d, e \geq p$ (where p is the parameter in Ogden's Lemma. We mark all positions of $k-1$. Then $s = uvwxy$, with $uv^iwx^iy \in S_k$ for all $i \geq 0$. The "pumping blocks" v, x cannot consist of more than one type of symbols, otherwise the pumped strings would fail to be a member of $\{k\}^* \diamond^* \{k-1\}^* \diamond^*$.

Therefore no more than two blocks (of the four in s) get pumped. One that definitely gets pumped is the first block of diamonds. Taking large enough i we obtain a contradiction, since the block that fails to get pumped will eventually have smaller length than the (pumped) first block of diamonds.

8 Proof of Theorem 15

Proof. The inclusion $\subseteq$ is easy: given $w \in L^k_{H,INT}$, conditions 1. and 2(a) hold, as the process HAD^k_{INT} inserts a digit (more precisely a k) before every diamond.

As for condition 2(b), each $\diamond$ takes at most one life of a particle. The total number of lives particles in p are endowed with at their moments of birth is $k(|p| - |p|_{\diamond})$. These lives are either preserved (and are counted by $s(p)$), or they are lost, in a move which (also) introduces a $\diamond$ in p. Thus $k(|p|-|p|_{\diamond}) \leq |p|_{\diamond}+s(p)$, which is equivalent to b.

The inclusion $\supseteq$ is proved by induction on $|w|$. What we have to prove is that every word that satisfies conditions 1–2 is an output of the process $HAD_{k,INT}$.

The case $|w| = 2$ is easy: the only word that satisfies conditions 1–2 is easily seen to be $w = k\diamond$, which can be generated in one move.

Assume now that the induction hypothesis is true for all words of lengths strictly less than 2n, and let $w = w_1 \ldots w_{2n}$ be a word of length $2n$ satisfying conditions 1–2.

Lemma 32. $w_{2n} = \diamond$.

Proof. Let $p = w_1 \ldots w_{2n-1}$. By condition 2(a), $|p|_{\diamond} \leq (2n-1)/2$, hence $|p|_{\diamond} \leq n-1$. Since $|w|_{\diamond} = n$, the claim follows.

Lemma 33. $w_1 = k$.

Proof. Let $q = w_1$. Since $|q|_{\diamond} \leq 1/2$, w_1 must be a digit. Since $s(q) \geq k|q| = k$, the claim follows.

Let now r be the largest index such that $w_r = k$. Let s be the leftmost position $s > r$ such that $w_s = \diamond$. Let t be the leftmost position $t > s$ such that $w_t \neq \diamond$, $t = 2n+1$ if no such position exists.

Consider the word b obtained from w by a. deleting positions w_r and w_s. b. increasing the digit at position w_t by one, if $t \neq 2n+1$. Note that, if $t \neq 2n+1$ then $w_t \neq k$, by the definition of index r. Also, $|b| = 2n-2 < 2n$.

w is easily obtained from b by inserting a k in position r and a diamond in position s, also deleting one lifeline from position t if $t \neq 2n+1$. To complete the proof we need to argue that b satisfies conditions 1–2(a), (b). Then, by induction, b is an output of the process $HAD_{k,INT}$, hence so is w.

Condition 1 is easy to check, since $|b| = 2n-2$, and b has exactly one $\diamond$ less than w, i.e. n-1 $\diamond$'s. As for 2(a)–(b), let p be a prefix of b. There are four cases:

- **Case 1:** $1 \leq |p| < r$: In this case p is also a prefix of w, and the result follows from the inductive hypothesis.
- **Case 2:** $r \leq |p| < s-1$: In this case $p = w_1 \ldots w_{r-1}w_{r+1} \ldots w_{|p|+1}$. Let $z_1 = w_1 \ldots\ldots w_{|p|+1}$ be the corresponding prefix of w.
  The number of diamonds in p is equal to the number of diamonds in z_1. Since z_1 does **not** end with a diamond (as $|p| < s-1$), the number of diamonds in z_1 is equal to that of its prefix u of length $|p|$. By the induction hypothesis $|p|_{\diamond} = |u|_{\diamond} \leq |u|/2 = |p|/2$. So condition 2(a) holds. On the other hand $s(p)+(k+1)|p|_{\diamond} = (s(z_1)-k)+(k+1)|z_1|_{\diamond} \geq k|z_1|-k = k|p|$, so 2(b) holds as well.
- **Case 3:** $s-1 \leq |p| < t-2$: Thus $p = w_1 \ldots w_{r-1}w_{r+1} \ldots w_{s-1}w_{s+1} \ldots w_{|p|+2}$. Let $z_2 = w_1 \ldots\ldots w_{|p|+2}$ be the corresponding prefix of w of length $|p|+2$ and z_3 the prefix of w of length $s-1$.

The number of diamonds in p is equal to the number of diamonds in z_2 minus one. By the induction hypothesis, this is at most $|z_2|/2 - 1$, which is at most $(|p| + 2)/2 - 1 = |p|/2$. Thus condition 2(a) holds. Now $s(p) + (k+1)|p|_{\diamond} =$

$$(s(z_2) - k) + (k+1)(|z_2|_{\diamond} - 1) = (s(z_3) - k) + (k+1)(|z_3|_{\diamond} + |z_2| - |z_3| - 1)$$
$$\geq k|z_3| - k + (k+1)(|p| + 2 - |z_3| - 1) = (k+1)(|p|+1) - |z_3| - k$$
$$= (k+1)|p| - |z_3| + 1 > k|p| + 1 + (|p| - |z_3|) > k|p|$$

so condition 2(b) is established as well. In the previous chain of (in)equalities we used the fact (valid by the very definition of t) that for all $s \leq i < t$, $w_i = \diamond$.

- **Case 4:** $t - 2 \leq |p| \leq 2n$:] In this case $p = w_1 \ldots w_{r-1} w_{r+1} \ldots w_{s-1} w_{s+1} \ldots (w_t + 1) \ldots$. Furthermore, p ends with $w_{|p|+2}$ (if $|p|+2 \neq t$) and with $w_{|p|+2}+1$ (if $|p| + 2 = t$). Let $z_4 = w_1 \ldots \ldots w_{|p|+2}$ be the prefix of w of length $|p| + 2$.
  - $|p|_{\diamond} = |z_4| - 1 \leq |z_4|/2 - 1 = (|p| + 2)/2 - 1 = |p|/2$.
  - On the other hand $s(p) + (k+1)|p|_{\diamond} = (s(z_4)) - k + 1) + (k+1)(|z_4| - 1) \geq k|z_4| - k + 1 - k - 1 = k \cdot (|p| + 2) - 2k = k|p|$.

  so conditions 2(a)–(b) are proved in this last case as well.

9 Proof of Theorem 16

Justifying correctness of algorithm ComputeMultiplicity is simple: a string w can result from any string z by inserting a k and deleting one life from the closest non-zero letter of z to its right. After insertion, the new k will be the rightmost element of a maximal block of w of consecutive k's. The letter it acts upon in z cannot be a k (in w), and cannot have any letters other than zero before it.

The candidates in w for the changed letter are those letters l succeeding the newly inserted k such that $0 \leq l \leq k - 1$ and the only values between k and l are zeros. Thus these candidates are the following: (a) letters in w forming the maximal block B of zeros immediately following k (if any), and (b) The first letter after B, provided it has value 0 to $k-1$. Since we are counting multiplicities and all these words lead to distinct candidates, the correctness of the algorithm follows.

For $k = 1$ the algorithm ComputeMultiplicity simplifies to a recurrence formula: Indeed, in this case there are no candidates of type (b) We derive:

$$F_1([a_1, b_1, \ldots, a_s, b_s]) = \sum_{\substack{i=1: \\ a_i > 1}}^{s} \sum_{\substack{j,l \geq 0 \\ j+1+l=b_i}} F_1([a_1, \ldots, a_i - 1, j, 1, l, a_{i+1}, \ldots, b_s])$$
$$+ \sum_{\substack{i=1: \\ a_i = 1}}^{s} \sum_{\substack{j,l \geq 0 \\ j+1+l=b_i}} F_1([a_1, \ldots, a_{i-1}, b_{i-1} + j, 1, l, a_{i+1}, \ldots, b_s]) \text{ if } b_s > 0, \text{ otherwise}$$
$$F_1([a_1, \ldots, a_s, 0]) = \sum_{\substack{i=1: \\ a_i > 1}}^{s-1} \sum_{\substack{j,l \geq 0 \\ j+1+l=b_i}} F_1([a_1, \ldots, a_i - 1, j, 1, l, a_{i+1}, \ldots, a_s, 0])$$
$$+ \sum_{\substack{i=1: \\ a_i = 1}}^{s-1} \sum_{\substack{j,l \geq 0 \\ j+1+l=b_i}} F_1([a_1, \ldots, a_{i-1}, b_{i-1} + j, 1, l, a_{i+1}, \ldots, b_s]) + F_1([a_1, \ldots, a_s - 1, 0]).$$

In spite of this, we weren't able to solve the recurrence above and compute the generating functions F_1 or, more generally, F_k, for $k \geq 1$. An inspection of the coefficients obtained by the application of the algorithm is inconclusive: We

w	1	10	11	100	101
$F_1(w)$	1	1	1	1	2
w	110	111	1000	1001	1010
$F_1(w)$	2	1	1	3	5
w	1011	1100	1101	1110	1111
$F_1(w)$	3	5	3	3	1

w	2	21	22	211	212	220	221
$F_2(w)$	1	1	1	1	2	1	1
w	222	2111	2112	2120	2121	2122	2201
$F_2(w)$	1	1	3	2	3	3	1
w	2202	2210	2211	2212	2220	2221	2222
$F_2(w)$	3	1	1	2	2	1	1

Fig. 5. The leading coefficients of formal power series (a) F_1. (b) F_2.

tabulated the leading coefficients of series F_1 and F_2, computed using the Algorithm 3 in Figs. 5(a) and (b) The second listing is restricted to 2-dominant strings only. No apparent closed-form formula for the coefficients of F_1, F_2 emerges by inspecting these values.

10 Application: Estimating the Value of the Scaling Constant λ_2

The computation of series F_2 allows us to tabulate (for small value of n) the values of the distribution of increments, a structural parameter whose limiting behavior determines the value of the constant λ_2 (conjectured, remember, to be equal to $\frac{1+\sqrt{5}}{2}$).

Definition 34. *Let w be a word that is an outcome of the process HAD_k. An* increment of w *is a position p in w (among the $|w|+1$ possible positions: at the beginning of w, at the end of w or between two letters of w) such that no nonzero letters of w appear to the right of p. The number of increments of word w is denoted by $\#inc_k(w)$. It is nothing but 1 plus the number of trailing zeros of w.*

Let L be an alphabet that contains Σ_k for some $k \geq 1$. Given a word $w \in L^*$ we denote by $s(w)$ the sum of the digit characters of w.

The fact that increments are useful in computing λ_2 is seen as follows: consider a word w of length n that is a sample from the HAD_k process. Increments of w are those positions where the insertion of a k does not remove any lifeline, thus increasing the number of heaps in the corresponding greedy "patience heaping" algorithm [9] by 1. If w has t increment positions then the probability that the number of heaps will increase by one (given that the current state of the process is w) is $t/(n+1)$.

What we need to show is that (as $n \to \infty$) the mean number of positions that are increments in a random sample w of length n tends to λ_k. Therefore the probability that a new position will increase the number of heaps by 1 is asymptotically equal to $\lambda_k/(n+1)$. The scaling of the expected number of heaps follows from this limit.

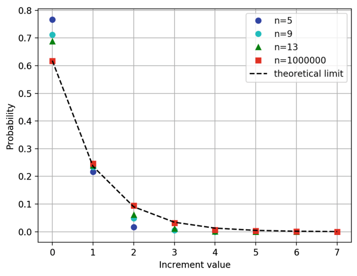


Fig. 6. Probability distribution of increments, for $k = 2$, and $n = 5, 9, 13, 1000000$.

n	2	3	4	5	6	7
$E[\#inc_2]$	1.0	1.166	1.208	1.250	1.281	1.307
n	8	9	10	100	100000	1 mil
$E[\#inc_2]$	1.329	1.347	1.363	1.520	1.575	1.580

Fig. 7. The mean values of the distributions of increments.

In Fig. 6, we plot the *exact* probability distribution of the number of increments (from which we subtract one, to make the distribution start from zero) for $k = 2$ and several small values of n. They were computed exactly by employing Algorithm 3 to exactly compute the probability of each string w, and then computing $\#inc_2(w)$. We performed this computation for $2 \leq n \leq 13$. The corresponding expected values are tabulated (for all values $n = 2, \ldots, 10$) in Fig. 7.

Unfortunately, as it turns out, the ability to exactly compute (for small values of n) the distribution of increments **does not** give an accurate estimate of the asymptotic behavior of this distribution, as the convergence seems rather slow, and not at all captured by these small values of n. Indeed, to explore the distribution of increments for large values of n, as exact computation is no longer possible, we instead resorted to *sampling* from the distribution, by generating 10000 independent random trajectories of length n from process HAD_2, and then computing the distribution of increments of the sampled outcome strings. The outcome is presented (for $n = 100, 100000, 1000000$, together with some of the cases of the exact distribution) in Fig. 6. The distribution of increments seems to converge (as $n \to \infty$) to a geometric distribution with parameter $p = \frac{\sqrt{5}-1}{2} \sim 0.618\cdots$. That is, we predict that for all $i \geq 1$, $\lim_{n\to\infty} Pr_{|w|=n}[\#inc_2(w) = i] = p \cdot (1-p)^{i-1}$. The fit between the (sampled) estimates for $n = 1000000$ and the predicted limit distribution is quite good: every coefficient differs from its predicted value by no more than 0.003, with the exception of the fourth coefficient, whose difference is 0.007. Because of the formula for computing averages, these small differences have, though, a cumulative effect in the discrepancy for the average $E[\#inc_2(w)]$ for $n = 10000000$ accounting for the 0.03 difference between the sampled value and the predicted limit: in fact most of the difference is due to the fourth coefficient, as $4 \times 0.007 = 0.028$.

Conclusion 1. *The increment data supports the conjectured value* $\lambda_2 = 1+p = \frac{1+\sqrt{5}}{2}$.

We intend to present (in the journal version of this paper) a similar investigation of the value of constant c_k in Conjecture 20.

11 Open Questions and Future Work

The major open problems raised by our work concerns the nature and asymptotic behavior of formal power series F_k, $F_{k,INT}$. An easy consequence of Corollary 8 is

Corollary 35. *For* $k \geq 2$ *formal power series* F_k, $F_{k,INT}$ *are* **not** $\mathbf{N}$*-rational.*

Open Problem 1. *Are formal power series* F_1, $F_{1,INT}$ $\mathbf{N}$*-rational? (We conjecture that the answer is negative).*

Note that Reutenauer [16] extended the Chomsky-Schützenberger criterion for rationality from formal languages to power series: a formal power series is rational if and only if the so-called *syntactic algebra* associated to it has finite rank. We don't know, though, how to explicitly apply this result to the formal power series we investigate in this paper. On the other hand, in the general case, the characterization of context-free languages as supports of $\mathbf{N}$-algebraic series (e.g. Theorem 5 in [14]), together with Theorem 14, establishes the fact that series $F_{k,INT}$ is not $\mathbf{N}$-algebraic.

Open Problem 2. *Are series* F_k $\mathbf{N}$*-algebraic? (Conjecture: the answer is negative).*

References

1. Aldous, D., Diaconis, P.: Hammersley's interacting particle process and longest increasing subsequences. Probab. Theor. Relat. Fields **103**(2), 199–213 (1995)
2. Aldous, D., Diaconis, P.: Longest increasing subsequences: from patience sorting to the Baik-Deift-Johansson theorem. Bull. A.M.S. **36**(4), 413–432 (1999)
3. Balogh, J., Bonchiş, C., Diniş, D., Istrate, G., Todincă, I.: Heapability of partial orders. arXiv preprint arXiv:1706.01230 (2017)
4. Basdevant, A.-L., Gerin, L., Gouéré, J.-B., Singh, A.: From Hammersley's lines to Hammersley's trees. Prob. Theory Related Fields **171**(1–2), 1–51 (2018). https://doi.org/10.1007/s00440-017-0772-2
5. Basdevant, A.-L., Singh, A.: Almost-sure asymptotic for the number of heaps inside a random sequence. arXiv preprint arXiv:1702.06444 (2017)
6. Berstel, J., Reutenauer, C.: Noncommutative Rational Series with Applications, vol. 137. Cambridge University Press, Cambridge (2011)
7. Byers, J., Heeringa, B., Mitzenmacher, M., Zervas, G.: Heapable sequences and subseqeuences. In: Proceedings of ANALCO 2011, pp. 33–44. SIAM Press (2011)
8. Dilworth, R.P.: A decomposition theorem for partially ordered sets. Ann. Math. **51**, 161–166 (1950)

9. Istrate, G., Bonchiş, C.: Partition into heapable sequences, heap tableaux and a multiset extension of Hammersley's process. In: Cicalese, F., Porat, E., Vaccaro, U. (eds.) CPM 2015. LNCS, vol. 9133, pp. 261–271. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19929-0_22
10. Istrate, G., Bonchiş, C.: Heapability, interactive particle systems, partial orders: results and open problems. In: Câmpeanu, C., Manea, F., Shallit, J. (eds.) DCFS 2016. LNCS, vol. 9777, pp. 18–28. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41114-9_2
11. Justicz, J., Scheinerman, E.R., Winkler, P.M.: Random intervals. Am. Math. Mon. **97**(10), 881–889 (1990)
12. Liggett, T.: Interacting Particle Systems. Springer, Heidelberg (2005). https://doi.org/10.1007/b138374
13. Moore, C., Lakdawala, P.: Queues, stacks and transcendentality at the transition to chaos. Physica D **135**(1–2), 24–40 (2000)
14. Petre, I., Salomaa, A.: Algebraic systems and pushdown automata. In: Droste, M., Kuich, W., Vogler, H. (eds.) Handbook of Weighted Automata, pp. 257–289. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01492-5_7
15. Porfilio, J.: A combinatorial characterization of heapability. Master's thesis, Williams College, May 2015. https://unbound.williams.edu/theses/islandora/object/studenttheses%3A907. Accessed Dec 2017
16. Reutenauer, C.: Séries formelles et algebres syntactiques. J. Algebra **66**(2), 448–483 (1980)
17. Romik, D.: The Surprising Mathematics of Longest Increasing Subsequences. Cambridge University Press, Cambridge (2015)
18. Salomaa, A., Soittola, M.: Automata-Theoretic Aspects of Formal Power Series. Springer, New York (1978). https://doi.org/10.1007/978-1-4612-6264-0
19. Szpankowski, W.: Average Case of Algorithms on Sequences. Wiley, New York (2001)
20. Welsh, D.: Complexity: Knots, Colourings and Counting. Cambridge University Press, Cambridge (1994)
21. Xie, H.: Grammatical Complexity and One-Dimensional Dynamical Systems. Directions in Chaos, vol. 6. World Scientific, Singapore (1996)

Minimizing Rules and Nonterminals in Semi-conditional Grammars: Non-trivial for the Simple Case

Henning Fernau[1], Lakshmanan Kuppusamy[2(✉)], Rufus O. Oladele[3], and Indhumathi Raman[4]

[1] CIRT, Fachbereich 4, Universität Trier, 54286 Trier, Germany
[2] School of Computer Science and Engineering, VIT, Vellore 632 014, India
klakshma@vit.ac.in
[3] Department of Computer Science, University of Ilorin, P.M.B.1515, Ilorin, Nigeria
roladele@unilorin.edu.ng
[4] School of Information Technology and Engineering, VIT, Vellore 632 014, India

Abstract. A simple semi-conditional (SSC) grammar is a form of regulated rewriting system where the derivations are controlled either by a permitting string alone or by a forbidden string alone and is specified in the rule. The maximum length i (j, resp.) of the permitting (forbidden, resp.) strings serves as a measure of descriptional complexity known as the degree of such grammars. In addition to the degree, the numbers of nonterminals and of conditional rules are also counted into the descriptional complexity measures of these grammars. We improve on some previously obtained results on computational completeness of SSC grammars by minimizing the number of nonterminals and/or the number of conditional rules for a given degree (i, j). More specifically, we prove that every recursively enumerable language is generated by an SSC grammar of (i) degree $(2, 1)$ with at most eight conditional rules and nine nonterminals, (ii) degree $(3, 1)$ with at most seven conditional rules and eight nonterminals and (iii) degree $(3, 1)$ with at most nine conditional rules and seven nonterminals.

Keywords: Simple semi-conditional grammars
Computational completeness · Descriptional complexity measures

1 Introduction

Context-free grammars play a vital role both in theory and practice. These grammars were invented by Chomsky [1,2] to describe the structures of words in sentences of natural languages. Soon, this idea was carried over to the study of artificial languages, most notably to the first high-level programming languages; see [9]. Already in these first papers, the decisive difference between the power of context-free grammars and that of general phrase-structure grammars was recognized; the latter describe the class of recursively enumerable (RE) languages. One of the key issues in many theoretical studies is therefore the following question: With what kind of control mechanisms, or regulations, can a context-free grammar describe all of RE? Examples of such regulations on context-free grammars are graph-controlled, programmed, matrix, random context, scattered context and also (simple) semi-conditional grammars. Another related issue investigated in formal language theory is the following question: How much of the available resources (e.g., how many nonterminals) is needed to achieve certain computability power with a given computational or, in our case, grammatical device? We refer to [3–5] to showcase the importance and history of nonterminal complexity for regulated context-free grammars. Another classical measure of descriptional complexity is the number of rules of a grammar [3,18]. With semi-conditional (SC) grammars, a variant of this measure is of special interest, counting in only the so-called conditional rules, which are the rules that are regulated and hence not truly context-free; clearly these are the rules which are responsible for (S)SC grammars to characterize RE [14–16,18]. This paper deals only with SSCG and not with SCG and hence a literature survey on non-simple semi-conditional grammars is avoided. In a nutshell, going from SCG to SSCG seems to necessitate more resources (nonterminals, conditional rules). Hence, none of our results puts us into a position to improve on the existing descriptional complexity results for SCG. On the other hand, this paper improves the existing results on the descriptional complexity measures of nontermainals, or conditional rules or both in obtaining computational completeness of SSC.

A semi-conditional grammar is an extension of context-free grammar in which each rule is associated with two strings called the permitting and forbidden string. A rule can be applied to a sentential form w only if w contains the permitting string (i.e., positive context) and does not contain the forbidden string (i.e., negative context) as a subword. The maximal length of the permitting string (say i) and the forbidden string (say j) constitutes the $d$egree of the grammar and is denoted by (i, j). A semi-conditional grammar is termed *simple* (and is denoted by SSCG for short) if for each rule at most either the permitting string or the forbidden string is present; refer to [14]. If both these control strings are absent in a rule, then the rule is said to be an unconditional. Otherwise the rule is termed *conditional.*

The descriptional complexity of an SSCG is measured by its number of nonterminals and its number of conditional rules. It is known that even simple semi-conditional grammars of degree $(1, 1)$ are computationally complete, i.e., they characterize the family RE of recursively enumerable languages; see [11]. However, nothing is known on the nonterminal complexity in this case. Our focus is therefore on minimizing (as much as possible) the number of nonterminals and/or the number of conditional rules required by SSCG of degree $(2, 1)$ and $(3, 1)$ which characterize RE. To arrive at such results, it is often helpful to make use of other similar results and to simulate the normal forms of type-0 grammars. We refer to [8] for several normal forms of type-0 grammars provided by Geffert.

Table 1. Races of descriptional complexity measures of SSCG to describe RE; #CR and #NT denote the number of conditional rules and nonterminals, respectively.

Degree (i, j)	# CR	# NT	Reference	Degree (i, j)	# CR	# NT	Reference
(2, 1)	12	13	[15]	(3, 1)	8	11	[18]
(2, 1)	10	12	[18]	(3, 1)	8	9	[16]
(2, 1)	9	10	[10]	(3, 1)	7	8	Theorem 2
(2, 1)	8	9	Theorem 1	(3, 1)	9	7	Theorem 3

We summarize the main results of this paper and also the previous results from the literature, on which it improves, in Table 1. Notice that if we neglect the number of conditional rules as a descriptional complexity aspect—especially for Theorem 3, which is rather a trade-off result—then our results in this paper improve on all previously published ones. Obviously, a kind of race going on—striving for more and more succinct descriptions of RE. Here, we improve on the existing results on nonterminals, or conditional rules or both to obtain computational completeness using SSCG; see Table 1.

A reader might wonder why the number of unconditional rules in SSCG is not accounted for in Table 1. It does not play a role in the descriptional complexity of SSCG, since the number of unconditional rules $(S \to x, 0, 0)$ of SSCG which simulating a context-free rule $S \to x$ of Geffert normal form is unbounded, because the number of context-free rules in Geffert normal form is unbounded. Hence a bound on the number of unconditional rules in SSCG is not studied in literature and so we follow this tradition in this paper.

Notice that this type of mechanisms can be quite tricky to handle when it comes to reducing both the number of conditional rules and the number of nonterminals. Quite recently, Oladele and Isah claimed in [17] that for every recursively enumerable language, there is an SSC grammar of degree $(2, 1)$ with no more than six conditional productions and seven nonterminals. However, the simulation presented in the paper is incorrect and hence we do not include the claim of [17] in Table 1. The bug in the result is the non-context-free rules $AA \to \lambda$ and $BBB \to \lambda$ are not properly simulated, as the simulation does not check, e.g., that at least two A's appear in the sentential form side-by-side.

2 Preliminaries and Definitions

In this paper it is assumed that the reader is familiar with the fundamentals of language theory and mathematics in general. Let $\mathbb{N}$ denote the set of non-negative integers. Let Σ^* denote the free monoid generated by a finite set Σ called the alphabet under an operation termed concatenation, where λ denotes the unit of Σ^*, also called the empty string. Any element of Σ^* is called a word or string (over Σ). Any subset of Σ^* is called a language. A word v is a subword (or substring) of $x \in \Sigma^*$ if there are words u, w such that $x = uvw$. Let

$sub(x) \subseteq \Sigma^*$ denote the set of all subwords of $x \in \Sigma^*$. Clearly, $sub(x)$ is a finite language. Given a word $w \in \Sigma^*$, $|w|$ represents the length of w.

2.1 Semi-conditional Grammars

A *semi-conditional grammar* is a quadruple $G = (V, T, P, S)$, where V is the total alphabet, $T \subset V$ is the terminal alphabet, $S \in V \setminus T$ is the starting symbol, P is a finite set of productions of the form $(A \to x, \alpha, \beta)$ with $A \in V \setminus T$, $x \in V^*$, $\alpha, \beta \in V^+ \cup \{0\}$, where $0 \notin V$ is a special symbol, intuitively meaning that the condition is missing. We extend the length function by setting $|0| = 0$ for this special symbol. The production $l : (A \to x, \alpha, \beta) \in P$ (with label l) is said to be *conditional* if $\alpha \neq 0$ or $\beta \neq 0$, and *unconditional* otherwise. The string α is called the permitting string and β is called the forbidden string, as formally explained now. The production labeled l can be applied on a string $u \in V^*(V \setminus T)V^*$ if and only if $A \in sub(u)$ and ($\alpha \in sub(u)$ or $\alpha = 0$) and ($\beta \notin sub(u)$ or $\beta = 0$). Under these conditions, $u = u_1 A u_2$ will be transformed into $v = u_1 x u_2$, which is denoted by $u \Rightarrow_l v$. When no confusion exists on the rule being applied we avoid mentioning the label and we simply write $u \Rightarrow v$. The language of G is defined as $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$, where $\Rightarrow^*$ denotes the reflexive and transitive closure of $\Rightarrow$. An SSCG is said to be of *degree* (i, j), where $i, j \in \mathbb{N}$, if in every rule $(A \to x, \alpha, \beta)$ of P we have $|\alpha| \leq i$ and $|\beta| \leq j$. We denote by SSCG$(i, j; p, n)$, a family of languages generated by SSCGs where (a) (i, j) is its degree, (b) p is an upper bound on the number of conditional production rules and (c) n is an upper bound on the number of nonterminals.

2.2 Geffert Normal Forms

In [8], quite a number of normal forms for type-0 grammars have been derived. They all differ by the number of nonterminals that are used and also by the number of non-context-free rules. We will hence speak of (n, r)-GNF to refer to a Geffert normal form with n nonterminals and r non-context-free rules. However, all these normal forms characterize the class of recursively enumerable languages, or RE languages for short.

The best known normal form is the $(5, 2)$-GNF with nonterminals S (the start symbol) and A, B, C, D that uses context-free rules with S as its left-hand side; after using the context-free rules, in a second phase non-context-free erasing rules $AB \to \lambda$ and $CD \to \lambda$ are applied to finally derive a terminal string $t \in T^*$. The derivation in a grammar in $(5, 2)$-GNF proceeds in two phases, where the first phase splits into two stages. In phase one, stage one, rules of the form $S \to uSa$ are used, with $u \in \{A, C\}^*$, $a \in T$. In stage two, rules of the form $S \to uSv$ are used, with $u \in \{A, C\}^*$ and $v \in \{B, D\}^*$. Also, rules of the form $S \to uv$ are available [7] that prepare the transition into phase two, where the erasing non-context-free rules are used exclusively.

We now discuss two other normal forms of type-0 grammars due to Geffert [8]. We summarize a list of important properties of these normal forms in two propositions. The properties basically follow from the constructions of these normal forms and from what is well-known about $(5,2)$-GNF.

A type-0 grammar is said to be in $(4,2)$-Geffert Normal Form (in short $(4,2)$-GNF) if it has exactly four nonterminals S, A, B, C and non-context-free erasing rules of the form $AB \to \lambda$ and $CC \to \lambda$. This normal form is obtained from the more classical one (using nonterminals S, A, B, C, D) by applying the morphism $A \mapsto CAA$, $B \mapsto BBC$, $C \mapsto CA$ and $D \mapsto BC$ to all rules. This implies:

Proposition 1. *The following properties hold for* $(4,2)$*-GNF grammars:*

1. *If $S \Rightarrow^* w$, then $w \in \{CA, CAA\}^*\{S, CC, \lambda\}\{BC, BBC\}^*T^*$.*
2. *No sentential form derivable in the grammar contains any of the substrings BA, AAA, BBB, CCC.*
3. *If $S \Rightarrow^* w$, with $w = w't$, where $w' \in \{A, B, C\}^+$ and $t \in T^*$, then w' contains a most one occurrence from $\{AB, CC\}$ as a substring. This also rules out $ABCC$ and $ABAB$ as substrings of sentential forms.*
4. *Again, the derivation proceeds in two phases, the first one split into two stages. Only in phase two, the central part (either AB or CC) will appear.*

A type-0 grammar is said to be in $(4,1)$-Geffert Normal Form (in short $(4,1)$-GNF) if it has exactly four nonterminals S, A, B, C and a single non-context-free erasing rule of the form $ABC \to \lambda$. This normal form is obtained from the more classical one (using nonterminals S, A, B, C, D) by applying the morphism $A \mapsto AB$, $B \mapsto C$, $C \mapsto A$ and $D \mapsto BC$ to all rules. This implies:

Proposition 2. *The following properties hold for* $(4,1)$*-GNF grammars:*

1. *If $S \Rightarrow^* w$, then $w \in \{A, AB\}^*\{S, ABC, \lambda\}\{BC, C\}^*T^*$.*
2. *No sentential form derivable in the grammar contains any of the substrings CA and BBB.*
3. *If $S \Rightarrow^* w$, with $w = w't$, where $w' \in \{A, B, C\}^+$ and $t \in T^*$, then w' contains exactly one occurrence from $\{ABC, AC, ABBC\}$ as a substring. We refer to this substring as the* central part *of w. Notice that only with ABC as central part, possibly $w' \Rightarrow^* \lambda$ as intended in a derivation that yields a terminal string.*
4. *Again, the derivation proceeds in two phases, the first one split into two stages. Only in phase two, a central part will appear.*

2.3 Changing the Order of Rule Applications

It is well-known that in a context-free grammar, two rules $r_1 : A \to u$ and $r_2 : B \to v$, with $A \neq B$, can be applied independently on a sentential form w in the following sense: Whenever $w \Rightarrow w' \Rightarrow w''$ by first applying r_1 and then r_2, then we also have $w \Rightarrow \bar{w} \Rightarrow w''$ by first applying r_2 and then r_1. In other words, we could *defer* the application of r_1 by first applying r_2, as an application of r_2 cannot render r_1 inapplicable, because $A \neq B$.

This easy argument is no longer true within regulated rewriting. However, in the case of simple semi-conditional grammars, weaker variants of this argument still hold. For instance, if $r_2 : (A \to u, 0, 0)$ is an unconditional rule and $r_1 : (B \to v, \alpha, 0)$ is a conditional rule, then we can defer applying r_1 if $A \notin sub(B\alpha)$. Similarly, if $r_2 : (A \to u, 0, 0)$ is an unconditional rule and $r_1 : (B \to v, 0, \beta)$ is a conditional rule with $|\beta| = 1$, then we can defer applying r_1 if $A \neq B$ and $sub(u) \cap sub(\beta) = \{\lambda\}$. In fact, also weaker conditions would suffice to ensure applicability; it simply has to be avoided that new forbidden substrings are created by first applying r_2. In particular, if the symbols occurring in A and u are different from the symbols occurring in α or β, and if $A \neq B$, then we can defer applying r_1. Clearly, this argument can also be iteratively applied. This will allow us to argue that in certain simulations, without loss of generality, always the unconditional rules are applied first. In particular, we can avoid discussing derivations where a sequence of applications of unconditional rules is interrupted by applying some conditional rules and then continuing to apply unconditional rules. We will refer to this type of reasoning by saying *by rule-deferring arguments.* This type of argument also applies under some additional conditions if r_1 is a conditional rule. We will use it in many places below to shortcut otherwise quite lengthy proofs. Let us make this more to the point by describing the situation found in the following proofs in the form of a lemma.

Lemma 1. *Assume $G = (V, T, P, S)$ is an SSCG with all unconditional rules being of the form $(S \to x, 0, 0)$ only, where x contains at most one occurrence of S, and the conditional rules are either of the form $(Y \to y, \alpha, 0)$ with $S \notin sub(Y\alpha)$ or of the form $(Y \to y, 0, \beta)$ with $|\beta| = 1$, then we can assume (w.l.o.g.) that any derivation $S \Rightarrow^* w$ first starts by applying unconditional rules and then ends by applying conditional rules.*

Proof. Let $S \Rightarrow^* w$ be a derivation of w in G. As S is the start symbol and S does not occur in conditional rules by assumption, any such derivation starts with a sequence of rule applications of unconditional rules. In contradiction to our claim, we consider a derivation that then continues with some shortest possible sequence of applications of conditional rules but then again switches back to using an unconditional rule, say, of $(S \to v, 0, 0)$. Namely, if there existed a counter-example to our claim, then there would also exist a counter-example with a shortest possible sequence of applications of conditional rules as postulated. Assume that $(Y \to \eta, \alpha, \beta)$ was the last conditional rule applied in the described sequence. Then, our previous considerations show that we can switch the order of said last conditional rule application and the (hitherto following) application of $(S \to v, 0, 0)$. This contradicts the minimality of the chosen sequence of applications of conditional rules, as now (after deferring applying $(Y \to \eta, \alpha, \beta)$) we have found a derivation of w with a shorter sequence of applications of conditional rules, followed by the applications of unconditional rules. □

However, although this type of argument appears to be quite simple, there are certain subtleties that should be taken care of. For instance, the reader might have wondered why we put down the condition $|\beta| = 1$ for the forbidden context condition β for r_2 above. However, if $u = \lambda$ within r_2, then it could be the case that a new forbidden context of length two or larger is created by executing r_1

before r_2. As we are dealing with semi-conditional grammars of degree $(i, 1)$ only in this paper, this caveat is not that important to us.

3 Main Results

In this section, we present the main results of this paper stated in Table 1.

3.1 SSCG of Degree $(2, 1)$

In the following, we show that 9 nonterminals and 8 conditional rules are sufficient to achieve computational completeness using a simple semi-conditional grammar of degree $(2, 1)$. This result improves on a result of Masopust [10] ($\mathrm{SSCG}(2, 1; 9, 10) = \mathrm{RE}$) in terms of both the nonterminals and the conditional rules, maintaining the degree. The arguments in the proof rely on a pretty extensive discussion of cases, which somehow indicates that it might be difficult to achieve further improved computational completeness results with this degree.

$$\begin{array}{llll}
R1\colon & (A \to \#\$A',\ 0,\ A') & R5\colon & (C' \to \lambda,\ A'A',\ 0) \\
R2\colon & (B \to B',\ 0,\ B') & R6\colon & (A' \to \#,\ A'\$,\ 0) \\
R3\colon & (C \to C'\$\#,\ 0,\ \#) & R7\colon & (\$ \to \#,\ \#\#,\ 0) \\
R4\colon & (B' \to A',\ B'C',\ 0) & R8\colon & (\# \to \lambda,\ 0,\ \$)
\end{array}$$

Fig. 1. Simulating rules of $\mathrm{SSCG}(2, 1; 8, 9)$ for $ABC \to \lambda$.

Theorem 1. *For each RE language L, there is an* $\mathrm{SSCG}(2, 1)$ *with only nine nonterminals and eight conditional rules for L, i.e.,* $\mathrm{SSCG}(2, 1; 8, 9) = \mathrm{RE}$.

The basic idea of the simulation of rule $ABC \to \lambda$ (as shown in Fig. 1) is to first mark ABC as $\#\$A'B'C'\$\#$ with the first three rules and then verify that the according replacements occurred side-by-side.

Proof. Consider some RE language $L \subseteq T^*$ represented by a type-0 grammar $G = (\{S, A, B, C\}, T, P, S)$ in $(4, 1)$-GNF. G is simulated by a simple semi-conditional grammar of degree $(2, 1)$ $G' = (V, T, P', S)$, where $V = N \cup \{A', B', C', \#, \$\} \cup T$ and $P' = \{(S \to x, 0, 0) \mid (S \to x) \in P\} \cup R$ where R is the set of 8 conditional rules listed in Fig. 1. Obviously, $|V \setminus T| = 4 + 5 = 9$.

We first prove that $L(G) \subseteq L(G')$. The intended derivations of phase one of the given $(4, 1)$-GNF grammar can be simulated by the same rules (i.e., $S \to x$) of G'. The following shows the intended simulation of the erasing rule $ABC \to \lambda$, according to the rules presented in Fig. 1. Here, we assume that $S \Rightarrow^* uABCvt$ according to G, i.e., $u \in \{A, AB\}^*$, $v \in \{BC, C\}^*$, $t \in T^*$.

$$\begin{aligned}
uABCvt &\Rightarrow_3 uABC'\$\#vt \Rightarrow_2 uAB'C'\$\#vt \Rightarrow_1 u\#\$A'B'C'\$\#vt \\
&\Rightarrow_4 u\#\$A'A'C'\$\#vt \Rightarrow_5 u\#\$A'A'\$\#vt \Rightarrow_6 u\#\$\#A'\$\#vt \\
&\Rightarrow_6 u\#\$\#\#\$\#vt \Rightarrow_7 u\#\$\#^4vt \Rightarrow_7 u\#^6vt \Rightarrow_8^6 uvt. \qquad (1)
\end{aligned}$$

To prove the reverse part, i.e., $L(G') \subseteq L(G)$, consider a sentential form w derivable by the simulating grammar G' that is also derivable by the $(4,1)$-GNF grammar G. If w contains S, then on applying the unconditional rule $(S \rightarrow x, 0, 0)$ of P', we could simulate the context-free rule $S \rightarrow x$ of G and get a resultant string w' as desired. Alternatively, we could have applied any of the rules $R1$, $R2$, or $R3$ on w. As w stems from phase one of G, we know that $w = uSvt$ with $u \in \{A, AB\}^*$, $v \in \{BC, C\}^*$, $t \in T^*$, see Proposition 2. By rule-deferring arguments (Lemma 1), we can avoid discussing what might happen if rules $R1$, $R2$ or $R3$ are applied on $w = uSvt$, as unconditional rules have to be applied afterwards anyways to finally derive a terminal string.

Henceforth, we assume that phase one has been correctly simulated and hence $w = uSvt$ derived $w' = u\alpha vt$. By Proposition 2, $w' = u\alpha vt$ with $u \in \{A, AB\}^*$, $\alpha \in \{ABC, AC, ABBC\}$, $v \in \{BC, C\}^*$, $t \in T^*$, i.e., α is the central part.

Consider $w' \Rightarrow w_1$. As $w' \in (\{A, B, C\} \cup T)^*$, only rules $R1$, $R2$, or $R3$ are applicable. Notice that $R2$ introduces B' which never occurs in any negative context but in $R2$ itself; however, it occurs in the positive context in $R4$. As we need to apply $R4$ in order to be able to apply $R5$ and then possibly $R6$, we can assume in our discussion that, w.l.o.g., first rule $R2$ was applied. Hence, w_1 is obtained from w' by replacing any occurrence of B in w' by B'.

Now, we consider the possibilities for $w_1 \Rightarrow w_2$. Looking at Fig. 1 again, it is clear that only $R1$ or $R3$ are applicable.

Case (i): $w_1 \Rightarrow_1 w_2$. So, some occurrence of A in w_1 was replaced by $\#\$A'$. None of the rules $R1$, $R2$ or $R3$ is applicable any more (in particular $R3$ is not, as $\#$ was introduced). As $\#$ is to the left of $\$$, $R6$ is not applicable. As C' is not present, $R4$ and $R5$ are not applicable. Hence, the derivation is stuck.

Case (ii): $w_1 \Rightarrow_3 w_2$. This means that some occurrence of C in w_1 was replaced by $C'\$\#$. How could we continue for $w_2 \Rightarrow w_3$ now? The only rule that is possibly applicable now is $R1$, turning one occurrence of A into $\#\$A'$. None of the rules $R1$, $R2$ or $R3$ is applicable any more. As $\{A'A', A'\$\} \cap sub(w_3) = \emptyset$, neither $R5$ nor $R6$ applies. Hence, only $R4$ might apply within $w_3 \Rightarrow w_4$, which means that we can think of having obtained w_4 from w' by replacing the substring BC by $A'C'\$\#$ and one occurrence of A by $\#\$A'$. Recall that we introduced the decomposition $w' = u\alpha vt$. Now, the substring BC could be found in αv, while an occurrence of A could be only present in $u\alpha$. Looking at the possibilities for $w_4 \Rightarrow w_5$, one observes that either $R2$ or $R5$ was applied. After possibly applying $R2$, it can be observed that now, only (possibly) $R5$ is applicable, leading us to the same string as if we had first applied $R5$ and then $R2$ (in particular, applying $R5$ first would not block any possibilities for applying $R2$). W.l.o.g., we can assume $w_4 \Rightarrow_5 w_5$. As $R5$ requires the substring $A'A'$ in w_4, this means that w_5 was obtained from $w' = u\alpha vt$ by replacing the substring ABC by $\#\$A'A'\$\#$. As $\alpha \in \{ABC, AC, ABBC\}$, the case $\alpha = ABC$ is enforced. We now consider $w_5 \Rightarrow w_6$. Again, we can argue that applying $R2$ on w_5 gives no real progress, as the next rule that must be applied is then $R6$. So, we can, w.l.o.g., assume that $w_5 \Rightarrow_6 w_6$, as using $R6$ does not block $R2$.

Hence, $w_6 = u\#\$A'\#\$\#vt$ or $w_6 = u\#\$\#A'\$\#vt$. No rule is applicable in the former case and hence in the latter case, we look at $w_6 \Rightarrow w_7$ now. We can either apply $R2$ or $R6$. Again, some rule-deferring argument applies, so that we can assume that $w_6 \Rightarrow_6 w_7 = u\#\$\#\#\$\#vt$ without loss of generality. Now, on w_7, rules $R1$, $R2$ or $R7$ apply. While the application of $R2$ is independent of the other two and may hence be deferred (see Cases A and B), this is not true for a possible application of $R1$, because this rule application introduces a new occurrence of \$ in particular, and applying $R7$ then gives a new situation, as in particular this newly introduced occurrence of \$ can now vanish again, which is not possible in a situation where $R7$ is not applicable. An application of $R7$ replaces one occurrence of \$ by #, leading to three possible results. Alternatively, we could first apply $R7$ on w_7 and then $R1$ to the resulting string, which is studied in Case A below. Now, or alternatively later, $R2$ might apply, so that again we could defer this application; see Case B and consider only the case when $R7$ is applied on w_7; see Case C.

Case A: We have to apply some thoughts if there is some occurrence of A within u of $w_7 = u\#\$\#\#\$\#vt$ that can be turned into $\#\$A'$ to get $u_1\#\$A'u_2\#\$\#\#\$\#vt = w_8$. On w_8, either $R2$ or $R7$ would apply. As the order of application of rules can be swapped, it is sufficient to consider only the case when $R7$ is applied; applying $R2$ is discussed in Case B below. This leads to $w_8 \Rightarrow_7^3 w_{11}$ with $w_{11} = u_1\#\#A'u_2\#^6vt$. Apart from applying $R2$ now (which can be again deferred to Case B), only $R8$ is applicable which yields $w_{11} \Rightarrow_8^8 w_{19} = u_1A'u_2vt$. For further continuation of derivation, see Case B below when $\alpha = \beta = \lambda$.
Case B: Applying $R2$ on $w'' = u_1\alpha A'u_2\beta vt$ where $\alpha \in \{\#\$, \#\#, \lambda\}$ and $\beta \in \{\#\$\#\#\$\#, \#^4\$\#, \#\$\#^4, \#^6, \lambda\}$.
In Case A discussions, we see that the application of $R2$ in each step is possible only if we assume that $u_2 = Bu_2'$ or that $u_2\beta = \lambda$ and $v = Bv'$. Applying $R2$ under these assumptions, we get $w_1' = u_1\alpha A'B'u_2'\beta vt$ or $w_1' = u_1\alpha A'B'v't$. No other rule application is possible except applying $R4$ in the latter case which yields the same as the former case when $\alpha = u_2'\beta = \lambda$ and $v = Cv'$. Now,

$$\begin{aligned} w_1' &= u_1A'B'Cv't \Rightarrow_3 u_1A'B'C'\$\#v't \Rightarrow_4 u_1A'A'C'\$\#v't \\ &\Rightarrow_5 u_1A'A'\$\#v't \Rightarrow_6^2 u_1\#\#\$\#v't \Rightarrow_7 u_1\#\#\#\#v't \Rightarrow_8^4 u_1v't \end{aligned} \tag{2}$$

This derivation is possible only when, in w_6, $AB \in sub(u)$, $v = Cv'$ or $A \in sub(u)$, $v = BCv'$. When the central part of w_6 vanishes, the u and v parts merge and this leads to another occurrence of ABC in the center. Equation (2) shows the simulation of $ABC \to \lambda$ of this newly occurred ABC which is allowed. One might wonder whether $R1$ can be applied on some A in $sub(u_1)$ in the penultimate step in Eq. (2). Such an $R1$ application will disable any further rule application since there is a # after u_1. This acts like a (right) guard and particularly $A'\#$ is not a permitted string to apply any rule.

Case C: $w_7 = u\#\$\#\#\$\#vt \Rightarrow_7 w_8$.

In this case, either $w_8 = u\#\$\#\#\#\#vt$ or $w_8 = u\#\#\#\#\$\#vt$. In either case, rules $R1$, $R2$ or $R8$ are applicable. By a rule-deferring argument, we assume that we prefer applying $R7$ over applying $R1$ or $R2$. Applying $R7$ on either possible variant of w_8, we get $w_9 = u\#\#\#\#\#\#vt$. Again by a rule-deferring argument, we can assume to apply $R8$, specifically, apply $R8$ repeatedly for six times on w_9 to get w_{15}. In other words, $w_9 \Rightarrow_8^6 w_{15} = uvt$. Obviously, $w' \Rightarrow^* w_{15}$ corresponds to applying the deletion rule $ABC \rightarrow \lambda$ once on w', and this is in fact the intended simulation. Finally, observe that the deferred applications of $R1$ or $R2$ in these discussions correspond to the discussions of Case (i) (and another rule-deferring argument given before) that also proves that this line of continuation would get stuck. By induction, $L(G') \subseteq L(G)$. □

As an illustration of the work of the previous construction, assume that $w = AABABCCBCab$ (i.e., $w = (A(AB(ABC)C)BC)ab$) could be derived using phase one of the Geffert normal form grammar. Clearly, our simulating grammar can derive w as well, using unconditional rules only. Now, we could terminate (using a sequence of 15 rule applications simulating $ABC \rightarrow \lambda$, here emphasized by underlining, as detailed in Eq. (1)) with

$$w = AAB\underline{ABC}CBCab \Rightarrow^{15} A\underline{ABC}BCab \Rightarrow^{15} \underline{ABC}ab \Rightarrow^{15} ab\,.$$

Alternatively, using the variation explained in Cases A and B above, we could derive ab as follows. Here, writing $\Rightarrow^7$ refers to the first seven derivations in Eq. (1) and writing $\Rightarrow^{10}$ refers to Eq. (2) which corresponds to applying $A'B'C \rightarrow \lambda$.

$$\begin{gathered} w = AAB\underline{ABC}CBCab \Rightarrow^7 AAB\#\$\#\#\$\#CBCab \Rightarrow_1 \\ A\#\$A'B\#\$\#\#\$\#CBCab \Rightarrow_7^3 A\#\#A'B\#^6CBCab \Rightarrow_8^8 \\ AA'BCBCab \Rightarrow_2 A\underline{A'B'C}BCab \Rightarrow^{10} \underline{ABC}ab \Rightarrow^{15} ab. \end{gathered}$$

In the following, we show that the central parts $AC, ABBC$ of $(4,1)$-GNF (according to Proposition 2) cannot be removed by the rules shown in Fig. 1.

$$ACab \Rightarrow_3 AC'\$\#ab \Rightarrow_1 \#\$A'C'\$\#ab.$$

$$\begin{gathered} ABBCab \Rightarrow_1 \#\$A'BBCab \Rightarrow_2 \#\$A'BB'Cab \Rightarrow_3 \\ \#\$A'BB'C'\$\#ab \Rightarrow_4 \#\$A'BA'C'\$\#ab \Rightarrow_2 \#\$A'B'A'C'\$\#ab. \end{gathered}$$

According to the above discussions, none of the rules from $R1$ to $R8$ could be applied neither on $\#\$A'C'\$\#ab$ (in the former case) nor on $\#\$A'B'A'C'\$\#ab$ (in the latter case) and hence both the derivations get stuck.

3.2 SSCG of Degree (3, 1)

In the next theorem, we improve on the existing results in terms of both the number of conditional rules and the number of nonterminals. In [16], Okubo showed that each recursively enumerable language can be generated by a simple semi-conditional grammar of degree $(3,1)$ with 8 conditional productions and 9 nonterminals, based on a simulation of a type-0 grammar in $(4,1)$-GNF. Okubo's simulation used two border markers \$ and #; here we show that only one border marker # is sufficient to achieve the desired result.

Theorem 2. *For each RE language L, there is an* SSCG(3, 1) *with no more than seven conditional production rules and eight nonterminals that describes L.*

As with the previous result, the simulation of $ABC \to \lambda$ again proceeds by first marking the central part ABC, this time as $\#A'B'C'$. However, with permitting context of length three, it is far easier to verify that the three replaced symbols A, B, C form a contiguous subword.

$$\begin{array}{llllllll}
R1\colon & (A \to \#A', & 0 & , \#) & R5\colon & (B' \to \lambda, & A'B'\#, & 0\ ) \\
R2\colon & (B \to B' & , 0 & , B') & R6\colon & (A' \to \lambda, & \#A'\#\ , & 0\ ) \\
R3\colon & (C \to C' & , 0 & , C') & R7\colon & (\# \to \lambda\ , & 0 & , A') \\
R4\colon & (C' \to \# & , A'B'C', & 0\ ) & r\colon & (S \to x\ , & 0 & , 0\ )
\end{array}$$

Fig. 2. Simulation of $ABC \to \lambda$ and of $S \to x$ by SSC(3, 1; 7, 8)

Proof. We assume that the recursively enumerable language $L \subseteq T^*$ is described by some type-0 grammar $G = (N, T, P, S)$ in $(4,1)$-GNF, with $N = \{S, A, B, C\}$. We are going to describe an $\text{SSCG}(3,1)$ $G' = (V, T, P', S)$ for L, with $V = N \cup \{A', B', C', \#\} \cup T$. The particular non-context-free erasing rules $ABC \to \lambda$ is simulated by the simple semi-conditional rules given in Fig. 2.

The intended derivations of phase one of the given $(4,1)$-GNF grammar can be simulated by the same rules (i.e., $S \to x$) of G'. The intended simulation of the erasing rule $ABC \to \lambda$ is as follows. Assume that $S \Rightarrow^* uABCvt$ according to G, i.e., $u \in \{A, AB\}^*$, $v \in \{BC, C\}^*$, $t \in T^*$.

$$\begin{aligned}
c\ uABCvt &\Rightarrow_1 u\#A'BCvt \Rightarrow_2 u\#A'B'Cvt \Rightarrow_3 u\#A'B'C'vt \\
&\Rightarrow_4 u\#A'B'\#vt \Rightarrow_5 u\#A'\#vt \Rightarrow_6 u\#\#vt \Rightarrow_7 u\#vt \Rightarrow_7 uvt.
\end{aligned}$$

This shows that $L(G) \subseteq L(G')$.

Conversely, consider some sentential form $w \in (N \cup T)^*$ of G'. By induction, we can assume that w is also a sentential form of G. We are going to argue that after some further derivation steps of G', either the derivation is stuck, or some sentential form is produced that is also a valid sentential form in G. By induction, this would show that $L(G') \subseteq L(G)$.

As w is a valid sentential form in G, it is possibly part of the derivation taking place in phase one in this $(4,1)$-GNF grammar only. If w contains S, then on applying the unconditional rule $(S \to x, 0, 0)$ of P', we could simulate the context-free rule $S \to x$ of G and get a resultant string w'. By rule-deferring arguments, we need not discuss applying any of the rules $R1$, $R2$, or $R3$ on w. As w stems from phase one of G, we know that $w = uSvt$ with $u \in \{A, AB\}^*$, $v \in \{BC, C\}^*$, $t \in T^*$, see Proposition 2.

In order to have some fruitful derivations, we assume that $w = uSvt \Rightarrow^* w' = u\alpha vt$ where $u \in \{A, AB\}^*$, $\alpha \in \{ABC, ABBC, AC\}^*$ is the central part, $v \in \{BC, C\}^*$, $t \in T^*$, see Proposition 2. Considering the possibilities for $w' \Rightarrow w_1$, it is clear that only rules $R1$, $R2$ or $R3$ are applicable. The case discussion can be simplified based on the following claims.

Claim 1: If any of the rules $R1$, $R2$ or $R3$ is applied, all of them have to be applied, one after the other, and the order of rule applications does not matter.

Namely, if $w' \Rightarrow_1 w_1$, then A' is introduced. In order to eliminate A' again (which must be done in any terminal derivation), $R6$ must be applied in some later stage. But this requires # to occur twice in the string. The second occurrence of # cannot been introduced by re-applying $R1$ and can be introduced by $R4$. However, $R4$ requires each of A', B', C' to be present in the string, so that all of the rules $R1$, $R2$ and $R3$ must have been applied before $R4$ can be applied. Similarly, if $w' \Rightarrow_3 w_1$, then C' is introduced. To get rid of C' again, $R4$ has to be used. This ensures that all of the rules $R1$, $R2$ and $R3$ have to be applied before $R4$. Finally, if $w' \Rightarrow_2 w_1$, then B' is introduced. B' is further processed only with $R5$, which requires A' to be introduced (by applying $R1$), effectively to the left of B', and also # should be introduced. But as # is to occur to the right of B', it cannot be the # occurrence introduced when applying $R1$. Hence, $R4$ had to be executed before, so that again all of the rules $R1$, $R2$ and $R3$ must have been applied earlier. Looking at the (negative) context conditions, it is clear that the order of rule applications is irrelevant.

Claim 2: If any one of the rules $R1$, $R2$ or $R3$ are applied, then no other rules are applicable but the mentioned ones. If two rules from $R1$, $R2$ or $R3$ are applied, then again no other rules are applicable but the mentioned ones.

This claim can be easily seen by looking at the permitting context of $R4$, $R5$ and $R6$, which requires all of the rules $R1$, $R2$, $R3$ to be applied, as discussed before. Neither $R7$ applies, as # (if present) sees A' to its right.

Claim 3: On applying all of the rules $R1$, $R2$, $R3$ and only these, none of these rules is applied twice.

This is seen by looking at the forbidden contexts of these rules. As a consequence, after applying all of the rules $R1$, $R2$, $R3$ exactly once, some rule different from these must be applied. This fact is made more precise in the following claim.

Claim 4: After applying all of the rules $R1$, $R2$ and $R3$, Rule $R4$ has to be applied next.

According to the previous three claims, we only need to discuss $w' \Rightarrow_1 w_1 \Rightarrow_2 w_2 \Rightarrow_3 w_3$. Hence, w_3 contains exactly one occurrence of A', B', C', and of #. This alone restricts the possible next rules to $R4$ and $R5$. But, as we have argued above, $A'B'\# \notin sub(w_3)$. This shows the claim.

Now, reconsider the structure of $w' = u\alpha vt$, where $u \in \{A, AB\}^*$, $\alpha \in \{ABC, ABBC, AC\}^*$ is the central part, $v \in \{BC, C\}^*$, $t \in T^*$. As we have to apply $R4$ on w_3, $A'B'C' \in sub(w_3)$, which requires that $ABC \in sub(w')$. This completely determines the structure of w' that we have to consider, which is $w' = uABCvt$, which also yields that $w_3 = u\#A'B'C'vt$. As the only applicable rule on w_3 is $R4$, we arrive at $w_4 = u\#A'B'\#vt$. We could now either apply $R3$ or apply $R5$. However, as $w_4 \Rightarrow_3 w_5$ does not create new permitting contexts and as then $w_5 \Rightarrow_5 w_6$ is enforced, we can defer applying $R3$ now. So, we are left with $w_4 \Rightarrow_5 w_5$. Hence, $w_5 = u\#A'\#vt$. We could now apply one of $R2$, $R3$, $R6$. However, as applying $R2$ or $R3$ does not create new permitting contexts (because A' is always missing in the correct position), we can defer applying these rules. Therefore, we are left with $w_5 \Rightarrow_6 w_6 = u\#\#vt$. We can now apply $R2$, $R3$ or $R7$. Again, we can argue that we can defer applying $R2$ and $R3$, i.e., $w_6 \Rightarrow_7 w_7 = u\#vt$. Once more, we can defer applying $R2$ and $R3$, so that only $w_7 \Rightarrow_7 w_8 = uvt$. This is in fact the intended derivation.

By induction, it is true that $L(G') \subseteq L(G)$. □

The following result is the currently best one if we focus on the nonterminal complexity of SSCG$(3, 1)$, as we can bring it down to seven. However, the number of conditional rules increases when compared to the previous theorem, and also when compared to Okubo's result. It is still an open question if six nonterminals suffice in order to describe all RE languages by SSCGs.

Theorem 3. *For every RE language L, there is an* SSCG$(3, 1)$ *with only seven nonterminals and 9 conditional rules describing L, i.e.,* SSCG$(3, 1; 9, 7) = $ RE.

$R1$: $(A \to \$_1\$_2,\ 0,\ \$_2)$		$R5$: $(C \to \$_2\$_1,\ 0,\ \$_2)$
$R2$: $(B \to \$_2\$_3,\ 0,\ \$_3)$		$R6$: $(C \to \$_3\$_2,\ 0,\ \$_3)$
$R3$: $(\$_1 \to \$_3,\ \$_1\$_2\$_2,\ 0)$		$R7$: $(\$_1 \to \$_3,\ \$_2\$_2\$_1,\ 0)$
$R4$: $(\$_2 \to \lambda,\ \$_3\$_2\$_2,\ 0)$		$R8$: $(\$_2 \to \lambda,\ \$_3\$_2\$_3,\ 0)$
$R9$: $(\$_3 \to \lambda,\ 0,\ \$_2)$		r: $(S \to x,\ 0,\ 0)$

Fig. 3. Simulating rules of an SSCG$(3, 1; 9, 7)$ grammar for $AB \to \lambda$, $CC \to \lambda$, $S \to x$.

The idea of this simulation already differs from the previous ones by using a different normal form, namely, $(4, 2)$-GNF to start with. The effect of this measure is that we might gain on the number of nonterminals, because less context information needs to be checked, but we lose on the number of conditional rules compared to the previous result. One main trick is to find a simulation where both $AB \to \lambda$ and $CC \to \lambda$ are simulated in a way that rules can be re-cycled. Otherwise, more nonterminals and more rules would be needed.

Proof. Consider a type-0 grammar $G = (N, T, P, S)$ in $(4, 2)$-GNF. We construct a simple semi-conditional grammar $G' = (V, T, P', S)$ where $V = N \cup \{\$_1, \$_2, \$_3\} \cup T$ and $P' = \{(S \to x, 0, 0) \mid (S \to x) \in P\} \cup R$ where R is the set of 9 conditional rules listed in Fig. 3 such that $L(G') = L(G)$. Obviously the number of nonterminals $|V \setminus T|$ of G' is seven.

We first prove $L(G) \subseteq L(G')$. The context-free rules $S \to x$ of G are incorporated without any context conditions into G'. The non-context-free erasing rules

$AB \to \lambda$ and $CC \to \lambda$ are simulated by the simple semi-conditional rules given in Fig. 3.

The intended simulations are as follows:

$$cAB \Rightarrow_1 \$_1\$_2B \Rightarrow_2 \$_1\$_2\$_2\$_3 \Rightarrow_3 \$_3\$_2\$_2\$_3 \Rightarrow_4 \$_3\$_2\$_3 \Rightarrow_8 \$_3\$_3 \Rightarrow_9^2 \lambda.$$
$$CC \Rightarrow_5 C\$_2\$_1 \Rightarrow_6 \$_3\$_2\$_2\$_1 \Rightarrow_7 \$_3\$_2\$_2\$_3 \Rightarrow_4 \$_3\$_2\$_3 \Rightarrow_8 \$_3\$_3 \Rightarrow_9^2 \lambda.$$

We first prove some properties of our grammar G' to facilitate proving the reverse inclusion $L(G') \subseteq L(G)$.

Claim 1: If $w \in V^*$ with $\$_1, \$_2, \$_3 \notin sub(w)$, then only the unconditional rules $(S \to x, 0, 0)$ or the rules $R1$, $R2$, $R5$, $R6$ given in Fig. 3 might apply.

Claim 2: If $w \in V^*$ with $\$_1, \$_2, \$_3 \notin sub(w)$, and if $w \Rightarrow_x w' \Rightarrow_r w''$ using some conditional rule given in Fig. 3, with $x \in \{1, \ldots, 9\}$, and some unconditional rule r in G', then there is also a valid derivation $w \Rightarrow_r v \Rightarrow_x w''$.
Proof of Claim 2: As $w' \Rightarrow_r w''$, S has to be present in w' (and hence in w, because none of the conditional rules ever touch S). But then, we could also first apply r to w, arriving at v, and then $v \Rightarrow_x w''$, as the applicability conditions of none of the conditional rules can be influenced by applying an unconditional rule in G'. By induction, this implies the following claim.

Claim 3: Any derivation $S \Rightarrow^* w$ of G' can be split, w.l.o.g., into two phases: first, only unconditional rules are applied, and then, only conditional ones.

Claim 4: If $S \Rightarrow^* w \in V^*$ with $\$_1, \$_2, \$_3 \notin sub(w)$, and if $w \Rightarrow_2 w' \Rightarrow^* w''$ or $w \Rightarrow_5 w' \Rightarrow^* w''$ in G', then $w'' \notin T^*$.
Proof of Claim 4: If $w \Rightarrow_2 w'$ or $w \Rightarrow_5 w'$, then both $\$_2$ and $\$_3$ occur in w', each of them exactly once, but no $\$_1$ occurs. Hence, none of the rules listed in Fig. 3 might apply. Possibly, we could apply an unconditional rule r, leading to some string w''. According to the proof of Claim 3, then we could re-arrange the derivation so that we arrive at a situation where we need not consider them. In particular, we can conclude that $w'' \notin T^*$.

Now, consider a sentential form w derivable by a simulating grammar G' as well as by the $(4, 2)$-GNF G. According to Claim 3, we can assume that the derivation process of G' that we are observing is split into two phases. If w contains S, then we are still in the first phase; on applying the unconditional rule $(S \to x, 0, 0)$ of P', we simulate the context-free rule $S \to x$ of G and get a resultant string w' that is also derivable in G.

Alternatively, we have reached the second phase. As w is also a sentential form of G, $w = u\alpha vt$ where $u \in \{CA, CAA\}^*$, $\alpha \in \{\lambda, CC\}$, $v \in \{BC, BBC\}^*$ and $t \in T^*$; see Proposition 1. If $w \in T^*$, nothing has to be shown, so that we may assume that $u\alpha v \neq \lambda$. Incidentally, also within G, we must have reached phase two, so that only rules $AB \to \lambda$ or $CC \to \lambda$ apply. Moreover, Claims 1 and 4 together show that in G', Rules $R1$ or $R5$ apply to make any progress, leading to $w \Rightarrow w'$. Now either $\$_1\$_2$ or $\$_2\$_1$ is a substring of w', but S is not, as we are in the second phase. Therefore, if $w' \Rightarrow w''$, then either Rule $R2$ or $R6$ was applied to w'. Hence, w'' contains exactly one occurrence of $\$_1$ and of $\$_3$ and two occurrences of $\$_2$. This means that the only applicable rules are now Rules $R3$,

$R4$, or $R7$. In order to apply them, we must find three symbols from $\{\$_1, \$_2, \$_3\}$ occurring next to each other. This necessitates that $w' \Rightarrow w''$ was implemented by replacing a B or a C next to the substring $\$_1\$_2$ or $\$_2\$_1$ of w'. In turn, this substring corresponded to a symbol A or C in w that was replaced to obtain w'. We are now discussing all possible different scenarios for $w = xXYyt$ with $t \in T^*$, $XY \in \{A, B, C\}^2$ and $x, y \in \{A, B, C\}^*$ with the idea that $w'' = x\xi yt$, with $\xi \in \{\$_1, \$_2, \$_3\}^4$.

$\underline{w \Rightarrow_1 w' \Rightarrow_2 w''}$: If $X = A$, then $Y = B$, so that $w'' = x\$_1\$_2\$_2\$_3yt$. If we now apply $R4$, then the derivation gets stuck. The only remaining applicable rule would be $R3$, yielding $w''' = x\$_3\$_2\$_2\$_3yt$. We mark this situation as (*). Now, only Rule 4 is applicable, then only $R8$ and then (only) $R9$ (twice), leading to xyt, which actually corresponds to applying $AB \to \lambda$ as a rule in G (as desired). Notice that on $x\$_3\$_3yt$, obtained after deleting $\$_2$ from w''' by using $R4$ and then $R8$, we could also apply $R1$ or $R5$. But doing so, it is not possible to produce any of the permitting substrings for rules $R3$, $R4$, $R7$ or $R8$, which would be necessary to continue, so that this line of discussion knows no continuation. If $X = B$ and $Y = A$, then $w'' = x\$_2\$_3\$_1\$_2yt$. As in particular none of the permitting strings of rules $R3$, $R4$, $R7$ or $R8$ is a substring of w'', the derivation is stuck.

$\underline{w \Rightarrow_1 w' \Rightarrow_6 w''}$: If $X = A$, then $Y = C$, so that $w'' = x\$_1\$_2\$_3\$_2yt$. From here on, no rule is applicable. If $X = C$ and $Y = A$, then $w'' = x\$_3\$_2\$_1\$_2yt$. Again, no rule is applicable.

$\underline{w \Rightarrow_5 w' \Rightarrow_2 w''}$: Assuming $X = C$ and $Y = B$, we get $w'' = x\$_2\$_1\$_2\$_3yt$. From here on, no rule is applicable. If $X = B$ and $Y = C$, then $w'' = x\$_2\$_3\$_2\$_1yt$. Again, no rule is applicable.

$\underline{w \Rightarrow_5 w' \Rightarrow_6 w''}$: Now, $X = Y = C$, but still, there are two possibilities for w'': (a) $w'' = x\$_2\$_1\$_3\$_2yt$. From here on, no rule is applicable. (b) $w'' = x\$_3\$_2\$_2\$_1yt$. Now, both Rules $R4$ and $R7$ are applicable. However, after applying $R4$, the derivation is stuck. On applying $R7$ instead, we arrive at $w''' = x\$_3\$_2\$_2\$_3yt$, a situation previously discussed under (*). Hence, xyt can be finally derived. Now, $w \Rightarrow^* xyt$ corresponds to applying the rule $CC \to \lambda$ of G to w, as desired.

By induction, the claimed inclusion $L(G') \subseteq L(G)$ follows. $\square$

This concludes the presentation of our results. We assume that considering even longer permitting contexts would help further reduce the other parameters of descriptional complexity, but as a degree of $(4, 1)$ was never attacked before, we refrain from attempting this here. However, as we have seen in this paper that going from permitting contexts of length two to permitting contexts of length three allows us to save nonterminals and/or conditional rules, we would expect further trade-offs when allowing for permitting contexts of length four.

4 Conclusions

Though (simple) semi-conditional grammars of degree $(1, 1)$ are known to characterize RE [11] (based on [13]), the construction of this result is not helpful to get any bounds on the nonterminal complexity of such systems. We pose it as an open question to construct a (simple) semi-conditional grammars of degree (1, 1) using only (any) fixed amount of nonterminals. The problem is still open even if we let the permitting or forbidden conditions to be sets of strings instead of strings, which brings us into an area usually called random context grammars.

Some further concrete research questions come to mind when comparing the results of this paper with those of the recent paper [5] that focused on general (i.e., non-simple) semi-conditional grammars. (a) Both with degree (2, 1) and with degree (3, 1), the number of nonterminals as well as the number of conditional rules increases when imposing simplicity on semi-conditional grammars. Is it always the case that simple semi-conditional grammars have to be less succinct compared to the general semi-conditional case? (b) With the general case, the nonterminal complexity could be further lowered when allowing for an unbounded number of conditional rules. For instance, five nonterminals are sufficient to describe any RE language with semi-conditional grammars of degree (3, 1). No results of this type are known for SSCG. Conversely, it seems to be open to find any non-trivial statement ruling out computational completeness for (S)SCG's with a certain (low) number of nonterminals. (c) For general semi-conditional grammars, some results are known for degree (2, 2); see [5], trading the length of the forbidden context for the number of conditional rules when compared to the best results for degree (2, 1). No interesting descriptional complexity results are known for simple semi-conditional grammars of degree (2, 2). Is there any principal reason why the length of the forbidden context should not matter for SSCG?

Let us finally discuss two related families of grammars.

(i) If we attach only sets of forbidden words, also called forbidding sets, to the rules of context-free grammars, then the grammar is called a *generalized forbidding grammar*. In this case, an upper bound on the size of the forbidding set is also taken as a (new) descriptional complexity measure. If we denote by $\mathrm{GFG}(j,k;p,n)$ the family of languages generated by generalized forbidding grammars, where j denotes the maximum length of a forbidden string, k denotes the maximum cardinality of a forbidding set, p denotes the number of conditional rules and n denotes the number of nonterminals, then it is proved in [12] that each of the following classes equals RE. (A) $\mathrm{GFG}(2,4;11,10)$; (B) $\mathrm{GFG}(2,6;10,9)$; (C) $\mathrm{GFG}(2,\infty;9,8)$; where ∞ signals that there is no bound on this parameter. We leave it for future study to improve on the above-stated results of [12].

(ii) An interesting fact is that the counterpart grammars of the above, namely generalized permitting grammars, are proved to be strictly included in the class of context-sensitive languages; see [6]. This shows the importance of the forbidding condition compared to the permitting condition.

Acknowledgments. This research would not have been possible without a visit of L. Kuppusamy and R. O. Oladele to the first author at the University of Trier, Germany, subsidized by CIRT, Center for Informatics Research and Technology. We also like to thank the referees for carefully reading and commenting on the submitted version.

References

1. Chomsky, N.: A note on phrase structure grammars. Inf. Control (now Inf. Comput.) **2**, 393–395 (1959)
2. Chomsky, N.: On certain formal properties of grammars. Inf. Control (now Inf. Comput.) **2**, 137–167 (1959)
3. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. EATCS Monographs in Theoretical Computer Science, vol. 18. Springer, Heidelberg (1989)
4. Fernau, H., Freund, R., Oswald, M., Reinhardt, K.: Refining the nonterminal complexity of graph-controlled, programmed, and matrix grammars. J. Autom. Lang. Comb. **12**(1/2), 117–138 (2007)
5. Fernau, H., Kuppusamy, L., Oladele, R.O.: New nonterminal complexity results for semi-conditional grammars. Accepted at Computability in Europe, CiE 2018 (2018)
6. Gazdag, Z., Tichler, K.: On the power of permitting semi-conditional grammars. In: Charlier, É., Leroy, J., Rigo, M. (eds.) DLT 2017. LNCS, vol. 10396, pp. 173–184. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62809-7_12
7. Geffert, V.: How to generate languages using only two pairs of parentheses. J. Inf. Process. Cybern. EIK **27**(5/6), 303–315 (1991)
8. Geffert, V.: Normal forms for phrase-structure grammars. RAIRO Informatique théorique et Applications/Theor. Inf. Appl. **25**, 473–498 (1991)
9. Ginsburg, S., Rice, H.G.: Two families of languages related to ALGOL. J. ACM **9**(3), 350–371 (1962)
10. Masopust, T.: Formal models: regulation and reduction. Ph.D. thesis, Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic (2007)
11. Masopust, T.: A note on the generative power of some simple variants of context-free grammars regulated by context conditions. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 554–565. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00982-2_47
12. Masopust, T., Meduna, A.: Descriptional complexity of generalized forbidding grammars. In: Geffert, V., Pighizzini, G. (eds.) 9th International Workshop on Descriptional Complexity of Formal Systems - DCFS, pp. 170–177. University of Kosice, Slovakia (2007)
13. Mayer, O.: Some restrictive devices for contrext-free languages. Inf. Control (now Inf. Comput.) **20**, 69–92 (1972)
14. Meduna, A., Gopalaratnam, M.: On semi-conditional grammars with productions having either forbidding or permitting conditions. Acta Cybern. **11**, 309–323 (1994)
15. Meduna, A., Svec, M.: Reduction of simple semi-conditional grammars with respect to the number of conditional productions. Acta Cybern. **15**(3), 353–360 (2002)
16. Okubo, F.: A note on the descriptional complexity of semi-conditional grammars. Inf. Process. Lett. **110**(1), 36–40 (2009)
17. Oladele, R.O., Isah, S.N.: Reduction of simple semi-conditional grammars. Pac. J. Sci. Technol. **18**(1), 112–116 (2017)
18. Vaszil, G.: On the descriptional complexity of some rewriting mechanisms regulated by context conditions. Theor. Comput. Sci. **330**, 361–373 (2005)

Minimal Useful Size of Counters for (Real-Time) Multicounter Automata

Viliam Geffert and Zuzana Bednárová(✉)

Department of Computer Science, P. J. Šafárik University,
Jesenná 5, 04154 Košice, Slovakia
{viliam.geffert,zuzana.bednarova}@upjs.sk

Abstract. We show that, for nondeterministic and alternating machines with weak space bounds, the minimal space that is required for accepting a nonregular language by real-time or one-way multicounter automata is $(\log n)^{\varepsilon}$. The same space is required for two-way multicounter automata, independent of whether they are deterministic, nondeterministic, or alternating, and of whether they work with strong or weak space bounds. On the other hand, for deterministic, nondeterministic, and alternating machines with strong space bounds, and also for deterministic machines with weak space bounds, we show that the minimal space required for accepting a nonregular language by real-time or one-way multicounter automata is n^{ε}. All these bounds hold both for unary and general nonregular languages. Here ε represents an arbitrarily small—but fixed—real positive constant; the "space" refers to the values stored in the counters, rather than to the lengths of their binary representation.

Keywords: Space complexity · Pushdown automata
Counter automata · Real-time automata

1 Introduction and Preliminaries

The minimal amount of necessary resources is one of the fundamental research directions in complexity theory. By the space hierarchy theorem [14], we know that with a small increase in space $s(n)$ we can solve new problems that could not be solved before: if a function $s_2(n)$ grows faster than $s_1(n)$, then there exists languages that can be accepted with space bound $s_2(n)$ but not with space bound $s_1(n)$. (For more details and an advanced version, we refer the reader to [8].) This holds both for *(i) strong space bound s(n)* that refers to the space used by any computation path, on all inputs of length n, and for *(ii) weak space bound s(n)* that refers to the minimal space that is required for an accepting computation path, for all accepted inputs of length n. (Some other variants of space complexity have also been considered in the literature, see [11,17,19].)

However, there is a gap between $s_2(n) = \log\log n$ and $s_1(n) = 0$: each language accepted with space below $\log\log n$ is necessarily regular, and hence

Supported under contracts VEGA 1/0056/18 and APVV-15-0091.

accepted with no requirements on the worktape space.[1] For this reason, a worktape the size of which is bounded by $o(\log\log n)$ is not useful. This result was gradually improved, beginning with two-way deterministic Turing machines with strong space bounds and ending by an argument for two-way alternating machines with weak space bounds [1,14–16]. The first nonregular language accepted by a two-way deterministic machine using this minimal useful space, i.e., with strong space bound $O(\log\log n)$, appeared already in [14].

Moreover, unary languages need a special attention, since they may require resources that are different from those for languages built over general (or binary) alphabets [3,4,11,17]. This is because a recognizer, already having too little space to remember an input head position, must also cope with the lack of any structure along the input. The first *unary* nonregular language accepted deterministically with strong space bound $O(\log\log n)$ was presented in [2]. The two-way alternating machines using $O(\log\log n)$ space can actually be quite strong, e.g., they can recognize some unary languages the binary coded versions of which are PSPACE-complete [10].

It turns out that the minimal space bounds for *one-way* machines accepting nonregular languages are different, namely [1,14]: $\log n$ for deterministic, nondeterministic, and alternating machines with strong space bounds, and also for deterministic machines with weak space bounds, but $\log\log n$ for nondeterministic and alternating machines with weak space bounds.

Also in the one-way case we do have unary nonregular witness languages matching these lower bounds: the language $\{1^p : p \text{ is a prime}\}$ can be accepted by a one-way deterministic Turing machine with strong space bound $O(\log n)$ [19, Sect. 3.1] and the language $\mathcal{L}$ introduced by (2) in Sect. 2, by a one-way nondeterministic Turing machine with weak space bound $O(\log\log n)$ [4] (see [17]).

The minimal useful space resources for one-way machines do not change even if we require a computation in *real-time*, by machines that move the input head forward in each computation step. Clearly, all lower bounds presented above for one-way machines must also hold for real-time machines. Second, these bounds cannot be raised up: the unary nonregular witness language accepted by a real-time deterministic (hence, also nondeterministic or alternating) Turing machine with strong space bound $O(\log n)$ appeared in [20], the unary nonregular language accepted by a real-time nondeterministic (hence, also alternating) machine with weak space bound $O(\log\log n)$ in [3].

Taking into account that the above bounds cannot be decreased, a natural question arises, namely, if we cannot improve the results by the use of even simpler computational models. For example, a machine may use a simpler kind of memory, like a pushdown store or a finite number of counters.

Several results should be mentioned. First [3], some nonregular languages can be accepted by two-way deterministic pushdown automata with strong space bound $O(\log\log n)$, and also by real-time nondeterministic pushdown automata with weak space bound $O(\log\log n)$. Thus, using a pushdown store instead of a worktape does not increase the minimal useful space. Since each unary

[1] Throughout the paper, $\log x$ denotes the *binary* logarithm of x.

context-free language is regular [13], only unary regular languages are accepted by one-way nondeterministic pushdown automata, but their alternating counterparts can simulate any alternating machine that uses linear space [5].

Consider now one-way machines using one counter instead of a pushdown store. We have a real-time alternating automaton recognizing a unary nonregular language by the use of one counter with weak space bound $O(\log n)$ [3], but only unary regular languages are accepted by one-way nondeterministic machines using one counter, by argument that all unary context-free languages are regular.

The primary computational model studied in this paper is a real-time automaton recognizing a unary language by the use of a finite number of counters, but several results easily extended to more powerful models.

First, we present a unary nonregular language that can be accepted by a real-time nondeterministic automaton using four counters with weak space bound $O(\log n)$. Then, by increasing the number of counters—but keeping the real-time processing of the input—we reduce the weak space bound for this language to $O((\log n)^{\varepsilon})$, where ε represents an arbitrarily small, but fixed, real positive constant. Next, we show that this upper bound cannot be decreased: with weak space bound $(\log n)^{o(1)}$, even two-way alternating (hence, also real-time nondeterministic) multicounter automata can recognize only regular languages. This gives an answer to a question stated in [20]—namely, we have shown that real-time nondeterministic and two-way alternating multicounter automata need the *same* amount of useful space (i.e., the alternation does not help here, even if a two-way head is available). This clearly carries over to all models with computational power in between.

For completeness, we also show that $O((\log n)^{\varepsilon})$ is the minimal useful space for recognizing unary nonregular languages by two-way multicounter automata, independent of whether they are deterministic, nondeterministic, or alternating, and of whether they work with strong or weak space bounds.

Finally, we present a unary nonregular language accepted by a real-time deterministic automaton using two counters with strong space bound $O(n^{1/2})$. By increasing the number of counters while keeping the real-time processing, we reduce the strong space bound for this language to $O(n^{\varepsilon})$. Also here the achieved bound cannot be decreased to $n^{o(1)}$, neither for one-way alternating (hence, neither for deterministic or nondeterministic) multicounter machines with strong space bounds, nor for one-way deterministic multicounter machines with weak space bounds. This improves [20, Theorem 11] that presents, for each $j > 1$, a nonregular language recognized by a real-time deterministic automaton using j counters with strong space bound $O(n^{1/j})$. This result did not give a *single* witness language accepted with space $O(n^{\varepsilon})$ at all but a sequence of languages with decreasing space bounds and the size of input alphabets growing in j.

The bounds obtained in this paper are summarized in Table 2.

We assume the reader is familiar with the standard models of finite state automata and pushdown automata (see, e.g., [12,15,19]).

A *nondeterministic multicounter automaton* is a nondeterministic machine equipped with a finite state control, a read-only input tape, and, for some $\ell \geq 0$, with ℓ *counters*, containing initially zeros. The set of operations with a counter $\mathcal{C}$ consists of testing its contents for zero ($\mathcal{C} \overset{?}{=} 0$), increasing by one ($\mathcal{C} := \mathcal{C} + 1$), decreasing by one ($\mathcal{C} := \mathcal{C} - 1$), and no change ($\mathcal{C} := \mathcal{C}$). The action of the multicounter machine depends on the current state, the symbol currently scanned by the input head, and which of the counters contain/do not contain zeros. In one step, the machine changes its state, moves its head at most one position along the input tape (forward, backward, or no move), and updates its counters, independently (increase, decrease, or no change). The computation is aborted, if the machine tries to decrease a counter containing zero.[2] Under the space used by counter we mean the value stored in the counter.

A multicounter automaton is *one-way*, if it never moves its input head backward, otherwise it is *two-way*. A two-way machine has its input enclosed in between two endmarkers. A *real-time* machine is a restricted one-way variant in which the input head moves forward in each computation step.

As usual, a given computation path is *accepting*, if it starts in the initial state and halts in any accepting state. For one-way machines, acceptance requires halting in an accepting state after reading the entire input.

A *deterministic* machine can be obtained from nondeterministic version by claiming that there is allowed at most one possible transition at a time. An *alternating* machine is obtained from nondeterministic machine by partitioning the state set into the sets of existential and universal states, disjointedly. In the existential states, the machine chooses one from among possible executable steps, but in the universal states it follows all possible branches in parallel. (For more details, see, e.g., [1,5,7,9,19].)

2 Weak Space Bounds

Here we shall present the minimal space that is required for accepting a nonregular language by multicounter real-time automata with weak bounds on space.

We begin by constructing a real-time nondeterministic machine accepting a unary nonregular language by the use of four counters, working with weak space bound $O(\log n)$. To this aim, consider the following function:

$$f(n) = \text{the smallest positive integer not dividing } n.$$

To give an idea how the function $f(n)$ develops, Table 1 shows some values. It is well-known [2,6,19] that $f(n)$ can be bounded by $O(\log n)$. For our purposes, we shall need a more exact upper bound, derived in [3, Lemma 6]:

$$f(n) < 2\cdot \log n\,, \quad \text{for each } n \geq 3. \tag{1}$$

[2] Equivalently, the counter may be viewed as a special case of the pushdown store, the contents of which are always in the form $\vdash X^h$, where $\vdash$ denotes the bottom-of-pushdown-store-endmarker.

Table 1. Some values of function $f(n)$.

n	0	1	2	3	4	5	6	$\cdots$	12	$\cdots$	60	$\cdots$	420	$\cdots$	840	$\cdots$
$f(n)$	$+\infty$	2	3	2	3	2	4	<5	5	<7	7	<8	8	<9	9	$\cdots$

Next, consider the following unary language:

$$\mathcal{L} = \{1^n : \ f(n) \text{ is not equal to a power of } 2\}. \tag{2}$$

For the special case of $n = 0$, we have $f(n) = +\infty$ and $1^0 \in \mathcal{L}$. Historically, the *complement* of $\mathcal{L}$ was the first known *unary* nonregular language accepted with only $O(\log\log n)$ space, by a two-way deterministic Turing machine with strong space bound [2]. Later, in [4] (see also [17]), it was shown that the language $\mathcal{L}$ (but not its complement) can be accepted by a *one-way* nondeterministic Turing machine, with the same—but weak—space bound $O(\log\log n)$. Quite recently [3], this was improved by showing that $\mathcal{L}$ can also be accepted by a *real-time* nondeterministic machine, still keeping the weak space bound $O(\log\log n)$. Both in [4] and in [3], the machines for $\mathcal{L}$ are based on the observation that, for each $n > 0$,

1. $1^n \in \mathcal{L}$ if and only if there exist two positive integers k and i satisfying $2^i < k < 2^{i+1}$, such that $n \bmod k \neq 0$ and $n \bmod 2^i = 0$.
2. Moreover, if $1^n \in \mathcal{L}$ and $n \geq 3$, the membership can be certified by taking $k = f(n)$ and $2^i = 2^{\lfloor \log k \rfloor}$. This gives $2^i < k < 2 \cdot \log n$, by (1).

The conditions in (1) guarantee that $1^n \in \mathcal{L}$, the item (2) gives an upper bound on the values 2^i and k that certify the membership in $\mathcal{L}$. These properties allow us to construct a machine for a minor modification of $\mathcal{L}$:

Theorem 1. *There exists a unary nonregular language—namely, $\mathcal{L}' = \mathcal{L} \cdot \{1\}$, for $\mathcal{L}$ introduced by (2)—accepted by a real-time nondeterministic automaton using four counters with weak space bound $O(\log n)$.*

Proof. We first present a nondeterministic machine $\mathcal{A}$ for $\mathcal{L}$ accepting in a non-standard way, using a set of accepting configurations rather than the set of accepting states. Then we shall replace $\mathcal{A}$ by $\mathcal{A}'$ for $\mathcal{L}'$ that accepts by classic halting in an accepting state.

At the very beginning on the given input 1^n, our nondeterministic machine $\mathcal{A}$ for the language $\mathcal{L}$ nondeterministically chooses between $n \leq 6$ and $n \geq 7$.

Short inputs with $n \leq 6$ are solved without using the counters at all.

Consider now the case of $n \geq 7$. The recognition is based on the facts presented by the items (1) and (2) above. That is, the machine $\mathcal{A}$ nondeterministically guesses an integer $k > 0$ different from a 8 power of two, takes $2^i = 2^{\lfloor \log k \rfloor}$, which ensures that $2^i < k < 2^{i+1}$, and checks whether the conditions $n \bmod k \neq 0$ and $n \bmod 2^i = 0$ are satisfied. To compute the length of the input modulo k, the machine uses two counters, denoted here by $\mathcal{C}_1$ and $\mathcal{C}_2$. Similarly, to compute n modulo 2^i, we use another two counters, $\mathcal{C}_3$ and $\mathcal{C}_4$.

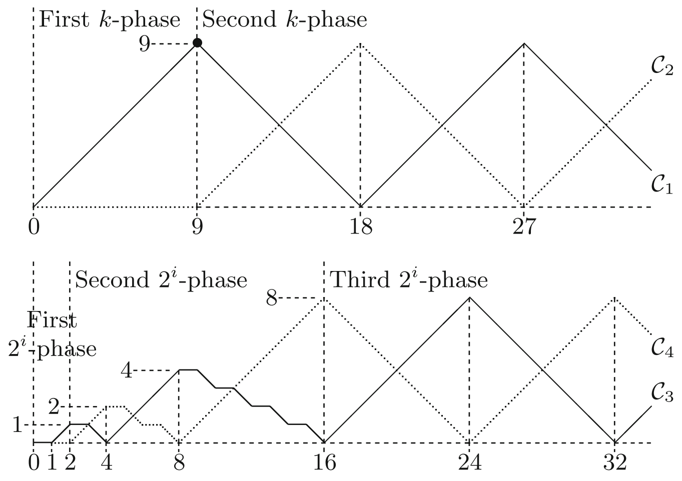


Fig. 1. The counters in the course of a computation that guesses $k = 9$, with $2^i = 8$.

Starting with counters that are all empty the main problem is the initial assignment of values k and 2^i to the counters $\mathcal{C}_1$ and $\mathcal{C}_3$, respectively, in such a way that $2^i < k < 2^{i+1}$. Recall that $\mathcal{A}$ should be a *real-time* machine and should move its input head forward *in each computation step.* To make this tricky procedure easier to follow, we shall describe manipulation with $\mathcal{C}_1, \mathcal{C}_2$ and $\mathcal{C}_3, \mathcal{C}_4$ separately. The reader is referred to Fig. 1.

First, let us describe the work with the counters $\mathcal{C}_1$ and $\mathcal{C}_2$ which are used to guess $k > 0$ and to compute the length of the input modulo k. With respect to k, the computation is divided into two phases (see also Fig. 1, top):

First k-phase: Moving k symbols to the right along the input, $\mathcal{A}$ guesses an integer $k > 0$ and saves this value in $\mathcal{C}_1$. More precisely:

– In each step, reading one symbol from the input, $\mathcal{A}$ increases the counter $\mathcal{C}_1$; the counter $\mathcal{C}_2$ does not change. This is repeated in a loop, the moment of leaving this loop is chosen nondeterministically.

That is, in each step, $\mathcal{A}$ nondeterministically chooses between carrying on and leaving the first k-phase.[3] Clearly, at the moment when $\mathcal{A}$ decides to leave this phase, $\mathcal{C}_1 = k$ (the number of input symbols read so far) and $\mathcal{C}_2 = 0$.

Second k-phase: From this moment on, the counters $\mathcal{C}_1$ and $\mathcal{C}_2$ are used to compute the length of the input modulo k. This is quite straightforward, $\mathcal{A}$ switches between the following two sweep modes:

– *Odd sweep*: reading one input symbol, $\mathcal{A}$ decreases $\mathcal{C}_1$ and increases $\mathcal{C}_2$. When $\mathcal{C}_1$ becomes empty, $\mathcal{A}$ switches to even-sweep mode below.

[3] An important detail is that leaving this loop is disabled, whenever $\mathcal{C}_3 = 0$ or $\mathcal{C}_4 = 0$. This ensures that we cannot choose k be equal to a power of two—to be described later.

– *Even sweep*: reading one symbol from the input, $\mathcal{A}$ decreases $\mathcal{C}_2$ and increases $\mathcal{C}_1$. When $\mathcal{C}_2 = 0$, $\mathcal{A}$ switches to odd-sweep mode.

Thus, we keep $\mathcal{C}_1 + \mathcal{C}_2 = k$, which ensures counting the length of the input 1^n modulo k. Clearly, when the end of the input is reached, we have $\mathcal{C}_1 \neq 0 \wedge \mathcal{C}_2 \neq 0$ if and only if $n \bmod k \neq 0$.

Second, consider the work with the counters $\mathcal{C}_3$ and $\mathcal{C}_4$. These two counters are used to compute the length of the input modulo 2^i, for $2^i = 2^{\lfloor \log k \rfloor}$. Counting modulo 2^i runs in parallel with counting modulo k, described above. Among others, this means that the two procedures manipulating with $\mathcal{C}_1, \mathcal{C}_2$ and $\mathcal{C}_3, \mathcal{C}_4$ share the same input head, moving this head one position to the right in each computation step. With respect to 2^i, the computation is divided into three phases (see Fig. 1, bottom).

First 2^i*-phase*: After this initialization, we set $\mathcal{C}_3 := 1$ and $\mathcal{C}_4 := 0$:

– In the first two steps, reading the first two symbols from the input, $\mathcal{A}$ increases $\mathcal{C}_3$ from 0 to 1; the counter $\mathcal{C}_4$ does not change.

Second 2^i*-phase*: This involves the standard use of the counters $\mathcal{C}_3$ and $\mathcal{C}_4$ to multiply, repeatedly, the actual value by the factor of two. This is done by switching between the following two sweep modes:

– *Odd sweep*: reading two input symbols, $\mathcal{A}$ decreases $\mathcal{C}_3$ by 1 and increases $\mathcal{C}_4$ by 2. When $\mathcal{C}_3$ becomes empty, $\mathcal{A}$ switches to even-sweep mode below.
– *Even sweep*: reading two symbols in two steps, $\mathcal{A}$ decreases $\mathcal{C}_4$ by 1 and increases $\mathcal{C}_3$ by 2. When $\mathcal{C}_4 = 0$, $\mathcal{A}$ switches to odd-sweep mode.

Thus, starting with $\mathcal{C}_3 + \mathcal{C}_4 = 1$ after the first 2^i-phase and iterating this way i sweeps, for some $i > 0$, we get $\mathcal{C}_3 + \mathcal{C}_4 = 2^i$, with either $\mathcal{C}_3 = 0$ or $\mathcal{C}_4 = 0$, depending on parity of i.

The whole process prepares to terminate when the procedure manipulating the counters $\mathcal{C}_1$ and $\mathcal{C}_2$ (described above, running in parallel) nondeterministically decides to switch from the first k-phase to the second k-phase. This means that the final value k has been fixed at this moment. When this happens, the current sweep of the second 2^i-phase (no matter whether it is odd or even) is fixed as the last one, which is kept in the finite state control. The current sweep of the second 2^i-phase is still going to be completed, that is, we carry it on until we get $\mathcal{C}_3 = 0$ or $\mathcal{C}_4 = 0$, depending on parity of i. After that, $\mathcal{A}$ switches from the second 2^i-phase to the third 2^i-phase—to be described below. More precisely, depending on parity of i, we switch either from the odd sweep of the second 2^i-phase to the even sweep of the third 2^i-phase or, vice versa, from the even sweep of the second 2^i-phase to the odd sweep of the third 2^i-phase.

Let us calculate how many symbols are read from the input in the course of the first two 2^i-phases. Let i be the total number of sweeps iterated in the second 2^i-phase—including the one in which the final value k has been fixed, i.e., during which $\mathcal{A}$ switches nondeterministically from the first k-phase to the second k-phase. It is easy to see that the second 2^i-phase reads exactly $\sum_{j=1}^{i-1} 2^j =$

2^i-2 symbols in the first $i-1$ sweeps and exactly $\sum_{j=1}^{i} 2^j = 2^{i+1}-2$ symbols in the first i sweeps. Since, by Footnote 3, the machine $\mathcal{A}$ cannot switch from the first k-phase to the second k-phase whenever $\mathcal{C}_3 = 0$ or $\mathcal{C}_4 = 0$, the final value k must be fixed *in the middle* of the i-th sweep of the second 2^i-phase. Taking into account that exactly 2 symbols were read during the first 2^i-phase, we get:

$$2^i < k < 2^{i+1}. \tag{3}$$

This implies that the input tape segment traversed in the course of the first two 2^i-phases is of length 2^{i+1}. Using (3), it is also easy to see that $82^i = 2^{\lfloor \log k \rfloor}$.

Third 2^i-phase: From this moment on, the counters $\mathcal{C}_3$ and $\mathcal{C}_4$ are used to compute the length of the input modulo 2^i. Since we have traversed 2^{i+1} input symbols in the first two 2^i-phases, which is an integer multiple of 2^i, and the second 2^i-phase ends with $\mathcal{C}_3 + \mathcal{C}_4 = 2^i$, with either $\mathcal{C}_3 = 0$ or $\mathcal{C}_4 = 0$ (depending on parity of i), counting modulo 2^i can be implemented in the same way as in the second k-phase, using $\mathcal{C}_3, \mathcal{C}_4$ instead of $\mathcal{C}_1, \mathcal{C}_2$:

- *Odd sweep*: reading one symbol from the input, $\mathcal{A}$ decreases $\mathcal{C}_3$ and increases $\mathcal{C}_4$. When $\mathcal{C}_3 = 0$, $\mathcal{A}$ switches to even-sweep mode below.
- *Even sweep*: reading one symbol from the input, $\mathcal{A}$ decreases $\mathcal{C}_4$ and increases $\mathcal{C}_3$. When $\mathcal{C}_4 = 0$, $\mathcal{A}$ switches to odd-sweep mode.

Thus, at the end of the input, $\mathcal{C}_3 = 0 \vee \mathcal{C}_4 = 0$ if and only if $n \bmod 2^i = 0$.

Finally, by definition, $\mathcal{A}$ accepts if it reaches the end of the input

- in the second k-phase with $\mathcal{C}_1 \neq 0 \wedge \mathcal{C}_2 \neq 0$ and, at the same time, in the third 2^i-phase with $\mathcal{C}_3 = 0 \vee \mathcal{C}_4 = 0$.

The above nondeterministic machine $\mathcal{A}$ accepts an input 1^n if and only if $1^n \in \mathcal{L}$. However, $\mathcal{A}$ accepts in a somewhat nonstandard way, by getting to an accepting configuration (situation) rather than to an accepting state. Now we replace $\mathcal{A}$ by a new machine $\mathcal{A}'$ that simulates $\mathcal{A}$ but,

- each time $\mathcal{A}$ gets to an accepting situation, i.e., each time $\mathcal{A}$ is in the second k-phase with $\mathcal{C}_1 \neq 0 \wedge \mathcal{C}_2 \neq 0$ and, at the same time, in the third 2^i-phase with $\mathcal{C}_3 = 0 \vee \mathcal{C}_4 = 0$, the machine $\mathcal{A}'$ nondeterministically decides whether to carry on the simulation or to stop by reading one more symbol from the input and switching to its unique final state. The same applies to the special path handling short inputs of length $n \leq 6$.

This changes $\mathcal{A}$ for $\mathcal{L}$ to $\mathcal{A}'$ that accepts $\mathcal{L}' = \mathcal{L}\cdot\{1\}$ by classic halting in a unique accepting state at the end of the input. □

As the next step, we shall show that the space used in Theorem 1 can be decreased to $O((\log n)^{\varepsilon})$, where ε represents an arbitrarily small, but fixed, real positive constant, but using more than four counters. To this aim, let us show that we can save a substantial amount of space by simulating one counter with the help of two counters, preserving the real-time processing of the input.

Theorem 2. *Each automaton $\mathcal{A}$ using a counter $\mathcal{C}$ the space of which is bounded by $s(n)$—not excluding that $\mathcal{A}$ may also utilize some other computational resources—can be replaced by an equivalent automaton $\mathcal{A}'$ utilizing the same computational resources as $\mathcal{A}$ but, instead of the counter $\mathcal{C}$, it uses two counters $\mathcal{C}_1$ and $\mathcal{C}_2$, both of them with space bound $s'(n) \leq O(s(n)^{1/2})$. Moreover, if $\mathcal{A}$ is a real-time machine, so is $\mathcal{A}'$.*

Proof. A counter $\mathcal{C}$ with space bound $s(n)$ can be simulated by two counters $\mathcal{C}_1, \mathcal{C}_2$ with space bound $O(s(n)^{1/2})$, based on the one-to-one correspondence between $\mathbb{N}$, the set of natural numbers, and $\mathbb{N} \times \mathbb{N}$. (See, e.g., [19, Theorem 3.2.3].) The main problem to be fixed here is making such simulation real-time: each single-step operation with $\mathcal{C}$ should be simulated in one step, since the new machine $\mathcal{A}'$ is forced to move its input head forward in each computation step.

The current value in the counter $\mathcal{C}$ of the original machine $\mathcal{A}$ will be represented by three quantities, namely, by two values stored in the counters $\mathcal{C}_1, \mathcal{C}_2$, and by a sweep mode $s \in \{\mathsf{even}, \mathsf{odd}\}$, kept in the finite state control. This doubles the number of finite control states of the original machine. Initially, $\mathcal{A}'$ starts in even sweep mode and both counters are empty,[4] that is, $s = \mathsf{even}$, $\mathcal{C}_1 = 0$, and $\mathcal{C}_2 = 0$. All other computational resources of the original machine $\mathcal{A}$—including its current final control state and its head along the input tape—are manipulated in a straightforward way.

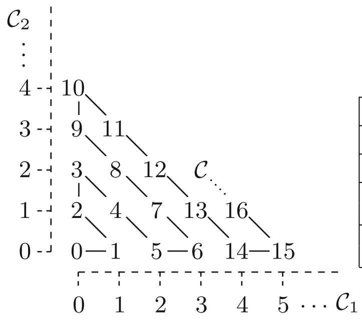
The main idea behind this implementation of a counter is simple: when $\mathcal{A}'$ simulates several operations $\mathcal{C} := \mathcal{C} + 1$ in a row, it imitates a sweep along a line in a two-dimensional grid. In the course of this sweep, the current coordinates in the grid are given by $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle$ and satisfy, for some $v \geq 0$, the condition $\mathcal{C}_1 + \mathcal{C}_2 = v$. When, sweeping along this line, $\mathcal{A}'$ hits either of the axes of the coordinate system (that is, when $\mathcal{C}_1 = 0$ or $\mathcal{C}_2 = 0$), the machine switches the sweep mode, from even to odd or vice versa. After that, $\mathcal{A}'$ starts a sweep that goes to the other axis in the grid along a new line, keeping this time a new invariant, $\mathcal{C}_1 + \mathcal{C}_2 = v + 1$ instead of $\mathcal{C}_1 + \mathcal{C}_2 = v$. The operation $\mathcal{C} := \mathcal{C} - 1$ is implemented as backing up along this zigzag trajectory towards the origin $\langle 0, 0 \rangle$. The point $\langle \mathcal{C}_1, \mathcal{C}_2 \rangle = \langle 0, 0 \rangle$ corresponds to $\mathcal{C} = 0$. Figure 2 (left) reflects the main idea, Fig. 2 (right) displays a more detailed transition table.

Both $\mathcal{C}_1$ and $\mathcal{C}_2$ are bounded by $\lceil (2 \cdot \mathcal{C})^{1/2} \rceil \leq O(s(n)^{1/2})$. It is also easy to see that one step of the original machine is simulated by exactly one step. □

By the use of the previous theorem, we can decrease the space requirements for arbitrarily many counters, down to $O(s(n)^{1/2})$, preserving the real-time processing of the input.

Lemma 3. *For each $\ell \geq 0$, each automaton $\mathcal{A}$ using ℓ counters with space bound $s(n)$ can be replaced by an equivalent automaton $\mathcal{A}'$ using $2 \cdot \ell$ counters with space bound $s'(n) \leq O(s(n)^{1/2})$. This holds for all computational models listed in the statement of Theorem 2. In particular, if $\mathcal{A}$ is a real-time machine, so is $\mathcal{A}'$.*

[4] Throughout the entire computation, $s = \mathsf{even}$ if and only if $\mathcal{C}_1 + \mathcal{C}_2$ is even.



$\mathcal{C} := \mathcal{C}+1$					
If			Then		
s	$\mathcal{C}_1$	$\mathcal{C}_2$	s	$\mathcal{C}_1$	$\mathcal{C}_2$
even	$*$	$>$	even	$+$	$-$
		$=$	odd	$+$	$\circ$
odd	$>$	$*$	odd	$-$	$+$
	$=$		even	$\circ$	$+$

$\mathcal{C} := \mathcal{C}-1$					
If			Then		
s	$\mathcal{C}_1$	$\mathcal{C}_2$	s	$\mathcal{C}_1$	$\mathcal{C}_2$
even	$>$	$*$	even	$-$	$+$
	$=$	$>$	odd	$\circ$	$-$
odd	$*$	$>$	odd	$+$	$-$
	$>$	$=$	even	$-$	$\circ$

Fig. 2. A real-time simulation of one counter by two smaller counters.

Proof. Let $\mathcal{A}$ be a machine equipped with ℓ counters, denoted here by $\mathcal{C}_1, \ldots, \mathcal{C}_\ell$. Each of these counters works with space bound $s(n)$. Now, by repeated application of Theorem 2, for $i = 1, \ldots, \ell$, we can replace the i-th counter $\mathcal{C}_i$ by a pair of new counters, $\mathcal{C}'_i$ and $\mathcal{C}''_i$. This also requires to keep a sweep mode $s_i \in \{\mathsf{even}, \mathsf{odd}\}$ in the finite state control. Both $\mathcal{C}'_i$ and $\mathcal{C}''_i$ work with space bound $s'(n) \leq O(s(n)^{1/2})$. Thus, for each $i \in \{1, \ldots, \ell\}$, the intermediate machine uses the counters $\mathcal{C}_{i+1}, \ldots, \mathcal{C}_\ell$ working with space bound $s(n)$ together with the counters $\mathcal{C}'_1, \ldots, \mathcal{C}'_i$ and $\mathcal{C}''_1, \ldots, \mathcal{C}''_i$ working with space bound $s'(n)$. In addition, the intermediate machine manipulates all sweep modes $s_1, \ldots, s_i \in \{\mathsf{even}, \mathsf{odd}\}$ in the finite state control, simultaneously. In the end, for $i = \ell$, we obtain an equivalent machine $\mathcal{A}'$ using $2{\cdot}\ell$ counters, all of them working with space bound $s'(n) \leq O(s(n)^{1/2})$. The price we pay is that $\mathcal{A}'$ uses $\|Q'\| = 2^\ell{\cdot}\|Q\|$ states, where $\|Q\|$ denotes the number of states in the original machine $\mathcal{A}$. □

The space reduction presented in Lemma 3 can be improved, by repeating the application of this lemma h times:

Lemma 4. *For each $h \geq 0$ and each $\ell \geq 0$, each automaton $\mathcal{A}$ using ℓ counters with space bound $s(n)$ can be replaced by an equivalent automaton $\mathcal{A}'$ using $2^h{\cdot}\ell$ counters with space bound $s'(n) \leq O(s(n)^{1/2^h})$. This holds for all computational models listed in the statement of Theorem 2. In particular, if $\mathcal{A}$ is a real-time machine, so is $\mathcal{A}'$.*

By taking $h = \lceil \log(1/\varepsilon) \rceil$ in the above lemma, we then get:

Theorem 5. *For arbitrarily small, but fixed, real constant $\varepsilon > 0$, each automaton $\mathcal{A}$ using ℓ counters with space bound $s(n)$ can be replaced by an equivalent automaton $\mathcal{A}'$ using $\ell' < 2\ell/\varepsilon$ counters with space bound $s'(n) \leq O(s(n)^\varepsilon)$. This holds for all computational models listed in the statement of Theorem 2. In particular, if $\mathcal{A}$ is a real-time machine, so is $\mathcal{A}'$.*

We are now ready to establish one of the main results, showing that $(\log n)^\varepsilon$ is the smallest possible weak space bound for accepting unary nonregular languages by multicounter real-time automata:

Theorem 6. *There exists a unary nonregular language—namely, $\mathcal{L}' = \mathcal{L}\cdot\{1\}$, for $\mathcal{L}$ introduced by (2)—such that, for arbitrarily small, but fixed, real constant $\varepsilon > 0$, it can be accepted by a real-time nondeterministic automaton using $\ell' < 8/\varepsilon$ counters with weak space bound $O((\log n)^{\varepsilon})$.*

Proof. From Theorem 1 we know the language $\mathcal{L}'$ is accepted by a real-time nondeterministic automaton using four counters with weak space bound $O(\log n)$. By Theorem 5 we get, for each $\varepsilon > 0$, an equivalent real-time nondeterministic automaton using $\ell' < 8/\varepsilon$ counters with weak space bound $O((\log n)^{\varepsilon})$. □

The above upper bound cannot be decreased. That is, the constant $\varepsilon > 0$ in $O((\log n)^{\varepsilon})$ cannot be replaced by a function $r(n)$ satisfying $\lim_{n\to\infty} r(n) = 0$, even if we use a more powerful computational model—utilizing the power of alternation and/or two-way input head movement, and even if the witness nonregular language is quite arbitrary—not necessarily unary:

Theorem 7. *If a two-way alternating automaton recognizes a nonregular language, using a finite number of counters with weak space bound $s(n)$, then $s(n) \notin (\log n)^{o(1)}$.*

Proof. Suppose that some nonregular language is accepted by a two-way alternating automaton using some ℓ counters with weak space bound $s(n)$. Such machine can be replaced by an equivalent two-way Turing machine using one worktape, keeping the contents of all counters in binary, separated by a special symbol. The total length of this worktape can be bounded by $s'(n) \leq \ell\cdot(2 + \log s(n)) = \log((4\cdot s(n))^{\ell})$. By [16], if a two-way alternating Turing machine accepts a nonregular language with weak space bound $s'(n)$, then $s'(n) \notin o(\log\log n)$. Hence there exist a real constant $e_0 > 0$ and an infinite sequence of input lengths $n_1 < n_2 < n_3 < \cdots$ such that, for each $i \geq 1$, we have $s'(n_i) \geq e_0\cdot\log\log n_i = \log((\log n_i)^{e_0})$. But then $\log((4\cdot s(n_i))^{\ell}) \geq \log((\log n_i)^{e_0})$, and $s(n_i) \geq \frac{1}{4}\cdot(\log n_i)^{e_0/\ell}$, for infinitely many input lengths. Thus, $s(n) \notin (\log n)^{o(1)}$. □

We point out that the constant $e_0 > 0$ in the proof of the above theorem (given by the proof in [16]) is smaller than one and makes our lower bound dependent also on the number of states in the original multicounter machine.

3 Two-Way Devices

This section shows that also for two-way multicounter automata the minimal useful space is $O((\log n)^{\varepsilon})$. For two-way devices, the bound is independent of whether the machines are deterministic, nondeterministic, or alternating, and of whether they work with strong or weak space bounds.

Theorem 8. *There exists a unary nonregular language—namely, $\mathcal{L}$ introduced by (2)—accepted by a two-way deterministic automaton using two counters with strong space bound $O(\log n)$.*

Proof. On the given input 1^n, for $n > 0$, our two-way machine $\mathcal{A}$ equipped with the counters $\mathcal{C}_1$ and $\mathcal{C}_2$ runs a loop for $k = 2, 3, 4, \ldots$, until it finds the first k not dividing n. In order to check whether a number k divides n, the machine traverses across the entire input, counting modulo k in the same way as in the second k-phase in the proof of Theorem 1. Thus, a traversal starts from one of the endmarkers with $\mathcal{C}_1 + \mathcal{C}_2 = k$ and with either $\mathcal{C}_1 = 0$ or $\mathcal{C}_2 = 0$. If k divides n, the machine reaches the opposite endmarker with either $\mathcal{C}_1 = 0$ or $\mathcal{C}_2 = 0$ again. After that, $\mathcal{A}$ increases $\mathcal{C}_1 + \mathcal{C}_2$ from k to $k + 1$ and starts a new traversal across the input in the opposite direction. This is repeated until $\mathcal{A}$ gets to the opposite endmarker with $\mathcal{C}_1 > 0 \wedge \mathcal{C}_2 > 0$.

When this happens, $\mathcal{C}_1 + \mathcal{C}_2 = k = f(n)$. From this moment on, the input head does not move—parked at one of the endmarkers. It only remains to check whether $k = \mathcal{C}_1 + \mathcal{C}_2$ is not equal to a power of 2. This is quite simple; we first move the contents of $\mathcal{C}_2$ to $\mathcal{C}_1$ by decreasing $\mathcal{C}_2$ and increasing $\mathcal{C}_1$, until we get $\mathcal{C}_2 = 0$ and $\mathcal{C}_1 = k$. Next, we divide the contents of $\mathcal{C}_1$ by two. Thus, in a loop, we decrease $\mathcal{C}_1$ by 2 and increase $\mathcal{C}_2$ by 1. When the counter $\mathcal{C}_1$ reaches zero, the original value in $\mathcal{C}_1$ has been halved, but now it is stored in $\mathcal{C}_2$. By swapping the roles of $\mathcal{C}_1$ and $\mathcal{C}_2$, we can halve this value again, ending this time with a result stored in $\mathcal{C}_1$. This halving is repeated until we find out that we have tried to halve a value that is odd. Then, $\mathcal{A}$ accepts if this last value was greater than one, i.e., the last integer division by two must not end with empty counters.

The trivial case of $1^0 \in \mathcal{L}$ is resolved at the very beginning: $\mathcal{A}$ verifies whether $n = 0$ by checking whether the first symbol to the left of the left endmarker is the right endmarker.

Clearly, by (1), the counters are bounded by $k = f(n) \leq O(\log n)$. □

By applying Theorem 5, we then get:

Theorem 9. *There exists a unary nonregular language—namely, $\mathcal{L}$ introduced by (2)—such that, for arbitrarily small, but fixed, real constant $\varepsilon > 0$, it can be accepted by a two-way deterministic automaton using $\ell' < 4/\varepsilon$ counters with strong space bound $O((\log n)^{\varepsilon})$.*

Also this upper bound cannot be decreased since, by Theorem 7, even a two-way alternating multicounter automaton with weak space bound $s(n) \in (\log n)^{o(1)}$ recognizes only a regular language.

4 Strong Space Bounds and/or Determinism

Here we present the corresponding minimal space that is required for accepting nonregular languages by real-time multicounter automata with strong bounds on space, and also for real-time deterministic machines with weak bounds on space. It turns out that, for these computational models, the corresponding minimal space is n^{ε}. To this aim, consider the following language:

$$\mathcal{L}'' = \{1^{(n_1+n_2)\cdot(n_1+n_2+1)/2+n_1} : n_1, n_2 \in \mathbb{N},\ (n_1+n_2) \bmod 2 = 0\}. \quad (4)$$

Theorem 10. *There exists a unary nonregular language—namely, $\mathcal{L}''$ introduced by (4)—accepted by a real-time deterministic automaton using two counters with strong space bound $O(n^{1/2})$.*

Proof. Our machine $\mathcal{A}$ traverses across the entire input and counts its length. The operation $\mathcal{C} := \mathcal{C}+1$ is simulated by the use of two counters, $\mathcal{C}_1$, and $\mathcal{C}_2$, in the same way as in Theorem 2. This also requires to keep a sweep mode $s \in \{\mathsf{even}, \mathsf{odd}\}$ in the finite state control. By definition, let even be the only initial and, at the same time, the only final state of $\mathcal{A}$.

Clearly, $\mathcal{A}$ accepts if and only if it halts with $s = \mathsf{even}$ after reading the entire input, with any values $\mathcal{C}_1 \geq 0$ and $\mathcal{C}_2 \geq 0$ in the counters. But (see also Footnote 4) this configuration is reached for each $\mathcal{C}_1, \mathcal{C}_2 \in \mathbb{N}$ such that $\mathcal{C}_1 + \mathcal{C}_2$ is even. This condition gives $\delta = \mathcal{C}_1$ and hence the desired configuration is reached after reading $\mathcal{C}$ symbols from the input, where

$$\mathcal{C} = \textstyle\sum_{v=0}^{\mathcal{C}_1+\mathcal{C}_2-1}(v+1) + \delta = \sum_{v=1}^{\mathcal{C}_1+\mathcal{C}_2} v + \mathcal{C}_1 = (\mathcal{C}_1+\mathcal{C}_2)\cdot(\mathcal{C}_1+\mathcal{C}_2+1)/2 + \mathcal{C}_1. \quad (5)$$

Thus, $\mathcal{A}$ is a real-time deterministic machine accepting $\mathcal{L}''$. Moreover, as shown in the proof of Theorem 2, we have $\mathcal{C}_1 + \mathcal{C}_2 < \lceil (2\cdot\mathcal{C})^{1/2} \rceil = \lceil (2\cdot n)^{1/2} \rceil \leq O(n^{1/2})$.

In only remains to show that $\mathcal{L}''$ accepted by $\mathcal{A}$ is not regular. First, for any given $h > 0$, take $\mathcal{C}_1 = 2\cdot h$ and $\mathcal{C}_2 = 0$. Since $\mathcal{C}_1 + \mathcal{C}_2 = 2\cdot h$ is even, the machine $\mathcal{A}$ reaches these two values in the counters with $s = \mathsf{even}$, after reading $\mathcal{C} = h\cdot(2h{+}3)$ symbols from the input—this value $\mathcal{C}$ is obtained by using $\mathcal{C}_1 = 2\cdot h$ and $\mathcal{C}_2 = 0$ in (5). Thus, for each $h > 0$, the input $1^{h\cdot(2h+3)}$ is accepted but, since $s = \mathsf{even}$ and $\mathcal{C}_2 = 0$, the procedure presented in the proof of Theorem 2 switches the sweep mode to odd after reading one more symbol from the input. Moreover, the sweep mode will not change in the subsequent $\mathcal{C}_1 + \mathcal{C}_2 = 2\cdot h$ steps, and hence $\mathcal{A}$ rejects $1^{h\cdot(2h+3)+i}$, for each $i = 1, \ldots, 2\cdot h + 1$. Consequently, this language cannot be accepted by a deterministic finite state automaton with fewer than $2\cdot h + 2$ states, since, after getting into a cycle the length which is bounded by $2\cdot h + 1$, such automaton cannot accept one input and then reject the next $2\cdot h + 1$ inputs in a row. Since $h > 0$ can be chosen arbitrarily large, this rules out all finite state automata. □

Also here the space requirements can be decreased by using more counters, by application of Theorem 5 on the machine constructed in Theorem 10:

Theorem 11. *There exists a unary nonregular language—namely, $\mathcal{L}''$ introduced by (4)—such that, for arbitrarily small, but fixed, real constant $\varepsilon > 0$, it can be accepted by a real-time deterministic automaton using $\ell' < 4/\varepsilon$ counters with strong space bound $O(n^{\varepsilon})$.*

The above upper bound is optimal and cannot be decreased for one-way machines with strong space bounds, even if we use the power of alternation. The same holds for one-way deterministic machines, even if we use a less restrictive weak space bound, not taking into account space used on inputs that are rejected:

Table 2. Minimal space used by multicounter automata accepting nonregular languages. All these bounds are tight both for unary and general languages.

	Strong[a]	Weak[a]	Two-way[b]
Deterministic	n^{ε}	n^{ε}	$(\log n)^{\varepsilon}$
Nondeterministic	n^{ε}	$(\log n)^{\varepsilon}$	$(\log n)^{\varepsilon}$
Alternating	n^{ε}	$(\log n)^{\varepsilon}$	$(\log n)^{\varepsilon}$

[a]Both real-time and one-way
[b]Both strong and weak

Theorem 12. *If a one-way alternating automaton recognizes a nonregular language, using a finite number of counters with strong space bound $s(n)$, then $s(n) \notin n^{o(1)}$. The same holds for weak space bounds in the case of one-way deterministic multicounter machines.*

Proof. The known lower bounds for accepting nonregular languages state that $s'(n) \notin o(\log n)$ both for one-way alternating Turing machines with strong space bounds and for one-way deterministic Turing machines with weak space bounds ([1,14], [19, Sect. 5.2]). The rest of argument mirrors the proof of Theorem 7, using n instead of $\log n$ the lower bound $s'(n) \notin o(\log n)$ for one-way Turing machines gives us the lower bound $s(n) \notin n^{o(1)}$ for one-way multicounter machines. □

5 Concluding Remarks

We have studied the minimal useful space that is required for recognizing nonregular languages by automata using a finite number of counters. The primary computational model was a real-time machine recognizing a unary language, but several results easily extended to a more general setting. All tight bounds are summarized in Table 2. The results in this table are derived by combining the upper bounds obtained in Theorems 6, 9, and 11 with the lower bounds obtained in Theorems 7 and 12, using also some trivial relations between upper and lower bounds for weaker and stronger computational models.

Both for unary and general nonregular languages, and both for real-time and one-way multicounter machines, the tight bounds do not differ. Allowing an unrestricted number of computation steps in between two forward moves along the input does not help to decrease the minimal useful space. Neither does the use of alternation instead of nondeterminism, for the same computational model.

We conjecture that if we fix the number of counters to some constant ℓ, all bounds n^{ε} in Table 2 will change to $\Theta(n^{1/\ell})$ while $(\log n)^{\varepsilon}$ to $\Theta((\log n)^{1/\ell})$. The argument would require a more efficient use of counters in upper bounds and, at the same time, a more precise analysis of the lower bounds.

However, we cannot exclude the possibility that, with the same number of counters, alternation may become more powerful, especially for small values ℓ, below 4. For example, we know a real-time alternating automaton recognizing a unary nonregular language by the use of one counter with weak space bound $O(\log n)$ [3], but only unary regular languages are accepted by one-way nondeterministic machines using one counter. Second, it is well known that each recursively enumerable language can be accepted by a one-way deterministic automaton with two counters [18] (see also [15]), but the values in the counters are double-exponential in space used by the original Turing machine, and hence such simulation is far from being real-time. The best known upper bound for a real-time deterministic automaton recognizing a unary nonregular language by the use of two counters is $O(n^{1/2})$, presented in Theorem 10. For two-way deterministic machines using two counters, this bound drops to $O(\log n)$, in Theorem 8. It is not clear whether these bounds cannot be decreased by the use of nondeterminism. For a small fixed number of counters, it is even not clear whether the bounds required for recognizing unary and general (or binary) nonregular languages do differ.

References

1. Alberts, M.: Space complexity of alternating Turing machines. In: Budach, L. (ed.) FCT 1985. LNCS, vol. 199, pp. 1–7. Springer, Heidelberg (1985). https://doi.org/10.1007/BFb0028785
2. Alt, H., Mehlhorn, K.: A language over a one symbol alphabet requiring only $O(\log\log n)$ space. SIGACT News **7**, 31–33 (1975)
3. Bednárová, Z., Geffert, V., Reinhardt, K., Yakaryılmaz, A.: New results on the minimum amount of useful space. Int. J. Found. Comput. Sci. **27**, 259–281 (2016)
4. Bertoni, A., Mereghetti, C., Pighizzini, G.: Strong optimal lower bounds for Turing machines that accept nonregular languages. In: Wiedermann, J., Hájek, P. (eds.) MFCS 1995. LNCS, vol. 969, pp. 309–318. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-60246-1_137
5. Chandra, A., Kozen, D., Stockmeyer, L.: Alternation. J. Assoc. Comput. Mach. **28**, 114–133 (1981)
6. Freedman, A., Ladner, R.: Space bounds for processing contentless inputs. J. Comput. Syst. Sci. **11**, 118–128 (1975)
7. Geffert, V.: A hierarchy that does not collapse: alternations in low level space. RAIRO Inform. Théor. Appl. **28**, 465–512 (1994)
8. Geffert, V.: Space hierarchy theorem revised. Theor. Comput. Sci. **295**, 171–187 (2003)
9. Geffert, V.: Alternating space is closed under complement and other simulations for sublogarithmic space. Inform. Comput. **253**, 163–178 (2017)
10. Geffert, V.: Unary coded PSPACE-complete languages in ASPACE(loglog n). In: Weil, P. (ed.) CSR 2017. LNCS, vol. 10304, pp. 141–153. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58747-9_14
11. Geffert, V., Mereghetti, C., Pighizzini, G.: Sublogarithmic bounds on space and reversals. SIAM J. Comput. **28**, 325–340 (1999)
12. Geffert, V., Yakaryılmaz, A.: Classical automata on promise problems. Discrete Math. Theor. Comput. Sci. **17**, 157–180 (2015)
13. Ginsburg, S., Rice, H.: Two families of languages related to ALGOL. J. Assoc. Comput. Mach. **9**, 350–371 (1962)

14. Hartmanis, J., Lewis II, P., Stearns, R.: Hierarchies of memory limited computations. In: IEEE Conference on Record on Switching Circuit Theory and Logical Design, pp. 179–190 (1965)
15. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston (2001)
16. Iwama, K.: ASPACE($o(\log\log n)$) is regular. SIAM J. Comput. **22**, 136–146 (1993)
17. Mereghetti, C.: Testing the descriptional power of small Turing machines on nonregular language acceptance. Int. J. Found. Comput. Sci. **19**, 827–843 (2008)
18. Minsky, M.: Computation: Finite and Infinite Machines. Prentice Hall, Englewood Cliffs (1967)
19. Szepietowski, A.: Turing Machines with Sublogarithmic Space. LNCS, vol. 843. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58355-6
20. Yakaryılmaz, A., Say, A.: Tight bounds for the space complexity of nonregular language recognition by real-time machines. Int. J. Found. Comput. Sci. **24**, 1243–1253 (2013)

A Framework for (De)composing with Boolean Automata Networks

Kévin Perrot, Pacôme Perrotin(✉), and Sylvain Sené

Aix-Marseille University, Toulon University, CNRS, LIS, Marseille, France
pacome.perrotin@lis-lab.fr

Abstract. Boolean automata networks (BANs) are a generalisation of Boolean cellular automata. In such, any theorem describing the way BANs compute information is a strong tool that can be applied to a wide range of models of computation. In this paper we explore a way of working with BANs which involves adding external inputs to the base model (via modules), and more importantly, a way to link networks together using the above mentioned inputs (via wirings). Our aim is to develop a powerful formalism for BAN (de)composition. We formulate two results: the first one shows that our modules/wirings definition is complete; the second one uses modules/wirings to prove simulation results amongst BANs.

Keywords: Boolean automata networks · Modules · Wirings Simulation

1 Introduction

Boolean automata networks (BANs) can be seen as a generalisation of cellular automata that enables the creation of systems composed of Boolean functions over any graph, while cellular automata only operate over lattices of any dimension. The study of the dynamics of a BAN, that describes the set of all computations possible in such a system, is a wide and complex subject. From very simple networks computing simple Boolean functions to possibly infinite networks able to simulate any Turing machine, the number of configurations always grows exponentially with the size of the network, making any exhaustive examination of its dynamics impractical. The study of such dynamics is nevertheless an important topic which can impact other fields. BANs are for example used in the study of the dynamics of gene regulatory networks [8,12,17] in biology.

Many efforts to characterise the dynamics of BANs have already been put forward. For example, some studies [1,14] examine the behaviour of networks composed of interconnected cycles. The modularity of BANs has been studied from multiple perspectives. In particular from a static point of view [2,13], and a functional one [4,7,16]. In this paper, we explore a compositional approach to BANs that allows to decompose a BAN into subnetworks called modules, and to compose modules together in order to form larger networks. We define a module as a BAN on which we add external inputs. These inputs are used to manipulate the result of the network computation by adding extra information. They can also be used to interconnect multiple modules, making more complex networks. Those constructions resemble the circuits described in Feder's thesis [9], and modules can be seen as a generalisation of circuits over any update mode.

Section 2 discusses the possible motivations for a (de)compositional study of BANs. Section 3 introduces BANs and update modes, and Sects. 4 and 5 develop a formalism for the modular study of BANs, justified by a first theorem showing that any network can be created with modules and wirings. We also present an application of our definitions to BAN simulation in Sect. 6, leading to a second theorem stating that composing with local simulations is sufficient to (globally) simulate a BAN. Finally, Sect. 7 presents and analyses two illustrations of the principles presented in Sect. 2.

The demonstrations of all results are available on the arxiv version of this paper (reference 1802.10400).

2 Motivations

BANs, despite being very simply defined locally, become complex to analyse as the representation of their dynamics grows exponentially in the size of their networks. BANs have been proven to be Turing-complete [5] and as most of Turing-complete systems are able to show complex and emergent properties.

Yet, an important number of networks can be partially understood when viewed through the lens of functionality (what an object is meant to achieve). Functionality enables to use abstraction to reduce the considered network (or some part of it) to the computation of a function or the simulation of a dynamical system. Assuming a functionality of the parts of a network can let us conclude on the functionality of the network itself, at the cost of letting aside an absolute characterisation of its dynamics (which is often practically impossible). Such a functional interpretation aims at offering the possibility to make verifiable predictions in a short amount of time.

It is not known if every Boolean automata network can be cut into a reasonable amount of parts to which one can easily affect a functionality. We will justify our present argument by illustrating it in Sect. 7.

3 Boolean Automata Networks

3.1 Preliminary Notations

Let us first describe some of the notations used throughout the paper. Let $f : A \to B$ be a mapping from set A to set B. For $S \subseteq A$ we denote $f(S) = \{b \in B \mid \exists a \in S, f(a) = b\}$. We denote $f\big|_S$ the restriction of f to the domain S, $f\big|_S \colon S \to B$ such that $f\big|_S(a) = f(a)$ for all $a \in S$. Let $\mathrm{dom}(f)$ be the domain of f, and $g \circ f$ the composition of f then g. For f and g two functions with disjoint domains of definition, we define $f \sqcup g$ as the function defined such that:

$$f \sqcup g(x) = \begin{cases} f(x) & \text{if } x \in \mathrm{dom}(f) \\ g(x) & \text{if } x \in \mathrm{dom}(h) \end{cases}.$$

We denote $\mathbb{B} = \{0, 1\}$ the set of Booleans. For K a sequence of m elements, the sub-sequence from the i-th element to the j-th element is denoted $K_{[i,j]}$. We sometimes define functions without naming them with the notation $a \mapsto b$, signifying that for any input a the function will return b. For example, the function $n \mapsto 2 \times n$ is a function that takes a number n and returns the value of n multiplied by 2.

3.2 Definitions

A BAN is based upon a set of automata. Each automaton is defined as a Boolean function, with arity the size of the network. Each variable of the function of each automaton is meant to correspond to an automaton in the network. By considering a configuration of Boolean values over this network, we can compute the Boolean function of each automaton and obtain a Boolean value for each automaton (*i.e.* a local state). These values can be used to update the global state of the network, that we call a configuration. If we decide to update the value of each automaton at once, the update mode is parallel. However, if only one automaton is updated at each time step, the update mode is sequential [10,15].

Definition 1. *A* configuration *on a set S is a function $x : S \to \mathbb{B}$.*

A BAN F defined over the set S associates a Boolean function to each element of S. Each of theses functions is defined from the set of all configurations of the BAN, $S \to \mathbb{B}$, to the Boolean set, $\mathbb{B}$.

Definition 2. *For S a set, a* Boolean automata network *(BAN) F is a function $F : S \to (S \to \mathbb{B}) \to \mathbb{B}$.*

For each $s \in S$, we denote $f_s = F(s)$ the local function of automaton s.

For $s \in S$ we denote $x_s = x(s)$. A function x is a configuration at a given time over the network. Thus, we can define our function f_s to be part of the set $(S \to \mathbb{B}) \to \mathbb{B}$. This way, a BAN F can be defined as a function from the set S to the set $(S \to \mathbb{B}) \to \mathbb{B}$. We find again that the set of all BANs over S can simply be defined as $S \to (S \to \mathbb{B}) \to \mathbb{B}$. For any BAN F and configuration x, we can define the configuration which is computed by F from x. A naive way to do so would be to define $x' = F(x)$ such that $x'_s = f_s(x)$ for every s; this definition however is very limiting: it only allows parallel updates of our system. In a general definition of BANs, a computation of a BAN should allow updates of only a subset of the functions of the network. Slight changes to the update mode of a BAN can deeply change its computational capabilities [3,11]. Most results that assume a parallel update mode cannot be applied to a sequential network; the reciprocal is also true. We set the following definition of an update over our BAN to be as general as possible.

Definition 3. *Any $\delta \subseteq S$ is an* update *over S.*

One can apply multiple consecutive updates to a BAN to effectively execute the BAN over an update mode. An *update mode* is simply a sequence of updates that is denoted Δ, where Δ_k is the k^{th} update of the sequence. We define the union operator between updates modes as it will be useful for the proof of our last theorem.

Definition 4. *Let Δ, Δ' be two update modes over a set S. The* union *of Δ and Δ' denoted $\Delta \cup \Delta'$ is the update mode defined as $(\Delta \cup \Delta')_k = \Delta_k \cup \Delta'_k$. The size of $\Delta \cup \Delta'$ is the maximum among the sizes of Δ and Δ'.*

We assume that $\Delta_k = \varnothing$ if k is greater than the size of Δ. Given an update δ, we can define the endomorphism F_δ over the set of all configurations. For every configuration x, we set $F_\delta(x)(s) = f_s(x)$ if $s \in \delta$, and $F_\delta(x)(s) = x(s)$ if $s \notin \delta$. In other words, the value of s in the new configuration is set to $f_s(x)$ only if $s \in \delta$, otherwise the Boolean affectation of s remains x_s. Now, we can define the execution of F in a recursive way.

Definition 5. *The* execution *of F over x, under the update mode Δ, is the function $F_\Delta : (S \to \mathbb{B}) \to (S \to \mathbb{B})$ defined as $F_{\Delta[1,k]}(x) = F_{\Delta_k}(F_{\Delta[1,k-1]}(x))$, with $F_{\Delta[1,1]}(x) = F_{\Delta_1}(x)$.*

Throughout this paper we represent BANs as graphs called interaction graphs. Interaction graphs are a classical tool in the study of BANs. For a BAN F defined over S, the interaction graph of F is the oriented graph $G = (S, \epsilon)$, where $(s, s') \in \epsilon$ if and only if the variable x_s influences the computation of the function $F(s')$.

4 Modules

Modules are BANs with external inputs. Such inputs can be added to any local function of a module, and any local function of a module can have multiple inputs. When a local function has n inputs, the arity of this function is increased by n. These new parameters are referred to by elements in a new set E: the elements of E describe the inputs of the module; those of S describe the internal elements of the module. To declare which input $e \in E$ is affected to each function f_s, we use function α.

Definition 6. *Let S and E be two disjoint sets. An* input declaration *over S and E is a function $\alpha : S \to \mathcal{P}(E)$ such that $\{\alpha(s) \mid s \in S\}$ is a partition of E.*

For each s, $\alpha(s)$ is the set of all external inputs of function f_s. The partition property is important because without it, some input could be assigned to multiple nodes, or to no node at all, which is contrary to our vision of input. To simplify notations, we sometimes denote $E_s = \alpha(s)$. Now, let us explicit the concept of a module.

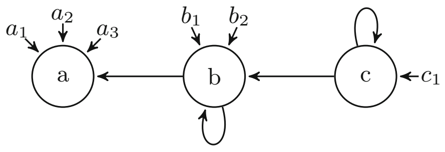


Fig. 1. Interaction graph of the module detailed in Example 1.

Definition 7. *A* module M *over* (S, E, α) *is defined such that, for each* $s \in S$, $M(s)$ *is a function* $M(s) : (S \cup E_s) \to \mathbb{B}$.

If M is a module defined over $(S, \varnothing, s \mapsto \varnothing)$, M is also a BAN. To compute anything over this new system, we need a configuration $x : S \to \mathbb{B}$ and a configuration over the elements of E.

Definition 8. *An* input configuration *over* E *is a function* $i : E \to \mathbb{B}$.

Let x be a configuration over S, and i an input configuration over E. As x and i are defined over disjoint sets, we define $x \sqcup i$ as their union. Such an union, coupled with an update over S, is enough information to perform a computation over this new model.

Definition 9. *Let* x *be a configuration over* S *and* i *an input over* E. *Let* δ *be an update over* S. *The* computation *of* M *over* x, i *and* δ, *denoted* $M_\delta(x \sqcup i)$, *is the configuration over* S *such that* $M_\delta(x \sqcup i)(s) = f_s(x \sqcup i\big|_{E_s})$ *for each* $s \in \delta$, *and* $M_\delta(x \sqcup i)(s) = x(s)$ *for every* $s \in S \setminus \delta$.

In the following example, we assume a total order over $S \cup E$, allowing us to intuitively write configurations as binary words. For example, $x = 101$ means $x(a) = 1$, $x(b) = 0$ and $x(c) = 1$.

Example 1. $S = \{a, b, c\}$, and $E = \{a_1, a_2, a_3, b_1, b_2, c_1\}$. We define α such that $\alpha(a) = \{a_1, a_2, a_3\}$, $\alpha(b) = \{b_1, b_2\}$ and $\alpha(c) = \{c_1\}$. Let M be a module over (S, E, α), such that $M(a) = x_b \vee a_1 \vee a_2 \vee a_3$, $M(b) = \neg x_b \vee x_c \vee \neg b_1 \wedge b_2$, and $M(c) = \neg c_1$. Let $x = 101$, $i = 000010$ and $\delta = \{a, b\}$. We get that $M_\delta(x \sqcup i) = M_{\{a,b\}}(101 \sqcup 000010)$ is such that $M_\delta(x \sqcup i)(a) = f_a(x \sqcup i\big|_{E_a}) = 0$, $M_\delta(x \sqcup i)(b) = f_b(x \sqcup i\big|_{E_b}) = 1$, and $M_\delta(x \sqcup i)(c) = x(c) = 1$. Therefore $M_\delta(x \sqcup i) = 011$. A representation of this module is pictured in Fig. 1.

Let us now define executions, while considering that the input configuration can change over time.

Definition 10. *Let* $t > 1$. *Let* $I = (i_1, i_2, \ldots, i_{t-1})$ *be a sequence of input configurations over* E, $X = (x_1, x_2, \ldots, x_t)$ *a sequence of configurations over* S, *and* Δ *an update mode over* S *of size* t. (X, I, Δ) *is an* execution *of* M *if for all* $1 \leq k < t$, $x_{k+1} = M_{\Delta_k}(x_k \cup i_k)$.

This definition allows for variation over the inputs over time. As this particular feature is not needed throughout this paper, we also propose a simpler definition of executions over modules which only allows fixed input values over time.

Definition 11. *Let i be an input configuration over E. The* execution *of M over $x \cup i$ with update mode Δ is an endomorphism over the set of all configurations, denoted M_Δ. It is defined as $M_{\Delta[1,k]}(x \sqcup i) = M_{\Delta_k}(M_{\Delta[1,k-1]}(x \sqcup i) \sqcup i)$, with $M_{\Delta[1,1]}(x \sqcup i) = M_{\Delta_1}(x \sqcup i)$.*

5 Wirings

The external inputs of a module can be used to encode any information. For instance, we could encode any periodic (or non-periodic) sequence of Boolean words into the inputs of a given module. We could also encode the output of a given BAN or module, combining in some way the computational power of both networks. Such a composition of modules is captured by our definition of wirings. A wiring is an operation that links together different inputs and automata from one more or modules, thus forming bigger and more complex modules.

We decompose this compositional process into two different families of operators: the non-recursive and the recursive wirings. The first ones connect the automata of one module to the inputs of another; the second ones connect the automata of a module to its own inputs. A wiring, recursive or not, is defined by a partial map ω linking some inputs to automata. Let us first define non-recursive wirings.

Definition 12. *Let M, M' be modules defined over (S, E, α) and (S', E', α') respectively, such that S, S' and E, E' are two by two disjoint. A* non-recursive wiring *from M to M' is a partial map ω from E' to S.*

The new module result of the non-recursive wiring ω is denoted $M \rightarrowtail_\omega M'$ and is defined over $(S \cup S', E \cup E' \setminus \mathrm{dom}(\omega), \alpha_\omega)$. The input declaration of $M \rightarrowtail_\omega M'$ is $\alpha_\omega(s) = \alpha(s) \setminus \mathrm{dom}(\omega)$ (in particular, $\alpha_\omega(s) = \alpha(s)$ if $s \in S$). Given $s \in S \cup S'$, the local function $M \rightarrowtail_\omega M'(s)$, denoted f_s^ω, is defined as

$$f_s^\omega(x \sqcup i) = \begin{cases} f_s(x|_S \sqcup i|_{E_s}) & \text{if } s \in S \\ f'_s(x|_{S'} \sqcup i|_{E'_s \setminus \mathrm{dom}(\omega)} \sqcup (x \circ \omega|_{E'_s})) & \text{if } s \in S' \end{cases}.$$

In this new module, some inputs of M' have been assigned to the values of some elements of M. Such assignments are defined in the wiring ω. For any $s \in S \cup S'$, the function $M \rightarrowtail_\omega M'(s)$ (denoted f_s^ω) is defined over $(S \cup S' \cup \alpha_\omega(s)) \to \mathbb{B}$. In the case $s \in S'$, the image of $x \sqcup i$ is given by f'_s which expects a configuration on $S' \cup E'_s$: the configuration on S' is provided by x, and the configuration on E' is partly provided by i (on $E'_s \setminus \mathrm{dom}(\omega)$), and partly provided by $(x \circ \omega)$ (on $\mathrm{dom}(\omega) \cap E'_s$).

Definition 13. *Let M be a module over (S, E). A* recursive wiring *of M is a partial map ω from E to S.*

With ω defining now a recursive wiring over a module M, the result is similar if not simpler than in the definition of non-recursive wirings. The new module obtained from a recursive wiring ω on M is denoted $\circlearrowright_\omega M$ and is defined over $(S, E \setminus \mathrm{dom}(\omega), \alpha_\omega)$ with the input declaration defined as, for any $s \in S$, $\alpha_\omega(s) = \alpha(s) \setminus \mathrm{dom}(\omega)$. Given $s \in S$, x and i, the local function $\circlearrowright_\omega M(s)$ is denoted f_s^ω and is evaluated to $f_s^\omega(x \sqcup i) = f_s(x \sqcup i\big|_{E_s \setminus \mathrm{dom}(\omega)} \sqcup (x \circ \omega\big|_{E_s}))$.

Recursive and non-recursive wirings can be seen as unary and binary operators respectively, over the set of all modules. For any ω, we can define the operators $\rightarrowtail_\omega$ and $\circlearrowright_\omega$. For simplicity we define that $M \rightarrowtail_\omega M' = \varnothing$ and $\circlearrowright_\omega M = \varnothing$ if the wiring ω is not defined over the same sets as M or M'. Notice that both the recursive and non-recursive wirings defined by $\omega = \varnothing$ are well defined wiring. They define two operators, $\circlearrowright_\varnothing$ and $\rightarrowtail_\varnothing$, that will be useful later on.

Property 1. The following statements hold.

(i) $\forall M, \quad \circlearrowright_\varnothing M = M.$
(ii) $\forall M, M', \quad M \rightarrowtail_\varnothing M' = M' \rightarrowtail_\varnothing M.$
(iii) $\forall M, M', M'', \quad M \rightarrowtail_\varnothing (M' \rightarrowtail_\varnothing M'') = (M \rightarrowtail_\varnothing M') \rightarrowtail_\varnothing M''.$

For simplicity of notations, we will denote the empty non-recursive wiring as the union operator over modules: $M \cup M' = M \rightarrowtail_\varnothing M'$.

It is quite natural to want to put two modules together, by linking the input of the first to states of the second, and conversely. Our formalism allows this operation in two steps: first, use a non-recursive wiring to connect all of the desired inputs of the first module to states of the second module. Then, use a recursive wiring to connect back all of the desired inputs of the second module to states of the first module.

We now express that recursive and non-recursive wirings are expressive enough to construct any BAN or module, in Theorem 1. Our aim is to show that for any division of a module into smaller parts (partitioning), there is a way to get back to the initial module using only recursive and non-recursive wirings.

Definition 14. *Let (S, E, α). Let P be a set such that $\{S_p \mid p \in P\}$ is a partition of S. We define the* corresponding partition *of E as $\{E_p = \bigcup_{s \in S_p} \alpha(s) \mid p \in P\}$.*

Definition 15. *We can now develop the corresponding partition of the input declaration, and define the partition of M itself. For every $p \in P$, we define $\alpha_p = \alpha\big|_{S_p}$ over S_p and E_p.*

Definition 16. *For every $p \in P$, let Q_p verify $Q_p \cap S = \varnothing$ and $|Q_p| = |S|$, and let $\tau_p : S \to Q_p$ be a bijection. For any $p \in P$, the* sub-module *M_p over $(S_p, E_p \cup \tau_p(S \setminus S_p), \alpha_p)$ is defined for $s \in S_p$ as, for all $x : S \to \mathbb{B}$ and for all $i : E \to \mathbb{B}$,*

$$M_p(s)(x\big|_{S_p} \sqcup i_p) = M(s)(x \sqcup i),$$

where $i_p(e) = i(e)$ if $e \in E_p$ and $i_p(e) = x(\tau_p^{-1}(e))$ if $e \in \tau_p(S \setminus S_p)$.

In the previous definition, the purpose of each Q_p is to work as a representation of the set S for every sub-module M_p. Without it, every module M_p would have used the set $(S \setminus S_p) \cup E_p$ as input set. However our definition of wiring requires the input sets of the wired modules to be disjoint from each other. The sets Q_p are a workaround to bypass this technical point.

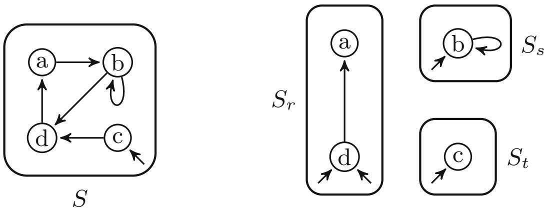


Fig. 2. Interaction graphs related to Example 2. The interaction graph of the original module is on the left and the interaction graphs of the partition of M are on the right. Notice that we did not represent the input sets E, Q_r, Q_s and Q_t.

Example 2. Let $S = \{a, b, c, d\}$, $E = \{e\}$, $P = \{r, s, t\}$ and $S_r = \{a, d\}$, $S_s = \{b\}$ and $S_t = \{c\}$. For each $p \in P$, we define $Q_p = \{a_p, b_p, c_p, d_p\}$. In the module M_r, $\alpha_r(a) = \varnothing$ and $\alpha_r(d) = \{b_r, c_r\}$. In the module M_s, $\alpha_s(b) = \{a_s\}$. In the module M_t, $\alpha_t(c) = \{e\}$. The modules M_r, M_s and M_t are defined over disjoint sets and can be wired (see Fig. 2 for an illustration).

As a reminder, the union operator over modules is defined to be the result of an empty non-recursive wiring.

Theorem 1. *Let M be a module and $\{M_p \mid p \in P\}$ any partition of that module, then there exists a recursive wiring ω such that $M = \circlearrowleft_\omega \left(\bigcup_{p\in P} M_p\right)$.*

Sketch of Proof. We construct ω to wire every link lost in partition P.

Theorem 1 allows to say that our definition of wiring is complete: any BAN or module can be assembled with wirings. It can be reworked more algebraically. Let $\mathcal{M}$ denote the set of all modules (which includes $\varnothing$), and for any $n \in \mathbb{N}$, let $\mathcal{M}_n$ denote the set of all modules of size n (we have $\mathcal{M} = \bigcup_{n\in\mathbb{N}} \mathcal{M}_n$). For any subset $A \subseteq \mathcal{M}$ we denote $\overline{A}^\omega$ the closure of A by the set of wiring operators $\bigcup_\omega \{\rightarrowtail_\omega, \circlearrowleft_\omega\}$. The following result is a direct corollary of Theorem 1.

Corollary 1. *The set of all modules is equal to the closure by any wiring of the set of modules of size* 1,

$$\mathcal{M} = \overline{\mathcal{M}_1}^\omega.$$

Every module in $\mathcal{M}_1$ is of size 1, but as the set of inputs E of a module is not bounded, the set $\mathcal{M}_1$ is infinite. In our opinion, this corollary is enough to demonstrate that our definition of modules and wirings is sound.

6 Simulation

BANs are by nature complex systems and sometimes, we like to understand the computational power of a subset of them by demonstrating that they are able to simulate (or be simulated by) another subset of BANs. By *simulation*, we generally mean that a BAN is able to reproduce, according to some encoding, all the possible computations of another BAN.

Simulation is a powerful way to understand the limitations and possibilities of BANs. It is still difficult to prove if any two BANs simulate each other. In the present paper our aim is to prove that the property of simulating any BAN can be reduced in some cases to the property of locally simulating any Boolean function. Locally simulating a function means that a module reproduces any computation of that function, when the parameters of the function are encoded in the module inputs. Our claim is that if we can locally simulate every function of a BAN, in a way such that the simulating modules are able to communicate with each other, then we can simulate the same BAN with a bigger module which is obtained by a wiring over the locally simulating modules. In this context, modules become a strong tool to reduce the complexity of simulation (which is a global phenomena) to a local scale, which is more tractable.

Let us go into further details. For F a BAN over the set S, our aim is to simulate F. For this purpose, for each $a \in S$, we create M_a, a module which is defined over some sets (T_a, E_a, α_a) and locally simulates the function f_a. To assert this local simulation we need to define a Boolean encoding ϕ_a over the configurations of M_a. We also need to define how these modules communicate with each other, and in the end how they will be wired together. For any couple $a, b \in S$ such that $a \neq b$, we define the set $U_{a,b}$ as a subset of T_a. This set represents all the automata of M_a that are planned to be connected to inputs of M_b. We can say that the elements of $U_{a,b}$ are the only way for the module M_a to send information to the module M_b. We define which information is sent from M_a to M_b at any time with a Boolean encoding $\phi_{a,b}$ over the set of configurations on $U_{a,b}$. By definition we always have that if $U_{a,b} \neq \varnothing$, then $\phi_a(x|_{T_a}) \neq \bullet \Rightarrow \phi_{a,b}(x|_{U_{a,b}}) = \phi_a(x|_{T_a})$. This means that if a module encodes an information ($\bullet$ being the absence of information, *i.e.* in this case $\phi_a(x|_{T_a})$ equals 0 or 1), the same information is sent from that module to each module that is meant to receive information from it. In other words, all encodings are coherent.

Now that our modules are set to communicate with each other, we only need to wire them to each other. The precise nature of this wiring is defined, for every pair $a, b \in S$ such that $a \neq b$, by the function $I_{a,b} : E_b \to U_{a,b}$ which we call interface between a and b. By definition:

- for every $s \in U_{a,b}$, there exists $e \in E_b$ such that $I_{a,b}(e) = s$ (surjectivity);
- for every $b \in S$, $\bigsqcup_a I_{a,b}$ is a total map from E_b to $\bigcup_a U_{a,b}$.

With such an interface defined for every pair (a, b), the final wiring connecting all modules together is decomposed in two steps. The first one empty-wires every

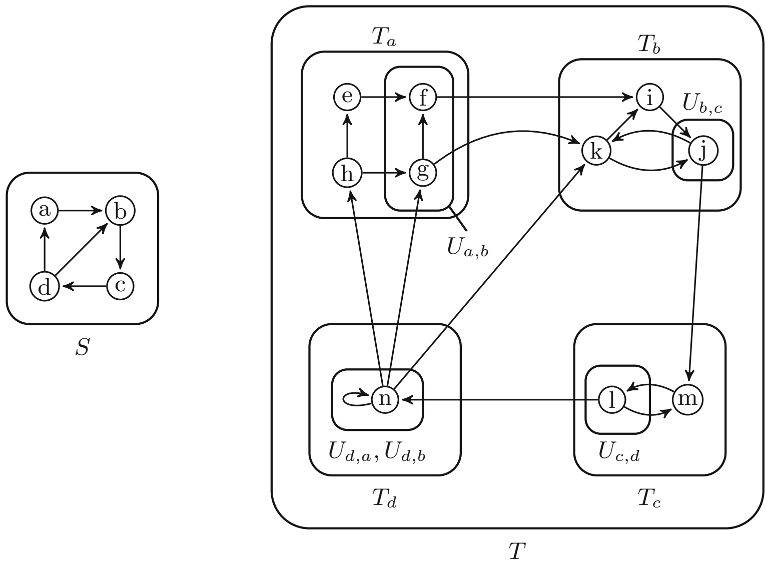


Fig. 3. Interaction graphs of the modules detailed in Example 3. The interaction graph of the original BAN is on the left and the interaction graph of the simulating BAN is on the right. The simulating BAN is decomposed into four sub-modules, one for each node in S. Notice that we did not represent the input sets E_a, E_b, E_c and E_d. The connections between the sets T_a, T_b, T_c and T_d are based upon the interfaces defined in the example.

module together, the second one applies a recursive wiring which is defined as the union of every interface $I_{a,b}$. The last condition that we have stated over the definition of an interface lets us know that the obtained module has no remaining inputs; it can be considered as a BAN, defined over $T = \bigcup_{a \in S} T_a$. All these sets are illustrated in Fig. 3.

Example 3. Let $S = \{a, b, c, d\}$. Let $T_a = \{e, f, g, h\}$, $T_b = \{i, j, k\}$, $T_c = \{l, m\}$ and $T_d = \{n\}$. Let $T = T_a \cup T_b \cup T_c \cup T_d$. Let $E_a = \{e_g, e_h\}$, $E_b = \{e_i, e_k, e'_k\}$, $E_c = \{e_m\}$ and $E_d = \{e_n\}$. Let $U_{a,b} = \{f, g\}$, $U_{b,c} = \{j\}$, $U_{c,d} = \{l\}$, $U_{d,a} = U_{d,b} = \{n\}$, and any other U set empty. We will define interfaces as the following: $I_{a,b}(e_i) = f$, $I_{a,b}(e_k) = g$, $I_{b,c}(e_m) = j$, $I_{c,d}(e_n) = l$, $I_{d,a}(e_h) = n$, $I_{d,a}(e_g) = n$ and $I_{d,b}(e'_k) = n$ (see Fig. 3).

Definition 17. *Let A be a set. A* Boolean encoding *over A is a function $\phi : (A \to \mathbb{B}) \to (\{0, 1, \bullet\})$, such that there exists at least one x such that $\phi(x) = 0$ and one x such that $\phi(x) = 1$.*

For $x : A \to \mathbb{B}$ (a Boolean configuration over a set A), $\phi(x) = 1$ means that x encodes a 1, $\phi(x) = 0$ means that x encodes a 0, and $\phi(x) = \bullet$ means that x does not encode any value. Each ϕ_a is defined as an encoding over T_a, and each $\phi_{a,b}$ as an encoding over $U_{a,b}$.

By definition we enforce that

$$\text{if } U_{a,b} \neq \varnothing, \text{ then } \phi_a(x\big|_{T_a}) \neq \bullet \Rightarrow \phi_{a,b}(x\big|_{U_{a,b}}) = \phi_a(x\big|_{T_a}).$$

Given a BAN on S and some $a \in S$, let us now define the local simulation of function f_a by a module M_a. We want to express that given any configuration $x : S \to \mathbb{B}$, all the configurations $x' : T_a \to \mathbb{B}$ and input configurations $i' : E_a \to \mathbb{B}$ such that x', i' encode the same information as x, the result of the dynamics on x', i' in the simulating module must encode the result of the dynamics on x in the simulated automaton. To express that x' encodes the state of a in x is easy: $\phi_a(x') = x_a$. To express that i' encodes the state of all $b \neq a$ in x requires an additional notation. On the one hand we have $\phi_{b,a} : (U_{b,a} \to \mathbb{B}) \to (\{0, 1, \bullet\})$, and on the other hand we have $i' : E_a \to \mathbb{B}$ describing the input-configuration of module M_a, and $I_{b,a} : E_a \to U_{b,a}$ describing the interface from b to a. To plug these objects together, we put forward the hypothesis that if $I_{b,a}(e) = I_{b,a}(e')$, then $i'(e) = i'(e')$ for any $e, e' \in E_a$. This hypothesis is justified by the fact that the wiring applied by $I_{b,a}$ enforces the value of two inputs connected to the same element to be the same. Now, we define $i' \circ I_{b,a}^{-1}$ the configuration over $U_{b,a}$ such that $i' \circ I_{b,a}^{-1}(s) = i'(e)$ for any e such that $I_{b,a}(e) = s$. By our hypothesis this configuration is well defined.

Definition 18. *Let $a \in S$, f_a be a Boolean function over S and M_a a module over (T_a, E_a, α_a), with ϕ_a (resp. $\phi_{b,a}$) a Boolean encoding over T_a (resp. $U_{b,a}$). Given a finite update mode Δ over T_a, M_a* locally simulates *f_a, denoted by $M_a \prec_\Delta f_a$, if for all $x : S \to \mathbb{B}$,*

1. *and for all $x' : T_a \to \mathbb{B}$ such that $\phi_a(x') = x_a$,*
2. *and for all $i' : E_a \to \mathbb{B}$ such that for all $b \neq a$ we have $\phi_{b,a}(i' \circ I_{b,a}^{-1}) = x_b$,*
3. *we have:*

$$\phi_a({M_a}_\Delta(x' \sqcup i')) = f_a(x).$$

This local simulation can be defined on a wide range of update modes Δ. To ensure that the simulation works as planned at the global scale, we restrict the range of update modes Δ used for the local simulations, to those where no automata with input(s) are updated later than the first update.

Definition 19. *An update mode Δ over a module M is defined to be* input-first *if for all $k > 1$ and all $s \in \Delta_k$, we have $\alpha(s) = \varnothing$.*

Definition 20. *We define that M is able to* input-first simulate *f if there exists an input-first Δ such that $M \prec_\Delta f$.*

Intuitively, such update modes let us make parallel the computation of modules; all information between modules is communicated simultaneously at the first frame of computation (update), followed by isolated updates in each module. To define global simulation, we introduce the global encoding $\Phi : (S \to \mathbb{B}) \to (S' \to \mathbb{B}) \cup \{\bullet\}$ which always verifies that for all $x' : S' \to \mathbb{B}$, there exists $x : S \to \mathbb{B}$ such that $\Phi(x) = x'$.

Definition 21. *Let F and F' be two Boolean automata networks over S and S' respectively. We define that F* simulates *F', denoted by $F \prec F'$, if there exists a global encoding Φ such that for all x', x such that $\Phi(x) = x'$, and for all $\delta' \subseteq S'$, there exists a finite update mode Δ over S such that $\Phi(F_\Delta(x)) = F'_{\delta'}(x')$.*

Given the definitions of local and global simulation, for any BAN F over a set S, we define each module M_a as earlier, each defined over (T_a, E_a, α_a), along side each set $U_{a,b}, I_{a,b}$ and each encoding $\phi_a, \phi_{a,b}$.

Theorem 2. *Let F be a BAN over S. For each $a \in S$, let M_a be a module over (T_a, E_a, α_a) that locally simulates $F(a)$ in an input-first way. There exists a recursive wiring ω over $T = \bigcup_{a \in S} T_a$ such that*

$$\circlearrowleft_\omega \left(\bigcup_{a \in S} M_a \right) \prec F.$$

Sketch of Proof. We prove that the execution of the module M obtained from the wiring ω can be built from the execution of each M_a. We apply the hypothesis of local simulation on each M_a, and obtain a global simulation.

This theorem helps us investigate if every BAN can be simulated by a BAN with a given property, hence justifying that theoretical studies can impose some restrictions without loss of generality. If every function f can be locally simulated by a given module with a property $\mathcal{P}$, and if property $\mathcal{P}$ is preserved over wirings, then we know that any BAN can be simulated by another BAN with the property $\mathcal{P}$. This is formally proven for the following cases.

Corollary 2. *Let F be a BAN. There exists F' such that $F' \prec F$ and every function of F' is a disjunctive clause.*

Corollary 3. *Let F be a BAN. There exists F' such that $F' \prec F$ and every function of F' is monotone.*

Sketch of Proof. Both of theses results are obtained by replacing the automata of F by modules that locally simulates them. For disjunctivity, the module has one automaton for each clause of the conjunctive normal form of the simulated function, and one for the result (using De Morgan's law we convert the outer conjunction to a disjunction). For monotony, we use a lemma that shows that we can always construct a monotone function from any function at the cost of duplicating each variable. Using this lemma we construct a network with twice the automata which locally simulates any function. The results are obtained by the Theorem 2.

It can seem strange that this particular theorem applies to BANs and not to modules (as it would be a more general result). Such a result would need a definition of simulation between modules, and such a definition would imply an interpretation of the information provided by the simulating module's inputs. We choose not to develop this particular idea, as this theorem was only meant to apply to BANs, but a generalisation of this result to modules would be a good subject for future works.

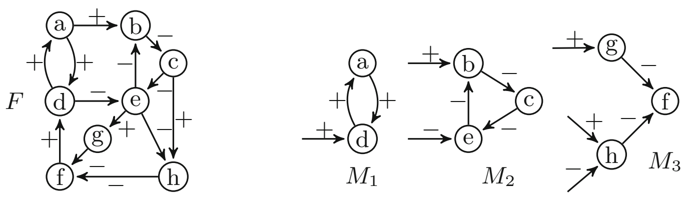


Fig. 4. Representation of a handmade Boolean automata network F next to the three different modules M_1, M_2 and M_3 that compose it. The function of each automaton is defined as a disjunctive clause with a positive literal for each incident "+" edge, and a negative literal for each incident "−" edge. For example, $f_h(x) = x_c \vee \neg x_e$.

7 Examples

To illustrate and justify the notions that are presented in Sect. 2, we shall now present two examples of BANs that can be partially understood by cutting them into modules. The first example is a toy BAN illustrated in Fig. 4. In this representation we assume the function of each automaton to be a disjunctive clause with one literal for each incident edge, the sign of which dictates the sign of the literal.

Looking at this example, it does not seem easy to express the entire behaviour of the BAN F. Its representation is a strongly connected graph with multiple interconnected positive and negative cycles. Yet, cutting this graph into multiple modules and analysing the functionality of each of them is an easy way to understand interesting parts of the dynamics of the network.

By assuming the decomposition of F as shown in Fig. 4, we can start to attach a functionality to each module. Module M_1 is a positive cycle, where the configuration $x_a = x_d = 1$ is a fixed point (whatever the input). Its functionality can be identified as a "one time button" that cannot be pushed back. Module M_2 is a negative cycle, which are known for their long limit cycles. The difference here is that as M_2 has two inputs, its behaviour can be stabilised into a fixed point by a fixed input. For example, the fixed point $x_b = x_e = 1, x_c = 0$ can be obtained with the constant input $i_b = 1, i_e = 0$. Finally, the module M_3 is acyclic and thus only computes the Boolean function $\neg i_g \vee (\neg i_h \wedge i_{h'})$. It follows that M_3 stabilises to a fixed point under any constant input.

This simple analysis leads us to the following conclusion: every fair execution (meaning executing every automaton an infinite amount of time) of F which verifies $x_a = x_d = 1$ at any moment stabilises into a fixed point. This is true because $x_a = x_d = 1$ implies that the "one time button" of M_1 is pushed in, which locks the behaviour of M_2 into a fixed point, which leads M_3 to compute a Boolean function over a fixed input. This somewhat informal demonstration has led us to a conclusion that was not easily implied by the architecture of the network, showcasing the usefulness of understanding networks as composition of parts to which one can assign functionalities.

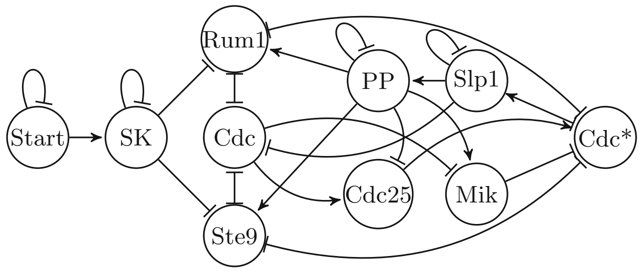


Fig. 5. Representation of the network simulating the cell cycle sequence of fission yeast extracted from [6]. Activating interactions are represented by simple arrows and inhibiting interactions by flat arrows. The detail of each node's function is available in the original paper.

The second example is drawn from a model predicting the cell cycle sequence of fission yeast [6]. This network is represented in Fig. 5, and can be decomposed into a more abstract network, where each node represents a module of the original network. This network is represented in Fig. 6 and its modules are constructed as follows: $C = \{Rum1, Ste9\}, D = \{Cdc, Cdc*\}, F = \{Cdc25\}, G = \{Mik\}, I = \{Start, SK\}, J = \{PP, Slp1\}$. A quick analysis of these modules leads us to sort them into three categories: cycles (C, D), functions (F, G) and igniters (I, J). Let us now explain this organisation in an informal way.

The two cycle modules C and D are organised in a 4-cycle of negative feedback which means that if considered separately from the rest of the network, those two modules would behave as antagonists: in most cases, when the automata of C (resp. D) are evaluated to 1, the automata of D (resp. C) will be evaluated to 0. Modules F and G can be viewed as functions which help D and C respectively to be evaluated to 1; they both are influenced by J in different ways. Modules I and J are called igniters because they turn themselves to 0 every time they are evaluated to 1, but not before influencing the other nodes. Module I inhibits C when activated, and can be considered as the input of the whole network. Module J is activated by D, activates C and G, and inhibits F.

From this we can conclude that if the network stabilises, it will more likely stabilise by evaluating C to 1 and D to 0. This conclusion arises from the fact that D activates J, which in turn inhibits D directly, but also inhibits F (which activates D) and activates G (which inhibits D). This also means that F will be evaluated to 0 and G to 1. Finally, I and J will naturally be evaluated to 0 because of the natural negative feedback that compose them. This particular evaluation of the network (only C and G to 1) is actually the main fixed point of the network's dynamics put forward in [6] and is named $G1$. This shows that such a fixed point can be described without the need to compute the $2^{10} = 1024$ different configurations of the network and their dynamics.

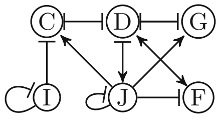


Fig. 6. Abstract representation of the interactions between the modules C, D, F, G, I and J based upon the network represented in Fig. 5.

8 Conclusion

The two theorems formulated in this article tell us that seeing BANs as modular entities is a way to discover useful results. With the simple addition of inputs to BANs, we have expressed a general simulation structure that can be used to understand the computational nature and limits of given properties over BANs. Let us underline that all the definitions and results can be applied to BANs and modules defined over countably infinite sets of automata and inputs.

Wherever Turing-completeness is observed, complex behaviours emerge that cannot be simply or quickly formulated from the basic rules of the computation. In such situations, the solution is either to compute every single possibility to capture the whole dynamics of the observed system, or to simplify the model. We believe that the framework developed in this paper is a strong candidate to enable us to decompose complex networks into parts with tractable functionalities, and to make conclusions about the whole network at a cheaper cost. This approach is still very informal at this moment and will be the focus of further developments.

Acknowledgements. This work has been supported "Investissement d'avenir" program ANR-16-CONV-00001 and PACA Project Fri 2015_01134.

References

1. Alcolei, A., Perrot, K., Sené, S.: On the flora of asynchronous locally non-monotonic Boolean automata networks. In: Proceedings of SASB 2015, ENTCS, vol. 326, pp. 3–25 (2016)
2. Alon, U.: Biological networks: the tinkerer as an engineer. Science **301**, 1866–1867 (2003)
3. Aracena, J., Gómez, L., Salinas, L.: Limit cycles and update digraphs in Boolean networks. Discrete Appl. Math. **161**, 1–12 (2013)
4. Bernot, G., Tahi, F.: Behaviour preservation of a biological regulatory network when embedded into a larger network. Fund. Inform. **91**, 463–485 (2009)
5. Cook, M.: Universality in elementary cellular automata. Complex Syst. **15**, 1–40 (2004)
6. Bornholdt, S., Davidich, M.I.: Boolean network model predicts cell cycle sequence of fission yeast. PLoS One **3**, e1672 (2008)
7. Delaplace, F., Klaudel, H., Melliti, T., Sené, S.: Analysis of modular organisation of interaction networks based on asymptotic dynamics. In: Gilbert, D., Heiner, M. (eds.) CMSB 2012. LNCS, pp. 148–165. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33636-2_10

8. Demongeot, J., Goles, E., Morvan, M., Noual, M., Sené, S.: Attraction basins as gauges of robustness against boundary conditions in biological complex systems. PLoS One **5**, e11793 (2010)
9. Feder, T.: Stable networks and product graphs. Ph.D thesis, Stanford University (1990)
10. Fogelman, F., Goles, E., Weisbuch, G.: Transient length in sequential iteration of threshold functions. Discrete Appl. Math. **6**, 95–98 (1983)
11. Goles, E., Salinas, L.: Comparison between parallel and serial dynamics of Boolean networks. Theor. Comput. Sci. **396**, 247–253 (2008)
12. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. J. Theor. Biol. **22**, 437–467 (1969)
13. Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, U.: Network motifs: simple building blocks of complex networks. Science **298**, 824–827 (2002)
14. Noual, M.: Updating Automata Networks. Ph.D thesis, École Normale Supérieure de Lyon (2012)
15. Robert, F.: Discrete Iterations: A Metric Study. Springer, Heidelberg (1986). https://doi.org/10.1007/978-3-642-61607-5
16. Siebert, H.: Dynamical and structural modularity of discrete regulatory networks. In: Proceedings of COMPMOD 2009, EPTCS, vol. 6, pp. 109–124 (2009)
17. Thomas, R.: Boolean formalization of genetic control circuits. J. Theor. Biol. **42**, 563–585 (1973)

An Extension of Interval-Valued Computing Equivalent to Red-Green Turing Machines

Benedek Nagy[1](✉) and Sándor Vályi[2]

[1] Department of Mathematics, Eastern Mediterranean University, via Mersin-10, Famagusta, North Cyprus, Turkey
nbenedek.inf@gmail.com

[2] Institute of Mathematics and Informatics, University of Nyíregyháza, Nyíregyháza, Hungary
valyi.sandor@nye.hu

Abstract. Interval-valued computing is a kind of massively parallel computing. It operates on specific subsets of the interval [0,1) – unions of subintervals. They serve as basic data units and are called interval-values. It was established that this system (in its unrestricted version) has computing power equivalent to Turing machines, by a rather simple observation. However, this equivalence involves an infinite number of interval-valued variables. In this paper, a more refined equivalence is established using only a fixed number of interval-valued variables. This fixed number depends only on the number of states of the Turing machine – logarithmically. This method makes it also possible to extend interval-valued computations into infinite length to capture the computing power of red-green Turing machines.

Keywords: Unconventional computing
Massively parallel computing · Interval-valued computing
Red-green Turing machines · Simulation · Hypercomputation

1 Introduction

Interval-valued computing is a kind of massively parallel computing. It operates on specific subsets of interval $[0, 1)$ – unions of subintervals. They serve as basic data units, these are called interval-values. The operators are – aside from pointwise Boolean ones and shifts to left and right – a kind of zooming, so called product [6]. This operator is a 1-dimensional analogue of the zooming operator in optical computing on pictures [12]. It is natural to involve the other operators if the interval-values are considered as bit sequences indexed by $[0, 1)$. The massive parallelism of this computing concept lies in the fact that these operators are executed in one unit, all at once. It is similar in this aspect to bit vector machine computation [11].

J. Durand-Lose and S. Verlan (Eds.): MCU 2018, LNCS 10881, pp. 137–152, 2018.
https://doi.org/10.1007/978-3-319-92402-1_8

It was established in [6] that this system (in its unrestricted version) has computing power equivalent to Turing machines, by a rather simple observation. In this unrestricted version, the interval-valued computation sequence is generated by an arbitrary Turing machine, so it is easy to simulate any Turing machine by an unrestricted interval-valued computation and vice versa. However, this equivalence involves an infinite number of interval-valued storage places. If we restrict the interval-valued computation, then the arising computation class can have a restricted computing power.

If the first argument of a product operator applications is restricted to $\left[0, \frac{1}{2}\right)$ and the length of computation sequence is restricted polynomially, then we get a characterization of $PSPACE$ [6]. A stronger restriction to characterize NP is found in [9,10].

In this paper, a more refined Turing equivalence is established. For any given Turing machine, we give a representation of its configurations by some fixed number of interval-values. It is also shown that its configuration transition function can be implemented by a fixed length interval-valued computation sequence. This sequence can be looped then indefinitely until acceptance. By this way, we give an interval-valued computation sequence which is using a fixed number of interval-valued variables (storage places) and simulates the given Turing machine.

This fixed number of used interval-valued variables and fixed length of a loop cycle grow logarithmically along with the number of states of the Turing machine but do not change along the size of input. The larger is the size of work tape the more dense is the information representation in interval-values occuring in this computation – the main advantage of this paradigm, massive parallelism, is deployed. Of course, the length of simulating interval-valued computation cannot be restricted polynomially to capture full computing power of Turing machines. This interval-valued computation sequence contains products not only with $\left[0, \frac{1}{2}\right)$ but we conjecture that one can transform into such a form.

After this achievement, the computing power of interval-valued computing is enough to simulate not only ordinary Turing machines, but even red-green Turing machines, if one allow infinitely looping interval-valued computations on a finite number of interval-valued variables.

We note that it is a novelty to the previous papers [6–10] that input word appears not only as an encoded sequence of operators but it has an explicit representation in the interval-values itself. It somehow takes better advantage of the massive parallelism of this paradigm.

The organization of this paper is the following. In Sect. 2, the preliminaries are enumerated, mainly the notions concerning red-green Turing machines. In Sect. 3, the definitions are given. In Sect. 4, an example of an infinite running interval-valued computation is given where the input is not a discrete word but a specific interval-value. We will extract the binary digits of *any* real number a (in $[0, 1]$) by an interval-valued computation starting on input interval-value $[0, a)$. It is the first example where an interval-valued computing sequence executes a task that is non Turing computable (if a is a non recursively enumerable real).

We can remark that this is the first example where interval-valued computing manifests itself as an analog computing system. In Sect. 5, we state and prove our main results. In Sect. 6 we pose some open problems related to interval-valued computing.

2 Preliminaries

Turing machines are the best known universal models of computation. However, there are well-known non computable problems in these models. Motivated in this way or other way, some theoretical extensions of classical computing theory are developed, these extensions, including hypercomputation models usually deal with infinite computations.

One of the newest extensions of the Turing machine that is capable to compute the "uncomputable" is the red-green Turing machine [3] that we recall briefly. A red-green Turing machine (for us) is essentially a deterministic Turing machine that is performing an ω-computation on a finite input without halting. Its set of states is partitioned to red and green states such that the initial state is a red state. When, during a computation the state is changed from red to green or vice versa, a "mind change" happens. The "acceptance condition" of the computations is based on counting the number of mind changes: a computation is "recognizing" if no red states are visited infinitely often, but there are one or more green states that are visited infinitely often. It is known that all recursively enumerable languages are recognised by computations with at most one mind change. Observe that a red-green Turing machine could reject an input in two essentially different ways: either "stabilizing the computation" in red states (that is never to have a mind change to a green state after a point of the computation, it is equivalent to have an even number of mind changes during the computation) or by never stabilizing in any color (i.e., to have infinitely many mindchanges during the computation). However, the model allows computations recognizing some input after any odd (finite) numbers of mind changes.

Since this model is also capable to recognise the complement of any recursively enumerable language, it is clearly more powerful than the traditional computational model, i.e., traditional Turing machines. We note that there are other computational models proven to be equivalent to the red-green Turing machines, e.g., red-green register machines, and Watson-Crick T0L systems [1,2].

3 Interval-Valued Computations: Definitions

The notions of interval-valued computing have been coined in [4] and formulated first in mathematical precision in [5]. Interval-valued computations were designed to solve decisional problems, e.g., q-SAT [5] and also to compute functions, e.g., prime factorization is done by an interval-valued computation in [7].

In this section we restate the paradigm to remain self-contained.

We extend the notion of accepting runs by allowing ω-runs where some kind of fixation of a given interval-value means acceptance. In this way we describe a model which is already more general than the previously used variants and capable to capture computing power of red-green Turing machines.

An *interval-value* is a subset of the interval $[0,1)$ which is a finite union of left-closed, right-open subintervals. The set of interval-values is denoted by $\mathbb{V}$. The maximal subintervals of an interval-value are called its *components*. Maximality, of course, is to be understood as maximality for inclusion. The first component of the interval-value is its leftmost component in the usual sense: $[a,b)$ is the first component of $v \in \mathbb{V}$ if and only if $[0,a) \cap v = \emptyset$ and $[a,b) \subseteq v$ and $b \notin v$.

Operators of interval-valued computation are Boolean set operators AND, OR, NOT, and three other: $PRODUCT$, $RSHIFT$, $LSHIFT$. NOT is a unary operator, all the others are binary. For the sake of denotational simplicity, we consider also NOT as a binary operator where the second operand is superfluous.

A *computation sequence* is a finite sequence indexed by natural numbers, the first element is the constant $FIRSTHALF$ and every other element consists of a triplet (op, i, j) where op is an operator and i, j are positive integers less than the index of the actual element.

The *value* of an interval-valued computation is defined by induction of the length of the computation. Let S denote an interval-valued computation sequence, its value, denoted by $\|S\|$ is the interval-value that is obtained by the last operation of the sequence S. Let $S_{\to k}$ denote the prefix of S with last element index k. First $\|FIRSTHALF\|$ is fixed to $\left[0, \frac{1}{2}\right)$. If the last element of S is (op, i, j), containing a Boolean operator, then

- $\|S\| = \|S_{\to i}\| \cup \|S_{\to j}\|$ if op is OR,
- $\|S\| = \|S_{\to i}\| \cap \|S_{\to j}\|$ if op is AND,
- $\|S\| = [0,1) \setminus \|S_{\to i}\|$ if op is NOT.

The function $Flength : \mathbb{V} \to \mathbb{R}$ can be defined by $Flength(v) = b - a$ if $[a,b)$ is the first component of v. If such a component does not exist ($v = \emptyset$) then $Flength$ returns 0. The *left-shift* operator will shift the first interval-value to the left by the first-length of the second operand and remove the part which is shifted out of the interval $[0,1)$ (i.e., below 0). As opposed to this, the *right-shift* operator is defined in a circular way, i.e., the parts shifted above 1 will appear at the lower end of $[0,1)$. In this definition we write interval-values in their "characteristic function" notation instead of the above subset notation. That is, for any $A \in \mathbb{V}$ and $x \in [0,1)$, $A(x) = \begin{cases} 1, \text{ if } x \in A, \\ 0, \text{ otherwise.} \end{cases}$

The binary operators $Lshift$ and $Rshift$ on $\mathbb{V}$ are defined in the following way. If $x \in [0,1)$ and $A, B \in \mathbb{V}$, then

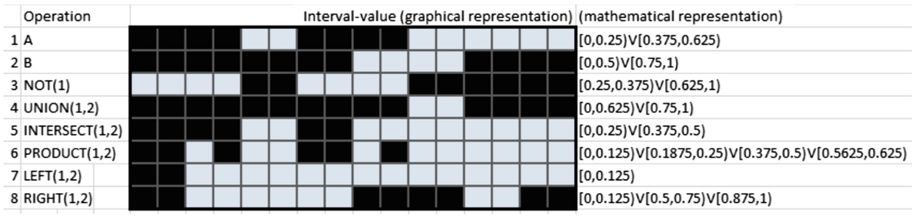
$$Lshift(A,B)(x) = \begin{cases} A(x + Flength(B)), & \text{if } 0 \le x + Flength(B) \le 1, \\ 0, & \text{in other cases} \end{cases}$$

$$Rshift(A,B)(x) = A(frac(x - Flength(B))).$$

Here the function *frac* gives the fractional part of a real number, i.e., $frac(x) = x - \lfloor x \rfloor$, where $\lfloor x \rfloor$ is the greatest integer which is not greater than x.

Let A and B be interval-values and $x \in [0,1)$. Then the *product* of A and B includes x if and only if $B(x) = 1$ and $A\left(\frac{x - x_B}{x^B - x_B}\right) = 1$, where x_B denotes the lower end-point of the B-component including x and x^B denotes the upper end-point of this component, that is, $[x_B, x^B)$ is the maximal subinterval of B containing x. The product of A and B is zooming out the interval-value A onto the components of B.

For visual examples on operations we refer to Fig. 1. Further examples, especially on shifts and product, are shown, e.g., in figures no 1 and 2 in [6], on page 211.



	Operation	Interval-value (graphical representation)	(mathematical representation)
1	A	■■■■□□■■■■□□□□□□	[0,0.25)V[0.375,0.625)
2	B	■■■■■■■■□□□□■■■■	[0,0.5)V[0.75,1)
3	NOT(1)	□□□□■■□□□□■■■■■■	[0.25,0.375)V[0.625,1)
4	UNION(1,2)	■■■■■■■■■■□□■■■■	[0,0.625)V[0.75,1)
5	INTERSECT(1,2)	■■■■□□■■□□□□□□□□	[0,0.25)V[0.375,0.5)
6	PRODUCT(1,2)	■■□■□□■■□■□□□□□□	[0,0.125)V[0.1875,0.25)V[0.375,0.5)V[0.5625,0.625)
7	LEFT(1,2)	■■□□□□□□□□□□□□□□	[0,0.125)
8	RIGHT(1,2)	■■□□□□□□■■■■□□■■	[0,0.125)V[0.5,0.75)V[0.875,1)

Fig. 1. Examples of some interval-valued operators, NOT stands for negation, UNION for or (disjunction), INTERSECT for and (conjunction), LEFT and RIGHT for left- and right-shift, respectively.

As we defined earlier, an *interval-valued computation sequence* is a sequence $S_0, S_1, \ldots$ starting with a constant $FIRSTHALF$ as S_0 and continuing with operator applications on so far constructed interval-values. For example, we denote the fact, that the value S_k is the product of the values of S_i and S_j by $S_k = (PRODUCT, i, j)$. $\|S_{\to k}\|$ means the interval-value of the computation sequence to S_k. As a start, $\|FIRSTHALF\|$ is set to $[0, \frac{1}{2})$. In the example, $\|S_{\to k}\|$ is the product of the values of S_i and S_j.

Let Σ be a finite alphabet, without loss of generality, it can be simply $\{0, 1\}$ and let L be a language over Σ. (If $|\Sigma| > 2$ then, by an appropriate coding, all the following definitions, statements and proofs go through without any essential change, only technical details making them less followable.) In previous papers interval-valued computations are used in the following way to decide a language: $L \subseteq \Sigma^*$ is *decidable by a linear interval-valued computation* if and only if there is a positive constant c and a logarithmic space algorithm $\mathcal{A}$ with the following properties. For each input word $w \in \Sigma^*$, $\mathcal{A}$ constructs an interval-valued computation sequence $\mathcal{A}(w)$ such that $|\mathcal{A}(w)|$ is not greater than $c|w|$ and $w \in L$ if and only if $\|\mathcal{A}(w)\|$ is nonempty. A language $L \subseteq \Sigma^*$ is *decidable by a polynomial interval-valued computation* if and only if there is a positive constant c, an integer $k \geq 0$ and a logarithmic space algorithm $\mathcal{A}$ with the following properties. For each input word $w \in \Sigma^*$, $\mathcal{A}$ constructs an interval-valued computation sequence $\mathcal{A}(w)$ such that $|\mathcal{A}(w)\|$ is not greater than $c|w|^k$ and $w \in L$ if and only if $\|\mathcal{A}(w)\|$ is nonempty.

We note that in this paper we make the input word ($w \in \Sigma^*$) to appear in the computation sequence as an encoded input interval-value. It fits into the previous framework of language decision because we will write a logspace algorithm to construct the mentioned encoded interval-value from w. We underline that it is a novelty (versus the previous papers) that the input word appears not only as an encoded sequence of operators but it has an explicit representation in the interval-value itself.

The computing power of red-green Turing machines is equivalent to that of non-deterministic circular Turing machines [3,13]. To capture this computing power one has to extend the original interval-valued computing system. It is quite natural to consider ω-length interval-valued computation sequences.

The first question arising here is how to measure resource complexity in this kind of interval-valued computations. The computation length is no further relevant, of course. The more relevant and reasonable measure can be the minimal number of needed storage places of interval-values that is enough to execute an infinite computation sequence. Let us call such a storage place a variable. If the interval-value of a member in the computation sequence is not used later (that is, if its index does not occur later in the computation sequence, as operand) then the execution of the sequence can reuse its variable for storing newer computation subresults. It may occur that infinitely many indices are used later in the sequence infinitely often. In this case, the measure is surely infinite. However, in some cases, the execution can be performed, by reuse of variables (we call this recycling), utilizing only a finite number of such storage places. In this cases, we will say that the computation sequence use only a fixed number of interval-valued variables. In this paper we do not give the exact numbers only establish that this number is fixed for a given computation sequence.

The second question arising here is what an appropriate accepting condition can be. We will establish that the following condition is fine for the purpose of capturing the computing power of ordinary Turing machines:

there is an assigned variable Accepting storing an interval-value involved in the computation sequence. The computing sequence is accepting by definition if and only if its interval-value stabilizes in $[0, 1)$. (Normal Acceptance Condition)

If the purpose is to capture computing power of red-green Turing machines:

there are two assigned variables (Red and Green) storing interval-values involved in the computation sequence. The computing sequence is accepting by definition if and only the interval-value of Green infinitely often $[0, 1)$ *but that of Red is not.* (Red-green Acceptance Condition)

4 On Infinite Interval-Valued Computation Sequences

In our computations, the first interval-value is fixed. In this paper, we use either $FIRSTHALF = [0, \frac{1}{2})$ or an input interval $[0, a)$ with a real number $0 \leq a \leq 1$. Because of the fixed first interval-value, it can easily be seen by induction (considering the defined operations on interval-values) that only [ )- type interval components are used in the computations of the next sections. Moreover, if we start the computation sequence with $FIRSTHALF$, each interval-component $[b, c)$ that occurs in the computation in an interval-value has the property that both b and c are of the form $\frac{x}{2^m}$ for some integers $m \geq 1$, $0 \leq x \leq 2^m$.

Actually, the largest value m that is needed to express all components of the interval-values of the computation sequence, gives a kind of resolution of the actual computation. This resolution is connected to the number of used $PRODUCT$ operations: multiplying $FIRSTHALF$ by an interval-value (with the property that its components can be expressed with a value m), the components of the resulted interval-value can be expressed with the number $m' = m+1$. One may assign bit sequences to the interval-values (with resolution m), based on the fact that the segments $\left[\frac{x}{2^m}, \frac{x+1}{2^m}\right)$ (for integer values $0 \leq x \leq 2^m - 1$) are included in the interval-value or not. In this notion, the multiplication by $FIRSTHALF$ doubles the number of stored bits in an interval-value. This measure, the bit-height of a computation, was introduced in [6] where a $PSPACE$-complete problem, namely the q-SAT, was solved by linear interval-valued computation using the product operator only in a way that one of its operand was always $FIRSTHALF$. In this way, an interval-valued computation uses a dynamically changing amount of information in interval-values, i.e., growing number of bits can be coded into an interval-value as the computation proceeds. We note that in other possible variations of interval-valued computing, other types of interval-values can occur, e.g., having components of types $[b, c]$ and/or (b, c). One may also use further interval-values which may not be computable by the described operations from $FIRSTHALF$ by any finite sequences of computation. Such a change may affect the computing power of the system significantly. In this paper we consider infinite computations and in the next part of the section we present computations which uses also an input interval-value.

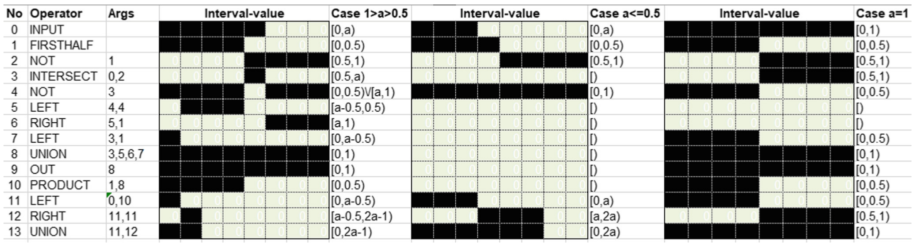
As an example, we demonstrate an infinite interval-valued computation sequence that writes out for an input interval-value $[0, a)$ the binary digits of a, for an arbitrary real $a \in [0, 1]$. The number a can be *any* real, even a Chaitin constant or any not recursively computable real constant. The only "magic" is that the computation gets *exactly* $[0, a)$ as input. What does it mean – output of an interval-valued computation? It is a word over $\{0, 1\}$, initially the empty one. We extend the list of operations by a statement $(OUTPUT, i)$ – it has value that of $\|S_{\rightarrow i}\|$ and as side effect, writes a bit into the output. This bit is true if $\|S_{\rightarrow i}\| = [0, 1)$ and false if $\|S_{\rightarrow i}\| = \emptyset$, other cases remain undefined. It is somewhat different to the definition of output in [7] and in [8] – we want to describe a better separated output.

Instead of $RSHIFT$, $LSHIFT$, AND and OR we use the more readable operator symbols $RIGHT$, $LEFT$, $INTERSECT$, $UNION$, resp. We give the following computation sequence in a somewhat shortened fashion – using the Boolean operator $UNION$ for more than 2 arguments, as abbreviations.

Let S_0 be the input interval-value. Then the computation can be executed in an infinite loop as follows. For any nonnegative integer i, let

$S_{13i+1} = FIRSTHALF$, and
$S_{13i+2} = (NOT, 13i + 1)$,
$S_{13i+3} = (INTERSECT, 13i, 13i + 2)$,
$S_{13i+4} = (NOT, 13i + 3)$,
$S_{13i+5} = (LEFT, 13i + 4, 13i + 4)$,

$S_{13i+6} = (RIGHT, 13i+5, 13i+1),$
$S_{13i+7} = (LEFT, 13i+3, 13i+1),$
$S_{13i+8} = (UNION, 13i+3, 13i+5, 13i+6, 13i+7),$
$S_{13i+9} = (OUTPUT, 13i+8),$
$S_{13i+10} = (PRODUCT, 13i+1, 13i+8),$
$S_{13i+11} = (LEFT, 13i, 13i+10),$
$S_{13i+12} = (RIGHT, 13i+11, 13i+11),$
$S_{13i+13} = (UNION, 13i+11, 13i+12).$



No	Operator	Args	Interval-value	Case 1>a>0.5	Interval-value	Case a<=0.5	Interval-value	Case a=1
0	INPUT		■■■■■000	[0,a)	■■■00000	[0,a)	■■■■■■■■	[0,1)
1	FIRSTHALF		■■■■0000	[0,0.5)	■■■■0000	[0,0.5)	■■■■0000	[0,0.5)
2	NOT	1	0000■■■■	[0.5,1)	0000■■■■	[0.5,1)	0000■■■■	[0.5,1)
3	INTERSECT	0,2	0000■000	[0.5,a)	00000000	[)	0000■■■■	[0.5,1)
4	NOT	3	■■■■0■■■	[0,0.5)V[a,1)	■■■■■■■■	[0,1)	■■■■0000	[0,0.5)
5	LEFT	4,4	0■■■0000	[a-0.5,0.5)	00000000	[)	00000000	[)
6	RIGHT	5,1	00000■■■	[a,1)	00000000	[)	00000000	[)
7	LEFT	3,1	■0000000	[0,a-0.5)	00000000	[)	■■■■0000	[0,0.5)
8	UNION	3,5,6,7	■■■■■■■■	[0,1)	00000000	[)	■■■■■■■■	[0,1)
9	OUT	8	■■■■■■■■	[0,1)	00000000	[)	■■■■■■■■	[0,1)
10	PRODUCT	1,8	■■■■0000	[0,0.5)	00000000	[)	■■■■0000	[0,0.5)
11	LEFT	0,10	■0000000	[0,a-0.5)	■■■00000	[0,a)	■■■■0000	[0,0.5)
12	RIGHT	11,11	0■000000	[a-0.5,2a-1)	000■■■00	[a,2a)	0000■■■■	[0.5,1)
13	UNION	11,12	■■000000	[0,2a-1)	■■■■■■00	[0,2a)	■■■■■■■■	[0,1)

Fig. 2. Writing out the first binary digit of a real number $a \in [0,1]$ for three different possible input numbers. Left: a case, where the first binary digit is 1. Middle: a case when the first binary digit is 0. Right: the special case of $a = 1$, with the whole $[0,1)$ as input. The shown sequences do not only give the first output bit (line 9), but also prepare the new interval-value which can be used (similarly as the input interval-value) to produce the second and further digits of the original input.

This computation, when $i = 0$, writes out the first binary digit of a in its 9-th step and then prepares the interval-value to the next cycle by constructing $[0, 2a-1)$ if $a > \frac{1}{2}$ and just $[0, 2a)$ otherwise. From S_{14}, the previous computation is "recycling" the variables substituting S_{13} into S_0. This means that, actually, we do not need to "store" every interval-value that is obtained for a long time, but during the computation we need to remember only the last 13 interval-values, thus, a fixed constant 13 "slots" of an "interval-valued memory" would be enough to manage this computation.

It is easy to see that this computation writes out while its ω-run all digits of a one by one (if $a = 1$ then it is written out as eternally repeating 1's). Further, it can run utilizing only 13 interval-values (or maybe few more because of using abbreviations in our description). We underline that, apart from our only constant interval-value $FIRTSHALF$, we need a fixed number of interval-values to continue the computation process, recycling the variables.

We could write all the interval-values in such formulae as

$$S_0 = [0, a), S_1 = \left[0, \frac{1}{2}\right), S_2 = \left[\frac{1}{2}, 1\right),$$

and

$$S_3 = \begin{cases} \emptyset & \text{if } a \leq \frac{1}{2} \\ \left[\frac{1}{2}, a\right) & \text{if } a > \frac{1}{2} \end{cases}.$$

To be concise, we write a table showing 3 typical cases and tracking the first 13 steps (see Fig. 2). The main technical difficulty in constructing of this sequence lies in handling the various cases in a uniform way.

5 Interval-Valued Computations for Turing Machine Simulation

In this section a simulation of Turing machines is made possible by a representation of Turing machine configurations using a fixed number of interval-valued variables, where 'fixed' means fixed for a given Turing machine. This number depends logarithmically on the number of states of the machine.

Without any loss of generality, a Turing machine is assumed to have states $\{0, \ldots, 2^s - 1\}$ for some integer $s \geq 0$ with initial state 0 and accepting state set $\{2^s - 1\}$. We think of a state as an s-length bit sequence $(q_1, \ldots, q_s)$. In this view, the null vector maps to the initial state and $q_1 = 1 \wedge \ldots \wedge q_s = 1$ expresses that the state is accepting.

Further, the tape alphabet is restricted to $\{0, 1\}$ and the machine cannot write a blank symbol. The empty word is avoided as input word. These are not essential restrictions on Turing machines concerning only computation universality. Every Turing decidable language can be decided by a Turing machine restricted in these ways, it can be seen introducing a new blank symbol, handling the old one as an ordinary tape symbol and by some other input translations. These restrictions make the following parts of this paper technically more readable.

For the sake of illustration we take an example Turing machine M with state set $\{0, 1, 2, 3\}$ and with the following program:

q_1	q_2	$Tape$	q'_1	q'_2	$Tape'$	$Move$
0	0	0	0	1	1	$\rightarrow$
0	0	1	0	0	0	$\circ$
0	1	0	1	0	0	$\rightarrow$
0	1	1	1	1	1	$\leftarrow$
1	0	0	1	1	0	$\rightarrow$
1	0	1	1	0	1	$\circ$
1	1	0	0	1	0	$\circ$
1	1	1	1	1	1	$\circ$

We use $6 + s$ interval-valued variables to represent the configurations of a Turing machine where s is the number of bits representing the states. We should mention that the following computations use more interval-valued variables than the below named ones – for temporary storage reasons. By careful examination of them (in the following computations), one can exactly determine their number. We will only point out that this cardinality is finite for a given Turing machine. The identifiers of these interval-values are

$$First, Last, End, Head, Tape, Accepting$$

and

$$Q_i \ (1 \le i \le s).$$

As usual, a configuration of Turing machine T consists of

– *a description of the nonempty part w of the tape,*
– *the position p of the head on it and*
– *the description of the actual state q.*

These interval-valued variables describe a configuration (w, p, q) (where $w \in \{0,1\}^*, 1 \le p \le |w|, q = (q_1, \ldots, q_s) \in \{0,1\}^s$) by the following configuration representing rules. Let n be the only nonnegative integer such that $2^n < p \le 2^{n+1}$.

$$First = \left[0, \frac{1}{2^{n+1}}\right)$$

$$Last = \left[1 - \frac{1}{2^{n+1}}, 1\right)$$

$$End = \left[\frac{|w| - 1}{2^{n+1}}, \frac{|w|}{2^{n+1}}\right)$$

$$Tape = \bigcup_{1 \le i \le |w|, w_i = 1} \left[\frac{i-1}{2^{n+1}}, \frac{i}{2^{n+1}}\right)$$

$$Head = \left[\frac{p-1}{2^{n+1}}, \frac{p}{2^{n+1}}\right)$$

$$Q_i = \left[\frac{p-1}{2^{n+1}}, \frac{p}{2^{n+1}}\right) \text{ if } q_i = 1 \text{ and } Q_i = \emptyset \text{ if } q_i = 0.$$

In Fig. 3, an initial configuration of M is shown for the input word 10110 represented by these initial interval-values. For the input word 10110, the value of n is 2, $\frac{1}{2^{n+1}} = \frac{1}{8}$.

We can notice that the reason why a finite number of interval-values is enough to represent all the configurations is that we utilize massively parallel data representation in these interval-values.

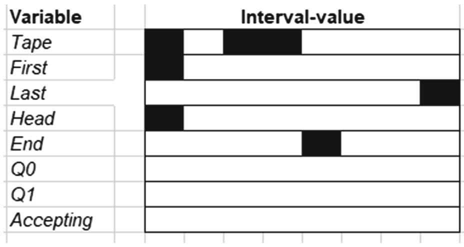


Fig. 3. Interval-values after initialization for input 10110.

Using this representation, we can establish the following theorem:

Theorem 1. *(i) There is a logspace algorithm (let us call it $\mathcal{A}$) that constructs an infinite interval-valued computation sequence to a given deterministic Turing machine T and to a given input word w that is accepting (in the normal sense) if and only if T accepts w. The number of the used interval-valued variables is fixed for a given machine and is growing logarithmically along the number of the states of T.*

(ii) There is a logspace algorithm ($\mathcal{B}$) that constructs an infinite interval-valued computation sequence to a given red-green Turing machine T and to a given input word w that is accepting (in the red-green sense) if and only if T accepts w. The number of the used interval-valued variables is fixed for the given machine and is growing logarithmically along the number of the states of T.

Proof. (i)

The basic difficulty in construction of such a computation lies in the absence of possibility of investigation of cases and switch in the execution accordingly to the result. It must remain uniform in all cases. This problem can be tackled by the following trick. We compute an interval-value that is different in the different cases and use it in a uniform way that works in every case.

The initial part of the computation created by $\mathcal{A}$ has to set up the representation of initial state according to w.

Let $n = max(0, \lceil \log_2(|w|) \rceil - 1)$. It is the least nonnegative integer that $2^n < |w| \leq 2^{n+1}$ (except if $|w| = 1$, in this case we start by a higher density what is really needed for technical simplicity).

For the sake of readability, we always write a new variable for interval-values, but, by a more careful investigation, one can always establish that only a finite number of them is really needed, using recycling. Further, we write $I := PRODUCT(J, K)$ for $I := (PRODUCT, j, k)$ if the index of J and K is just j and k, resp. The same for $RIGHT$, $LEFT$, AND, OR and NOT – the last is written in unary form.

$F := FIRSTHALF$, $I_0 := F$,
$(\forall k \in \{0, \ldots, n-1\})$ $I_{k+1} := PRODUCT(F, I_k)$,
$T_0 := \emptyset$, $S_1 := I_n$,
$(\forall p \in \{1, \ldots, |w|\})$

```
        if $w_p = 1$ then $T_p := T_{p-1} \cup S_p$,
        if $p = |w|$ then $End := S_p$,
        $S_{p+1} := RIGHT(S_p, I_n)$,
$Tape := T_{|w|}$, $First := I_n$, $Last := RIGHT(First, NOT(First))$.
$Head := First$, $Accepting := \emptyset$.
```

This assures that $\|I_n\| = \left[0, \frac{1}{2^{n+1}}\right)$ and all the configuration representing variables are set correctly respect to representing rules above.

In this part, only in this case (later not), we demonstrate what we understand under variable recycling. The length of the initial part of the computation sequence depends linearly from $|w|$, the length of the input word. However, it is enough to use a fixed number of interval-valued variables (F, I, T and S, and some more, for temporary stored results) and recycle them by the following modification of the original algorithm part. We chose writing indices on these variables to understand better the details of the computation. We remark that operation $I := \emptyset$ is also an abbreviation – the empty set can be combined from the value of $FIRSTHALF$ by some Boolean operations.

```
$F := FIRSTHALF$, $I := F$,
$(\forall k \in \{0, \ldots, n-1\})$ $I := PRODUCT(F, I)$,
$T := \emptyset$, $S := I$,
$(\forall p \in \{1, \ldots, |w|\})$
        if $w_p = 1$ then $T := T \cup S$,
        if $p = |w|$ then $End := S$,
        $S := RIGHT(S, I)$,
$Tape := T$, $First := I$, $Last := RIGHT(First, NOT(First))$.
$Head := First$, $Accepting := \emptyset$.
```

We will use a computation sequence that can extract the first component of an arbitrary interval-value I by a fixed length computation, i.e., obtain the interval-value that is exactly the first component of I.

Claim. There exists an interval-valued computation sequence of length less than 10 that starts with an arbitrary interval-value and the value of its last member (the value of the computation) is the first component of the input interval-value.

Proof. The computation can be constructed in the following way. Operation $MINUS$ should be considered as the usual abbreviation for a Boolean combination from AND and NOT. The first component appears in Fc.

$$Fc_1 := LEFT(I, I), \qquad Fc_2 := LEFT(Fc_1, NOT(I)),$$
$$Fc_3 := RIGHT(Fc_2, NOT(I)), \qquad Fc_4 := RIGHT(Fc_3, I),$$
$$Fc := MINUS(I, Fc_4).$$

If $I = [0, b)$ $(0 \le b \le 1)$, then $Fc_1 = Fc_2 = Fc_3 = Fc_4 = \emptyset$ and Fc is I, correctly.

If $I = [a, b)$ $(0 < a < b \le 1)$, then $Fc_1 = [max(2a - b, 0), a)$, $Fc_2 = Fc_3 = Fc_4 = \emptyset$ and Fc is I, correctly.

If $I = [a, b) \cup [c, d) \cup M$ $(0 < a < b < c < d < e \le 1, M \cap [0, e) = \emptyset)$ then $Fc_4 = [c, d) \cup M$ and then Fc is $[a, b)$, correctly and

if $I = [0,b) \cup [c,d) \cup M$ $(0 < b < c < d < e \leq 1, M \cap [0,e) = \emptyset)$ then $Fc_4 = [c,d) \cup M$ and then Fc is $[0,b)$, correctly again.
The computation uses variables $Fc_1, \ldots Fc_4, Fc$ and some temporary interval-values. This accomplishes the proof of this statement. $\square$

We continue now with the proof of Theorem 1. $\mathcal{A}$ (the algorithm to create the computation sequence) will work in an infinite loop from now on, after the initial phase.

The cycle begins here, at *Step 1*: it is to increase tape density if needed – first, because of a right move of machine head. This can be executed by taking product with an interval-value π that takes value either $\left[0, \frac{1}{2}\right)$ or $[0,1)$ depending on density increase is needed for this reason or not. So, first π will be computed as follows.

Let π_0 be the intersection of End, $Last$ and $Head$ and the Boolean combination of $Q_0, \ldots, Q_s, Tape \cap Head$ that expresses the pairs of machine states and tape symbols that prescribe a move to right by the program of T. π_0 will get value $\left[1 - \frac{1}{2^{n+1}}, 1\right)$ if a move-to-right is needed and $\emptyset$ if is not.

Now, $\pi_1 := LEFT(\pi_0, FIRSTHALF)$, $\pi_2 := RIGHT(\pi_1, \pi_1)$,
$\pi_3 := NOT(\pi_2)$.

$\|\pi_3\|$ is $\left[0, \frac{1}{2}\right) \cup \left[\frac{1}{2} + \frac{1}{2^{n+1}}, 1\right)$ if a move-to-right is needed and $[0,1)$ if is not. π can be correctly computed then as the first component of π_3.

If π is got then
$Tape := PRODUCT(\pi, Tape)$,
$First := PRODUCT(\pi, First)$,
$Head := PRODUCT(\pi, Head)$,
$Last := RIGHT(First, NOT(First))$ (this uses the new value of $Left$)
and $End := Head$. (also uses the new value)

We note that these steps do not move the head to the right just set up the "playground" more dense to have space enough to move right later.

Step 2 is to increase tape density if needed - second, because of a left move of machine head. This can be executed also by a product operator involving the appropriately computed interval-value, but also right shifts (with the actual, new value of $Head$) of variables $Tape$, $Head$, $Q_1, \ldots, Q_s$ are needed (to make space for a later move to the left with the head and the state representation interval-values).

In *Step 3*, the computation generated by $\mathcal{A}$ changes the content of actual tape cell. An interval-value δ will be determined that is equal to $Head$ if tape change is needed and $\emptyset$ if not.

Let δ be the union of

- intersection of $Head$, $Tape$ and the Boolean combination of $Q_0, \ldots, Q_s$ and $Head \cap Tape$ that expresses machine states that prescribe a tape cell change from 1 to 0
- intersection of $Head$, $NOT(Tape)$ and the Boolean combination of $Q_0, \ldots, Q_s$ and $Head \cap NOT(Tape)$ that expresses machine states that prescribe a tape cell change from 0 to 1.

If δ is got then $\mathcal{A}$ should generate a step $Tape := Tape\ XOR\ (Head \cap \delta)$ and the new content of the $Tape$ is correctly set. XOR, as expected, can be combined from some Boolean operators in the usual way.

In *Step 4*, in a similar way, the computation generated by $\mathcal{A}$ determines into an interval-valued variable whether change in the first state bit is needed or not and, using XOR, determines the new interval-value of Q_1. We note that the old interval-value of $Tape$ (before Step 3) is needed to do this step. This can be solved by storing temporarily also the previous value of $Tape$. Also the other interval-values for $Q_i (i \in \{1, \ldots, s\})$ will be determined.

In *Step 5*, eventually moving the interval-valued variable *End* to the right is implemented. The computation determines an interval-value μ that the right shift with it is correct in both cases – if a shift is needed or not. It is a copy of the interval-value of $Head$ in the first case and $\emptyset$ in the second one. μ can be computed as the intersection of $Head$, $Tape$, *End* and the Boolean combination of $Q_0, \ldots, Q_s$, $Head$ and $Tape$ that expresses machine state – tape symbol pairs that prescribe a move to the right. Finally *End* is shifted to the right by μ.

Step 6 is similar to Step 5, but concerns left shift of variable *End*.

In *Step 7*, eventually moving the interval-valued variables $Head$ and $Q_1,\ldots,$ Q_s to the right is implemented. The computation determines interval-values ν that the right shift with it is correct in both cases – if a shift is needed or not. It is a copy of the interval-value of $Head$ in the first case and $\emptyset$ in the second one. ν can be computed as the intersection of $Head$, $Tape$ and the Boolean combination of $Q_0, \ldots, Q_s$, $Head$ and $Tape$ that expresses machine state – tape symbol pairs that prescribe a move to the right. Finally $Head$ and $Q_1, \ldots, Q_s$ are shifted to the right by μ.

In *Step 8* is similar to Step 7, but concerns left shift of $Head$ and $Q_1,\ldots,Q_s$.

Finally, in *Step 9*, the value of *Accepting* can be computed from $Q_1 \cap \ldots \cap Q_s$. It is equal to $Head$ if the state is accepting and $\emptyset$ if not. This has to be translated it to $[0, 1)$ if accepting and $\emptyset$ if not. The first part is $X_1 := NOT(Accepting)$, $X_2 := PRODUCT(NOT(FIRSTHALF), Accepting)$,
$X_3 := PRODUCT(FIRSTHALF, Accepting)$. This ensures that the following condition holds:

$X_3 = [a, b)$ for some $0 < a < b < 1$ if *Accepting* is nonempty and X_3 is empty if and only if *Accepting* is empty

The computation can be the following. We use one plus abbreviation, $I = FC(J)$ as an abbreviation to a computation sequence that is starting with the interval-value of J and results in its first component, stored into variable I. We track the values all the named variables for both the two relevant cases in Fig. 4.

After *Step 9*, the computation sequence generated by $\mathcal{A}$ has built all the interval-values $Tape$, $Head$, etc. that represent the next configuration, thus restarting it at *Step 1* will result a loop that eternally generates, before *Step 1*, the next configurations of T one after the other. The acceptance condition clearly should be applied to the interval-value *Accepting*. If Turing machine T accepts w after n steps, then the computation sequence turns to be accepting

Operation	value if $X_3 = [a, b)$	value if $X_3 = \emptyset$
$X_4 := NOT(X_3)$,	$[0, a) \cup [b, 1)$	$[0, 1)$
$X_5 := FC(X_4)$,	$[0, a)$	$[0, 1)$
$X_6 := PRODUCT(NOT(FIRSTHALF), X_5)$	$[a/2, a)$	$[1/2, 1)$
$X_7 := MINUS(X_4, X_5)$,	$[b, 1)$	$\emptyset$
$X_8 := UNION(X_7, X_6)$,	$[a/2, a) \cup [b, 1)$	$[1/2, 1)$
$X_9 := RIGHT(X_8, X_7)$,	$[0, 1 - b) \cup [1 - b + a/2, 1 - b + a)$	$[1/2, 1)$
$X_{10} := FC(X_9)$,	$[0, 1 - b)$	$[1/2, 1)$
$X_{11} := MINUS(X_9, X_{10})$,	$[1 - b + a/2, 1 - b + a)$	$\emptyset$
$X_{12} := LEFT(X_{11}, X_7)$,	$[a/2, a)$	$\emptyset$
$X_{13} := LEFT(X_{12}, X_{12})$,	$[0, a/2)$	$\emptyset$
$X_{14} := UNION(X_{12}, X_{13})$,	$[0, a)$	$\emptyset$
$X_{15} := (X_{14}, X_3)$,	$[0, b)$	$\emptyset$
$X_{16} := (X_{15}, X_7)$.	$[0, 1)$	$\emptyset$

Fig. 4. Step 9: growing up a nonempty interval in [0,1) while the empty input keeping empty

exactly after n execution of the cycle. If T does not accept w, then the generated interval-valued computation sequence never accepts w.

$\mathcal{A}$ is clearly logspace, it generates a looping interval-valued computation sequence. The syntactic rules that generate the computation sequence are given in the above proof, only the indices should be manipulated by $\mathcal{A}$.

(ii) The proof of *(ii)* differs only the acceptance condition. The generated computation sequence (by $\mathcal{B}$) should differ between the two sets of states in T (red vs. green states). So B introduces two interval-valued variables *Red* and *Green* for continuously storing the "color" of the actual state in these variables – *Red* has interval-value $[0, 1)$ if the state is red and $\emptyset$ if green. *Green* behaves in an opposite way. Based upon the value of $Q_1, \ldots, Q_s$, in each loop cycle the value of *Red* and *Green* can be updated by a Boolean combination of them that describes the red state set. Then *Red* is equal to *Head* if the state is red and empty if green. It should 'expand' the nonempty interval-value input into $[0, 1)$, while keep the empty one as empty. The method is an easy modification of the method used in Step 9. Then the interval-valued computation will accept the input word clearly exactly when T accepts w, as a red-green Turing machine. The acceptance condition should be applied, of course, to interval-valued variables *Red* and *Green*. If T accepts w, then the green Turing machine states occur infinitely often but red states do not. This means that the simulating infinite length interval-valued computation is accepting in the red-green sense. □

We note that in the title we write equivalence of a modification of interval-valued computing and red-reen Turing machines. The given simulation is only one side of an equivalence. The other direction is quite obvious.

6 Conclusions, Further Remarks

It would be interesting to explore what density is allowed by today's optical technology. The investigated massively parallel model of computing remained rather theoretical yet. It seems that it is related to quantum computing (QC) in its high inner parallelism. It is worth to investigate the relation between problems solvable by this paradigm effectively and problems solvable by QC.

From the mathematical point of view, some interesting questions remain open. Is the first-order theory of the Boolean algebra of interval-values extended by the operations shifts and product decidable? Or at least, is the set of its valid equations decidable?

From the infinite computational point of view, one can ask: is it possible to encode a complete computation sequence of a Turing machine into one interval-value, if one allow to take a new fixpoint operation what is similar to the star of regular expressions?

References

1. Csuhaj-Varjú, E., Freund, R., Vaszil, G.: A connection between red-green turing machines and watson-crick T0L systems. In: Durand-Lose, J., Nagy, B. (eds.) MCU 2015. LNCS, vol. 9288, pp. 31–44. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23111-2_3
2. Csuhaj-Varjú, E., Freund, R., Vaszil, G.: Watson-Crick T0L systems and red-green register machines. Fundamenta Informaticae **155**(1–2), 111–129 (2017)
3. van Leeuwen, J., Wiedermann, J.: Computation as an unbounded process. Theoret. Comput. Sci. **429**, 202–212 (2012)
4. Nagy, B.: An interval-valued computing device. In: CiE 2005, Computability in Europe: New Computational Paradigms, Amsterdam, Netherlands, pp. 166–177 (2005)
5. Nagy, B., Vályi, S.: Solving a PSPACE-complete problem by a linear interval-valued computation. In: CiE 2006, Computability in Europe: Logical Approaches to Computational Barriers. University of Wales, Swansea, UK, pp. 216–225 (2006)
6. Nagy, B., Vályi, S.: Interval-valued computations and their connection with PSPACE. Theoret. Comput. Sci. **394**, 208–222 (2008)
7. Nagy, B., Vályi, S.: Prime factorization by interval-valued computing. Publicationes Mathematicae Debrecen **79**, 539–551 (2011)
8. Nagy, B., Vályi, S.: Computing discrete logarithm by interval-valued paradigm. Electron. Proc. Theoret. Comput. Sci. **143**, 76–86 (2014)
9. Nagy, B., Vályi, S.: A characterization of NP within interval-valued computing. In: Durand-Lose, J., Nagy, B. (eds.) MCU 2015. LNCS, vol. 9288, pp. 164–179. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23111-2_11
10. Nagy, B., Vályi, S.: A shift-free characterization of NP within interval-valued computing. Fundamenta Informaticae **155**(1–2), 187–207 (2017)
11. Pratt, V.R., Rabin, M.O., Stockmeyer, L.J.: A characterization of the power of vector machines. J. Comput. Syst. Sci. **12**, 198–221 (1976)
12. Woods, D., Naughton, T.: An optical model of computation. Theoret. Comput. Sci. **334**, 227–258 (2005)
13. Wiedermann, J., van Leeuwen, J.: Non-classical turing machines: extending the notion of computation. Proc. NCMA **2017**, 29–40 (2017)

Physical Computation and First-Order Logic

Richard Whyman(✉)

The University of Leeds, Leeds, West Yorkshire LS2 9JT, UK
mmrajw@leeds.ac.uk

Abstract. We introduce a computational formalism that is deployable within an arbitrary logical system. This formalism is intended to capture computation on an arbitrary system, both physical and unphysical, including quantum computers, Blum-Shub-Smale machines, and infinite time Turing machines. We demonstrate that for finite problems, the computational power of any device describable via a finite first-order theory is equivalent to that of a Turing machine. Whereas for infinite problems, their computational power is equivalent to that of a type-2 machine.

Keywords: Physical computation · Computable analysis
Blum-Shub-Smale machines · Hypercomputation
Computability theory · The Church-Turing thesis

1 Introduction

Since the 1930's various models of computation have been devised and indeed utilised. While some of these have been grounded in what is clearly physically possible (e.g. Turing machines [6,22], and quantum computers [16]), others have instead found utility in computing in ways that, whilst not proven to be physically impossible, are questionably achievable (e.g. Type-2 machines [24], Blum-Shub-Smale machines [4], and infinite time Turing machines [12]).

The diverse inequivalent nature of these formulations presents the question of what a computation actually is, and if computation can be "unphysical" then where does the boundary between "physical" and "unphysical" computation lie? For example, is it the transfinite aspect of an infinite time Turing machine that makes it "unphysical"? If so, then why is a quantum computer able to "physically" compute with an infinite continuous space?

One resolution to these questions is to invoke the Church-Turing (CT) thesis [6,7,22], often rendered as *"Every effectively calculable function is computable by a Turing machine,"* and assume that it applies to any physical process we compute with. This may seem to be a natural assumption, as the physical world provides us with the means to calculate. However, the Turing machine was designed to mimic how a *person* mathematically calculates something, rather than how a physical system might go about obtaining it. There are many aspects of physics we do not yet understand, so we cannot in good faith disregard computational structures capable of violating the CT thesis and label them "unphysical".

Indeed, even if we were to assume that the CT thesis defines physical computation, its implications are unclear. As given an arbitrary system $\mathcal{S}$ ("physical" or not) it is not immediately apparent what a computation on $\mathcal{S}$ actually *is*. Whilst a computation may typically be thought of as a sequential process governed by an algorithm [11], such a process requires an ordered acyclic notion of time to be present within $\mathcal{S}$. Not only does this effectively preclude computers which make use of closed time loops [2], it also does not accurately reflect our understanding of many real-world physical systems. For example, if we want to describe the action of a fluid-mechanical system we use differential formulas such as the Navier-Stokes equations [18]. However, this description does not directly tell us how the system evolves at each moment in time, and since there remains no general solution to the Navier-Stokes equations, we are in general unable to extract an algorithm detailing its evolution.

So in an attempt to resolve the issue of what a computation with an arbitrary system is, we introduce the concept of a **theory machine**, which will be formally defined in Sect. 2.1. Theory machines are similar to sequential abstract-state machines [11] as well as evolving multialgebras [10], in that their computations occur on logical structures and attempt to correspond exactly to the computational models that they are describing. However unlike sequential abstract-state machines and evolving multialgebras their computations do not necessarily occur sequentially. Instead a theory machine's computations are able to occur in an atemporal manner by ensuring that the laws of the system are consistently satisfied by the entire structure as a whole, rather than depending on a causally ordered sequence of operations.[1]

A theory machine consists primarily of a set of logical sentences $\mathcal{T}$ which describes the necessary aspects of a system that we wish to compute with. A set of admissible inputs $\mathcal{I}$ and measurable outputs $\mathcal{O}$ are also part of a theory machine. A key aspect of a theory machine is that for any $\Phi \in \mathcal{I}$ we can obtain a structure $\mathfrak{P}$ in which $\mathfrak{P} \models \mathcal{T} \cup \Phi$. Additionally, for each $\Phi \in \mathcal{I}$, there should be at most one output $\Theta \in \mathcal{O}$ such that $\mathcal{T} \cup \Phi \models \Theta$, and for any $\Psi \in \mathcal{O}$ if $\Psi \neq \Theta$ then there is no structure which satisfies $\mathcal{T} \cup \Phi \cup \Theta \cup \Psi$. In which case given *any* structure $\mathfrak{P}$ satisfying $\mathcal{T} \cup \Phi$, if we know that $\Theta \in \mathcal{O}$ is true in $\mathfrak{P}$, then we know that $\mathcal{T} \cup \Phi \models \Theta$. We then take $\mathcal{T} \cup \Phi \models \Theta$ to mean that the output Θ is computed by the machine on input Φ.

This mimics what happens when a person uses a physical system to carry out a computation process. Indeed, as Horsman et al. argued in [14], for a person to be capable of computing with a physical system they must be able to abstractly represent the system and also possess a reasonably correct theory detailing how the system behaves in the representation. Theory machines are based on this argument, representing a physical system as a logical structure and describing its theory as a logical theory $\mathcal{T}$.

[1] N.B. despite the similar name, theory machines are not related to the concept of a "logic theory machine" found in [19].

For example, suppose we wish to use a kinematic system of billiard balls to carry out a computation. Such a system can be represented by a structure with domain $\mathbb{R}$ and functions that specify the locations and velocities of the balls. Hence we may theorise that $\mathbb{R}$ satisfies the axioms of real arithmetic [1,23] and the functions obey the axioms of Newtonian mechanics. Together these axioms may form our theory $\mathcal{T}$ for the system (Newtonian mechanics may not be a perfect description of reality, but in many cases it is more than good enough for making reasonable predictions).

Each input $\Phi \in \mathcal{I}$ could consist of a non-contradictory description of the positions and velocities of the balls at some initial time t_0. This description should state what we *know* to be true about the input configuration, such knowledge should arguably have a degree of uncertainty around it. For example, part of Φ could state that at time t_0 the x-coordinate of the 1st ball is located between rationals q_1 and q_2. Whereas each output $\Theta \in \mathcal{O}$ could be a description of a finite precision position measurement at some final time t_1.

As this is a real physical situation and Φ can be satisfied within finite error bounds, it should always be possible to create a kinematic scenario from t_0 to t_1 in which $\mathcal{T} \cup \Phi$ is satisfied. Due to imprecision of the input there are likely to be many distinct scenarios that satisfy $\mathcal{T} \cup \Phi$. However if we know that in each of them only the output Θ is true, then the exact scenario created does not matter, and the computation may be reliably achieved.

We believe that the computational aspects of any physical computational device can be described by a theory machine. Indeed in Sect. 3 we demonstrate how theory machines can be used to describe Turing machines, type-2 machines, and Blum-Shub-Smale machines. Before explaining how the same is true for quantum computers, fluid-based computers, and infinite time Turing machines. Notably, the CT thesis is violated by some of these forms of computation, so in Sect. 4 we restrict ourselves to finite first-order theory machines. We then prove that a finite (respectively infinite) word function is decidable by a finite first-order theory machine if and only if it is computable by a Turing (respectively type-2) machine. Consequently, if we are to assume that the CT thesis applies to real world calculations, then the computational capabilities of any physically realisable system must be describable by a finite first-order theory machine.

Along with the above result we believe that the concept of a theory machine could serve as a useful tool to help us understand how various different models of computation compare to one another.

2 The Theory Machine

Before defining what a theory machine is, we should firstly make clear what we mean when we refer to a logical system [21]. Informally, a logical system is a formal system in which a semantic consequence relation can be defined, without which a theory machine would be unable to compute. Examples of logical systems include first-order logic, second-order logic, modal logic, and fuzzy logic [3,8,9]. When defining a theory machine we will avoid restricting ourselves to a particular logical system, so as to not preclude any possible form of physical computation.

Formally, each **logical system** $\mathfrak{LS}$ has a fixed set of typed logical symbols $\mathcal{S}$, as well as a collection of principals detailing how to construct the formulas of $\mathfrak{LS}_{\mathcal{V}}$ from $\mathcal{S}$, a set of typed symbols $\mathcal{V}$, and an unbounded set of variables.

A structure $\mathfrak{A}$ of $\mathfrak{LS}_{\mathcal{V}}$ consists of a non-empty set of values V, along with an operator $O_t :\subseteq V^{d_t} \to V$ for each $t \in \mathcal{V}$ and some $t \in \mathcal{S}$. Each logical system $\mathfrak{LS}$ has a well-defined notion of semantics detailing whether or not each formula ϕ of $\mathfrak{LS}_{\mathcal{V}}$ is true in $\mathfrak{A}$. This truth depends on the operators of $\mathfrak{A}$ and which elements of V the free variables of ϕ are assigned to. An **$\mathfrak{LS}_{\mathcal{V}}$-sentence** is a formula of $\mathfrak{LS}_{\mathcal{V}}$ in which there are no free variables.

We refer to the set $\mathcal{V}$ as a **vocabulary** of $\mathfrak{LS}$. For first order-logic the fixed set $\mathcal{S}$ consists of the logical connectives and quantifiers $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \forall, \exists\}$. Whereas a vocabulary for first order logic is a set of relations, functions, and constant symbols (e.g. $\mathcal{V} = \{<, +, 0\}$). A sentence of first-order logic either has no variables (e.g. $0 < 1$), or every variable is within the scope of a quantifier (e.g. $\forall x(0 < (x + 1))$).

Let $\mathfrak{B}$ be an $\mathfrak{LS}_{\mathcal{V}}$-structure, and Φ, Θ be sets of $\mathfrak{LS}_{\mathcal{V}}$-sentences. We say that $\mathfrak{B}$ is an **$\mathfrak{LS}_{\mathcal{V}}$-model** of Φ if every sentence of Φ is true in $\mathfrak{B}$, and denote this by $\mathfrak{B} \models_{\mathfrak{LS}_{\mathcal{V}}} \Phi$. We write $\Phi \models_{\mathfrak{LS}_{\mathcal{V}}} \Theta$ if every $\mathfrak{LS}_{\mathcal{V}}$-structure that models Φ also models Θ. If Θ has an $\mathfrak{LS}_{\mathcal{V}}$-model then Θ is **$\mathfrak{LS}_{\mathcal{V}}$-satisfiable**.

Though the definition of a logical system here might seem rather vague. Our key results and examples will each be given in terms of well-defined and well-studied systems of logic (namely first-order logic (FO) and second-order logic with equality ($SO^{=}$)), so our conclusions will be similarly well-defined.

2.1 The Definition of a Theory Machine

We shall now define our principal concept of a theory machine and its method of computation.

Definition 2.1. *Let $\mathfrak{LS}$ be a logical system and $\mathcal{V}$ a vocabulary of $\mathfrak{LS}$, an $\mathfrak{LS}_{\mathcal{V}}$-**theory machine** is a triple $\mathcal{M} = (\mathcal{T}, \mathcal{I}, \mathcal{O})$ where:*

- *$\mathcal{T}$ is a set of $\mathfrak{LS}_{\mathcal{V}}$-sentences,*
- *$\mathcal{I}$ and $\mathcal{O}$ are sets of sets of $\mathfrak{LS}_{\mathcal{V}}$-sentences,*
- *For every $\Phi \in \mathcal{I}$ the set $\mathcal{T} \cup \Phi$ is $\mathfrak{LS}_{\mathcal{V}}$-satisfiable,*
- *For every $\Phi \in \mathcal{I}$ and $\Theta, \Psi \in \mathcal{O}$ if $\Theta \neq \Psi$ then the set $\mathcal{T} \cup \Phi \cup \Theta \cup \Psi$ is not $\mathfrak{LS}_{\mathcal{V}}$-satisfiable.*

*We call $\mathcal{T}$ the **theory** of $\mathcal{M}$, call $\mathcal{I}$ the set of **inputs** of $\mathcal{M}$, and call $\mathcal{O}$ the set of **outputs** from $\mathcal{M}$.*
*We say that $\mathcal{M}$ **computes** $\Theta \in \mathcal{O}$ from $\Phi \in \mathcal{I}$ if $\mathcal{T} \cup \Phi \models_{\mathfrak{LS}_{\mathcal{V}}} \Theta$. We denote this by $\mathcal{M}(\Phi) = \Theta$. If for $\Theta, \Psi \in \mathcal{O}$ where $\Theta \neq \Psi$ there exists an $\mathfrak{LS}_{\mathcal{L}}$-model of $\mathcal{T} \cup \Phi$ where Θ is true and another where Ψ is true then $\mathcal{M}$ does not compute anything on input Φ and $\mathcal{M}(\Phi)$ is undefined.*

For a given physical computation system described by a theory machine $\mathcal{M}$, the theory $\mathcal{T}$ is intended to detail the laws that the system obeys.

Each element of $\mathcal{I}$ is intended to be a description of some variable input configuration (e.g. the positions of a collection of dials), it could be finite and word-like, it could be an infinite real, it could be a function on reals, or any number of possibilities. Whatever the case, if an object can be exactly defined by some set of properties then it can be inputted into a theory machine. The same is true for the outputs $\mathcal{O}$, allowing us to take the output from one theory machine and plug it in as an input to another theory machine.

Example 2.1. A set of sentences that defines the real number $c \in [0,1)$ with binary expansion $0.b_0b_1\ldots$ is $\left\{T^k(c) \bowtie_{b_k} \frac{1}{2}\right\}_{k\in\mathbb{N}}$. Where $T(x) = 2x - \lfloor 2x \rfloor$, and $\bowtie_0 \equiv <$ and $\bowtie_1 \equiv \geqslant$.

In any theory machine we require that for each input Φ there always exists a model in which Φ and the machine's theory are satisfied, as if this was not the case then inputting Φ would just not make sense within the system.

We also intend for it to be the case that every model of $\mathcal{T} \cup \Phi$ provides only the output computed by the machine (if one exists). Hence the fourth condition of Definition 2.1, which means that not only can we not have $\mathcal{M}(\Phi) = \Theta$ and $\mathcal{M}(\Phi) = \Psi$ for $\Theta \neq \Psi$, but there do not even exist two separate $\mathfrak{LS}_{\mathcal{V}}$-models of $\mathcal{T} \cup \Phi$ in which Θ and Ψ are true.

Example 2.2. Let $\mathcal{V} = \{R, f, c\}$ where R is a unary relation, f a unary function, and c a constant. A simple example of a $FO_{\mathcal{V}}$-theory machine is $\mathcal{M} = (\mathcal{T}, \mathcal{I}, \mathcal{O})$ where:

- $\mathcal{T} = \{\forall x(R(x) \leftrightarrow R(f(x)))\}$,
- $\mathcal{I} = \{\{R(c)\}, \{\neg R(c)\}\}$,
- $\mathcal{O} = \{\{R(f(c))\}, \{\neg R(f(f(c)))\}\}$.

We then have $\mathcal{M}(\{R(c)\}) = \{R(f(c))\}$ as in any model of $\mathcal{T}$, if $R(c)$ is true then $R(f(c))$ must also be true, so $\mathcal{T} \cup \{R(c)\} \models_{FO_{\mathcal{V}}} \{R(f(c))\}$. Whereas $\mathcal{M}(\neg R(c)) = \{\neg R(f(f(c)))\}$ as given $\neg R(c)$ by $\mathcal{T}$ we then have $\neg R(f(c))$ is true and so $\neg R(f(f(c)))$ is true, hence $\mathcal{T} \cup \{\neg R(c)\} \models_{FO_{\mathcal{V}}} \{\neg R(f(f(c)))\}$.

For simplicity, in the examples and definitions below our logical systems will contain the equality "=" as part of the fixed symbols of the system and in any structure of the system "=" will satisfy the usual equality axioms [9] of being an equivalence relation which preserves functions and relations.

Remark 2.1. In first-order logic if the number of symbols in the vocabulary $\mathcal{V}$ is finite then we do not need equality to be part of the system. We just need to add to the theory a finite set of sentences $EQ_{\mathcal{V}}$ that state that $= \in \mathcal{V}$ is an equivalence relation which preserves each of the functions and relations of $\mathcal{V}$.

2.2 Describing Words as Sets of Logical Sentences

As many examples of computation systems take words as their inputs and outputs we naturally require a standard manner in which to write words as logical sentences. Here we do this by assigning the values of a well-behaved sequence of ground terms [11] to the symbols in the word.

Definition 2.2. *We call a sequence of ground terms $\{\chi_i\}_{i\in\mathbb{N}}$ a* ***simple sequence*** *if every element is of the form $\chi_i = \gamma(\sigma^i(\delta))$ where δ is a ground term, and $\gamma(x)$ and $\sigma(x)$ are terms with a single free variable x.*

The idea behind a simple sequence of terms is that it expresses the repeated application of a function, such as the "next symbol on the right" function. Hence it can be easily and simply constructed. For a unary function f and a constant c the sequence of terms $\{f^i(c)\}_{i\in\mathbb{N}}$ is a simple sequence, as is $\{(g \circ f)^i(h(c,c))\}_{i\in\mathbb{N}}$ for a unary function g and a binary function h.

Definition 2.3. *Let $\mathcal{X} = \{\chi_i\}_{i\in\mathbb{N}}$ be a simple sequence. For a set of constants Σ with $\mathbf{B} \notin \Sigma$, the* ***finite $\mathcal{X}$-word set*** *corresponding to the finite word $w = w_0 w_1 \cdots w_n \in \Sigma^*$ is:*

$$\Phi_{\mathcal{X}}^{w} = \bigcup_{i=0}^{n} \{\chi_i = w_i\} \cup \{\chi_{n+1} = \mathbf{B}\}.$$

The ***set of finite $\mathcal{X}$-word sets*** *from an alphabet Σ is $\hat{\Sigma}_{\mathcal{X}}^{*} = \{\Phi_{\mathcal{X}}^{w} \mid w \in \Sigma^*\}$. The* ***infinite $\mathcal{X}$-word set*** *corresponding to the infinite word $u = u_0 u_1 \cdots \in \Sigma^\omega$ is:*

$$\Psi_{\mathcal{X}}^{u} = \bigcup_{i=0}^{\infty} \{\chi_i = u_i\}.$$

The ***set of infinite $\mathcal{X}$-word sets*** *from an alphabet Σ is $\hat{\Sigma}_{\mathcal{X}}^{\omega} = \{\Psi_{\mathcal{X}}^{u} \mid u \in \Sigma^\omega\}$.*

Hence a finite $\mathcal{X}$-word set $\Phi_{\mathcal{X}}^{w}$ maps each term χ_i of $\mathcal{X}$ to the ith symbol in w. So whilst $\mathcal{X}$ may be "simple" the $\mathcal{X}$-word set may be arbitrarily complex.

The symbol $\mathbf{B}$ is intended to represent the "blank" symbol, hence $\chi_{n+1} = \mathbf{B}$ implies that this is the end of the word. Note that if $\chi_i = \gamma(\sigma^i(\delta))$ then by adding the sentence $\forall x((\gamma(x) = \mathbf{B}) \rightarrow (\gamma(\sigma(x)) = \mathbf{B}))$ to the theory of a machine with inputs from $\hat{\Sigma}_{\mathcal{X}}^{*}$ we can ensure that $\chi_j = \mathbf{B}$ for each $j > n$. Conversely as an infinite word has no end there is no need for a blank symbol or the above sentence.

Remark 2.2. Rather than invoking sentences constructed from simple sequences we could have instead just required that there be a computable mapping from the sets of words to the input and output sets. However, our intention here to construct a concept of computation that does not itself rely on another formulation of computation.

3 Examples of Theory Machines

We shall now demonstrate how various well-known examples of computation can be described by theory machines in second-order logic with equality ($SO^=$).

3.1 Turing Machines

Let N be a deterministic halting Turing machine which computes the function $f : \Sigma^* \to \Gamma^*$. For simplicity, we shall take N's tape to be infinite in only the rightwards direction, with a symbol $\mathbf{L}$ marking its leftmost tape cell. As it will be relevant for future examples we will also take N to be a multi-tape Turing machine, with the input written on tape 0 and the output written on tape 1. The output consists of the tape 1's contents to the immediate right of $\mathbf{L}$ up to the first blank symbol.

Let N have tape alphabet $\Lambda \supset \Sigma \cup \Gamma$, including "blank" tape symbol $\mathbf{B} \notin \Sigma \cup \Gamma$, and $\mathbf{L} \notin \Sigma \cup \Gamma$. Let N use the set of internal states Π with initial state $s_0 \in \Pi$ and halting state $s_1 \in \Pi$, and follow the set of rules $\mathcal{R} = \{\rho_l\}_{l=1}^{K}$. For each l the rule ρ_l is of the form:

$$(t_l, b_l, i_l; u_l, c_l, j_l; p_l, k_l) \in (\Pi \setminus \{s_1\}) \times \Lambda \times \mathbb{N} \times \Pi \times \Lambda \times \mathbb{N} \times \{LEFT, RIGHT\} \times \mathbb{N},$$

which is read as "if the machine is in internal state t_l reading b_l from the i_lth tape then go to state u_l, replace the symbol being pointed to on the j_lth tape with c_l, and move p_l on the k_lth tape." To avoid halting early, for each $t \in \Pi \setminus \{s_1\}$ and $b \in \Lambda$ there is a rule of $\mathcal{R}$ beginning with (t, b). To prevent non-determinism, every a rule of $\mathcal{R}$ that begins with t must look at the same tape, so $i = g(t)$ for some function g. To avoid falling off the left end if $b_l = \mathbf{L}$ then $(c_l, j_l; p_l, k_l) = (\mathbf{L}, i, RIGHT, i)$ in ρ_l, and if $b_l \neq \mathbf{L}$ then $c_l \neq \mathbf{L}$.

We can then describe N as an $SO^{=}_{\mathcal{V}_{TM}}$-theory machine $\mathcal{N} = (\mathcal{TM}_{\mathcal{R}}, \hat{\Sigma}^*_{\mathcal{X}}, \hat{\Gamma}^*_{\mathcal{Y}})$, with vocabulary $\mathcal{V}_{TM} = \mathcal{V}_{PA} \cup \Lambda \cup \Pi \cup \{C, H, I, h\}$.

Where $\mathcal{V}_{PA} = \{<, \leqslant, +, \times, 0, 1\}$ are the usual symbols of Peano arithmetic, and $\Lambda \cup \Pi$ consists of constant symbols. C, H, I are functions such that for time step x, cell number y and tape number z; the cell contents are given by $C(x, y, z)$, the cell pointed to by the head is given by $H(x, z)$, and the machine's internal state is given by $I(x)$. The halting time is represented by the constant symbol h, its value depends on the input.

To input and output $\mathcal{N}$ uses the simple sequences $\mathcal{X} = \{C(0, \hat{n} + 1, 0)\}_{n \in \mathbb{N}}$ and $\mathcal{Y} = \{C(h, \hat{n} + 1, 1)\}_{n \in \mathbb{N}}$, where for each $n \in \mathbb{N}$ we denote $\underbrace{1 + \cdots + 1}_{n \text{ times}}$ by $\hat{n}$.

The theory of $\mathcal{N}$ is $\mathcal{TM}_{\mathcal{R}} = PA \cup IT_{\mathbf{B}} \cup ET_{\mathcal{R}} \cup HT_{s_1}$, where PA is the set of Peano arithmetic axioms [15], including the second-order induction axiom.[2] The set $IT_{\mathbf{B}}$ defines the initial configuration of the machine, $ET_{\mathcal{R}}$ describes the evolution of the machine, and HT_{s_1} ensures the machine halts when it reaches s_1. As $PA \subset \mathcal{TM}_{\mathcal{R}}$ any model of $\mathcal{TM}_{\mathcal{R}}$ must be an expansion of the usual structure of the natural numbers $\langle \mathbb{N}; <, \leqslant, +, \times, 0, 1 \rangle$ [15]. Explicitly, the initial configuration is given by:

$$IT_{\mathbf{B}} = \left\{ \begin{array}{l} \forall z((H(0,z) = 1) \wedge (C(0,0,z) = \mathbf{L}) \wedge (C(0,1,z+1) = \mathbf{B})), \\ \forall y \forall z((C(0,y,z) = \mathbf{B}) \to (C(0,y+1,z) = \mathbf{B})), \\ I(0) = s_0 \end{array} \right\}.$$

[2] This is the only second-order sentence in the theory of $\mathcal{N}$, and as we shall see in Example 4.2, it is unnecessary for describing a Turing machine computation.

So by $IT_{\mathbf{B}}$, in any model $\mathfrak{A}$ of $\mathcal{TM}_{\mathcal{R}} \cup \Phi^w_{\mathcal{X}}$ the head of every tape points to cell 1 at time 0, the left-most cell of each tapes contains $\mathbf{L}$, which is followed by a $\mathbf{B}$ in every tape except for the 0th one. The input $\Phi^w_{\mathcal{X}} = \bigcup_{i=0}^{|w|-1}\{C(0, \widehat{i+1}, 0) = w_i\} \cup \{C(0, \widehat{|w|+1}, 0) = \mathbf{B}\}$, specifies that at time 0 the word $w \in \Sigma^*$ is written on the 0th tape followed by a $\mathbf{B}$. So by the second sentence of $IT_{\mathbf{B}}$, at time 0 the contents of every cell that is neither defined by the input nor on the far end is blank. We also have that at time 0 the internal state is s_0, hence the initial configuration of $\mathfrak{A}$ must be the same as it is for N with input w. For evolving this configuration we have:

$$ET_{\mathcal{R}} = \left\{ \forall x(\mu_{(t_l, b_l, i_l)}(x, x) \to (\mu_{(u_l, c_l, j_l)}(x+1, x) \wedge \pi_{(p_l, k_l)}(x) \wedge \nu_{j_l}(x))) \right\}_{l=1}^{K}.$$

Each sentence of $ET_{\mathcal{R}}$ implements a rule of $\mathcal{R}$ via the following three sorts of terms. Firstly for each $s \in \Pi$, $a \in \Lambda$, and $n \in \mathbb{N}$ we have the term:

$$\mu_{(s,a,n)}(x_1, x_2) \equiv (I(x_1) = s) \wedge (C(x_1, H(x_2, \hat{n}), \hat{n}) = a),$$

which indicates that at time x_1 the internal state is s, and the cell pointed to by the nth head at time x_2 contains an a at time x_1. Secondly for $p \in \{LEFT, RIGHT\}$ and $n \in \mathbb{N}$ we have the term:

$$\pi_{(p,n)}(x) \equiv \begin{cases} H(x+1, \hat{n}) = H(x, \hat{n}) + 1 & \text{if } p = \text{RIGHT}, \\ H(x+1, \hat{n}) + 1 = H(x, \hat{n}) & \text{if } p = \text{LEFT}, \end{cases}$$

that states that at the time step after x the head on tape n is shifted to either the succeeding or preceding tape position. Finally for each $n \in \mathbb{N}$ we have:

$$\nu_n(x) \equiv \forall y \forall z((\neg(z = \hat{n}) \vee \neg(H(x, z) = y)) \to (C(x, y, z) = C(x+1, y, z))),$$

which ensures that the tape contents of any cell that is not on tape n or is not being pointed to by a tape head, is preserved moving from time x to time $x+1$. For halting we have:

$$HT_{s_1} = \{\forall x((I(x) = s_1) \leftrightarrow (h \leqslant x))\}.$$

So the halting time h is the first time step of $\mathfrak{A}$ that is in the state s_1. The output $\Phi^v_{\mathcal{Y}} = \bigcup_{i=0}^{|v|-1}\{C(h, \widehat{i+1}, 0) = v_i\} \cup \{C(h, \widehat{|v|+1}, 0) = \mathbf{B}\}$, is therefore defined at this time.

Hence the configurations of $\mathfrak{A}$ evolve from time 0 exactly as they do in the Turing machine N with input w, and likewise the model outputs when the halting state s_1 is reached. As $\mathbf{B} \notin \Gamma$, the output is unique and is completely determined by what happens prior to time h, thus it cannot be affected by whatever occurs afterwards in $\mathfrak{A}$. Therefore by induction and the fact that N is deterministic and halting, $\mathfrak{A} \models_{SO^{=}_{\mathcal{V}_{TM}}} \Phi^v_{\mathcal{Y}}$ if and only if $v = f(w)$.

3.2 Type-2 Machines

Type-2 machines [24] generalise the concept of a Turing machine by enabling it to take infinite inputs and give infinite outputs. To output an infinite word a type-2 machine must never halt and after writing each output symbol on its output tape it moves right and never change that symbol. This way we can stop a type-2 machine at any point and know that what it has written on the output tape must be a correct initial segment of the output. Besides this a type-2 machine behaves identically to a multitape Turing machine.

Let TN be a type-2 machine taking inputs from Σ^ω and giving outputs in Γ^ω. Let TN have rule set $\mathcal{U}$, along with alphabet Λ and internal states Π as in Subsect. 3.1. We can then describe TN as an $SO^{=}_{\mathcal{V}_{TM}}$ -theory machine $\mathcal{TN} = (\mathcal{TTM}_\mathcal{U}, \hat{\Sigma}^\omega_\mathcal{X}, \hat{\Gamma}^\omega_\mathcal{Y})$, where $\mathcal{X}$ and $\mathcal{Y}$ are also the same as in Subsect. 3.1, whilst the theory is $\mathcal{TTM}_\mathcal{U} = PA \cup IT_\mathbf{B} \cup TET_\mathcal{U}$.

TN never halts, so there are no halting conditions, however h is still present in the vocabulary and its purpose continues to be in defining the output. The set $TET_\mathcal{U}$ is the same as $ET_\mathcal{U}$ except that for each rule that writes on the output tape, the term $\mu_{(u,c,1)}(x+1,x)$ is replaced with:

$$\tau_{(u,c)}(x) \equiv (I(x+1) = u) \wedge (C(h, H(x,1), 1) = c).$$

So regardless of what time step $\mathcal{TN}$ is at, every entry of the output tape is written on to it at time h. As the contents of every cell on tape 1 are eventually defined and afterwards remain fixed, in any model of $\mathcal{TTM}_\mathcal{U} \cup \Psi^u_\mathcal{X}$ a unique output of the form $\Psi^v_\mathcal{Y} = \{C(h, \hat{i}+1, 1) = v_i\}_{i=0}^\infty$, expresses these cell values.

Notably this means that the output time step h does not need to occur at some transfinite time in order for the output to depend on the whole infinite computation. Indeed h can take any value in $\mathbb{N} \setminus \{0\}$. The atemporal nature of this output is an example of how a theory machine is able to compute without conforming to the usual assertion that any computational process must follow a causal temporal order.

To compute an infinite sequence a type-2 computation requires an infinite amount of time. However such a sequence can always be finitely computed to an arbitrary degree, which means that despite the infinite resource usage it is, in a sense, physically computable. An infinite type-2 computable sequence is typically referred to as “computable”, however it is not immediately clear why infinitely calculable sequences should be characterised by type-2 machines. The CT thesis implies that finite calculable problems can be computed by a Turing machine, and we believe that Theorem 2 in Subsect. 4.2 provides good evidence for why this should extend to infinite calculable problems and type-2 machines.

3.3 Blum-Shub-Smale Machines

Blum-Shub-Smale (BSS) machines [4] are a form of algebraic computer that act on rings. Usually on the reals or the integers with the usual ordering relation “$<$”. Unlike type-2 machines, a BSS machine acting on the reals is typically viewed as an idealised model of computation rather than a physically implementable one.

A BSS machine has a finite set of rules, each with an associated internal state. These rules are either computation rules or branch rules. Computation rules take the form (s_k, F_k, s_{i_k}) where F_k is a polynomial function. When applied to the ring element x the rule is read as "if the internal state is s_k then apply F_k to x before going to state s_{i_k}." Branch rules are of the form $(s_k, d_k, s_{i_k}, s_{j_k})$ where d_k is a ring element. The rule is read as "if the internal state is s_k and if $x < d_k$ go to state s_{i_k}, otherwise go to state s_{j_k}."

Let S be a BSS machine acting on $\mathbb{R}$ via the rules $\mathcal{P} = \{(s_k, F_k, s_{i_k})\}_{k=0}^{L} \cup \{(s_k, d_k, s_{i_k}, s_{j_k})\}_{k=L+1}^{K}$, with initial state s_0 and halting state s_1. For simplicity let the inputs and outputs of S lie in the interval $[0, 1]$. We can then describe S as the $SO^{=}_{\mathcal{V}_{BSS}}$-theory machine $\mathcal{S} = (\mathcal{BSS}_{\mathcal{P}}, \{0\hat{,}1\}^{\omega}_{\mathcal{Z}_0}, \{0\hat{,}1\}^{\omega}_{\mathcal{Z}_h})$, with vocabulary $\mathcal{V}_{BSS} = \mathcal{V}_{PA} \cup \{N, B, T, V, I, h, \frac{1}{2}\} \cup \{s_1, \ldots, s_K\}$.

Here $\mathcal{V}_{PA} \cup \{h\}$ are as in Subsect. 3.1, however instead of describing Peano arithmetic, in the models of $\mathcal{S}$ the elements of $\mathcal{V}_{PA}$ describe real arithmetic. To describe the discrete time evolution of S we have the unary relation N which defines $\mathbb{N} \subset \mathbb{R}$. For each $x \in \mathbb{N}$ the function $I(x)$ gives the internal state at time x, and $V(x)$ gives the real number value of the machine at time x. The unary functions B and T are used in defining the inputs, with $B(T^n(y))$ giving the nth binary digit of $y \in [0, 1]$. Specifically $B(y) = \lfloor 2y \rfloor$ and $T(y) = 2y - \lfloor 2y \rfloor$. Finally the states of S are denoted by the constants $\{s_1, \ldots, s_K\}$.

The inputs and outputs of $\mathcal{S}$ are respectively given by the simple sequences $\mathcal{Z}_0 = \{B(T^i(V(0)))\}_{i\in\mathbb{N}}$ and $\mathcal{Z}_h = \{B(T^i(V(h)))\}_{i\in\mathbb{N}}$. An input of the form $\bigcup_{i=0}^{\infty}\{B(T^i(V(0))) = b_i\}$ states that $V(0) = 0.b_0 b_1 \ldots$, where $0.111\ldots = 1$.

The theory of $\mathcal{S}$ is $\mathcal{BSS}_{\mathcal{P}} = RA \cup IOB \cup NT \cup EVB_{\mathcal{P}} \cup HT_{s_1}$ where RA is the set of axioms of real arithmetic [1,23], consisting of the first-order axioms of a dense ordered field together with the second-order least-upper bound axiom. The set IOB defines the functions used to describe the machine's input and output. The set of sentences NT defines the relation $N(x)$ to be true iff $x \in \mathbb{N}$. Which allows $EVB_{\mathcal{P}}$ to describe the implementation of the rules of $\mathcal{P}$ at natural time steps. Finally HT_{s_1} is as in Subsect. 3.1.

As $RA \subset \mathcal{BSS}_{\mathcal{P}}$ and $\mathcal{V}_{PA} \subset \mathcal{V}_{BSS}$ every model of $\mathcal{BSS}_{\mathcal{P}}$ must be an expansion of $\langle \mathbb{R}; <, \leqslant, +, \times, 0, 1\rangle$, the usual structure of the the real numbers [23]. At time 0 the internal state of the machine should be s_0 and the input should lie in $[0, 1]$, to ensure this and also define B and T we have:

$$IOB = \left\{\begin{array}{ll} I(0) = s_0, & \forall x((B(x) = 0) \leftrightarrow (x < \frac{1}{2})), \\ 0 \leqslant V(0), & \forall x((B(x) = 1) \leftrightarrow (\frac{1}{2} \leqslant x)), \\ V(0) \leqslant 1, & \forall x((x < \frac{1}{2}) \rightarrow (T(x) = (2 \times x))), \\ (\frac{1}{2} + \frac{1}{2}) = 1, & \forall x((\frac{1}{2} \leqslant x) \rightarrow (T(x) = ((2 \times x) - 1))) \end{array}\right\}.$$

To define the natural numbers within $\mathbb{R}$ we have:

$$NT = \left\{\begin{array}{l} N(0) \wedge N(h), \\ \forall x((\neg(x = 0) \wedge (x < 1)) \rightarrow \neg N(x)), \\ \forall x((0 \leqslant x) \rightarrow (N(x) \leftrightarrow N(x + 1))) \end{array}\right\}.$$

Note that NT also ensures that $h \in \mathbb{N}$. To apply the rules we have:

$$EVB_{\mathcal{P}} = \{\forall x(N(x) \rightarrow \phi_k(x))\}_{k=0}^{L} \cup \{\forall x(N(x) \rightarrow \beta_k(x))\}_{k=L+1}^{K}.$$

Where the kth computation and branch rules are respectively implemented by:

$$\phi_k(x) \equiv (I(x) = s_k) \rightarrow ((V(x+1) = F_k(V(x))) \wedge (I(x+1) = s_{i_k})), \text{ and}$$
$$\beta_k(x) \equiv (I(x) = s_k) \rightarrow (((V(x) < d_k) \wedge (I(x+1) = s_{i_k})) \vee (I(x+1) = s_{j_k})).$$

As $F_k(y)$ is a polynomial function, it can be defined within $\phi_k(x)$ in terms of $+, \times, 0, \frac{1}{2}, 1$, possibly with the aide of some additional constants that can be added to the vocabulary. Each d_k can be similarly defined within $\beta_k(x)$, or it can be defined as an additional constant in a similar manner to $V(0)$.

Given all of this, in any model $\mathfrak{C}$ of $\mathcal{BSS}_{\mathcal{P}} \cup \Psi^u_{\mathcal{Z}_0}$ the computation at time 0 begins as it should do at $0.u_0u_1\ldots$ and in state s_0, before evolving according to the rules of $\mathcal{P}$ until it reaches state s_1, at time h. Consequently the output set $\Psi^v_{\mathcal{Z}_h} = \bigcup_{i=0}^{\infty}\{B(T^i(V(h))) = v_i\}$ is true in $\mathfrak{C}$ iff $0.v_0v_1\ldots$, is the output of S on input $0.u_0u_1\ldots$. Hence $\mathcal{S}$ simulates the BSS machine S.

We can describe other types of BSS machines as theory machines by replacing RA with the axioms of another ring, and adapting the input and output sets. Some BSS machines are capable of deciding problems that are not computable by either a Turing machine or a type-2 machine. For example a BSS machine may be equipped with a binary encoding of the halting problem, which it can then use to decide whether a given input halts. Notably though this requires a theory machine with an infinitely large theory. For a real function to be computable by a type-2 machine in a given encoding it must be continuous in that encoding [24]. In contrast, BSS machines ignore the encoding, and are able to implement discontinuous functions via the branch rules. Both of these capabilities are achievable using machines described by a finite second-order theory.

3.4 Other Examples of Computation

Quantum Computers

Perhaps the most famous example of physical computation is quantum computation [16]. Quantum computers utilise quantum mechanics to efficiently perform calculations that do not appear to be implementable by Turing machines in polynomial time. A well-studied model of quantum computation is the quantum circuit [16].

We can describe a quantum circuit as a second-order theory machine by giving it the axioms of the complex numbers [1], which like those of real arithmetic are all first-order sentences apart from the second-order least upper-bound axiom. To implement a circuit with n qubits we can use a binary function V with range $\mathbb{C}$, and such that the value of $V(x, y)$ is equal to the value of the state $y \in \{|0\rangle, \ldots, |2^n - 1\rangle\}$ at time $x \in \mathbb{N}$. To apply a quantum gate to the kth qubit at time x we can make $V(x+1, y) = (\alpha \times V(x, y)) + (\beta \times V(x, y'))$ for some $\alpha, \beta \in \mathbb{C}$ and each y. Where y' is the state whose binary expansion is the same as y in every digit except for the kth digit.

As any computable sequence of quantum circuits can be arbitrarily approximated by a finite set of gates [16] we can use a Turing machine $\mathcal{N}$ as in Subsect. 3.1 to generate the circuit before implementing it on V.

Fluid-Based Computers

As noted in the introduction, theory machines are capable of computing in an atemporal manner. This capability is exemplified by fluid-based computation, as we require simultaneous differential equations (e.g. the Navier-Stokes equations [18]) to understand how fluids evolve, we cannot in general translate these into a finite set of rules and apply them at discrete sequential time steps.

Suppose we have a Newtonian fluid mechanical system contained within some fixed rigid structure. To input we can adjust the initial state of the fluid at time t_0, whereas the output can be the state of the fluid at time t_1.

We can describe this system as a theory machine by describing the pressure at each time and space coordinate by a quaternary function $P(x, y, z, t)$, and the velocity vector of the fluid by three quaternary functions F_1, F_2, F_3. To ensure that these are functions from $\mathbb{R}^4$ to $\mathbb{R}$ we can include the axioms of real arithmetic RA [1,23] in the machine's theory.[3] It is then possible to define every partial derivative of each of these functions. For example, the partial derivative of P in the 1st dimension is the quaternary function $\partial_1 P$ which satisfies:

$$\begin{aligned}&\forall x \forall y \forall z \forall t \forall \epsilon \exists \delta (((0 < \epsilon) \wedge (0 < \delta)) \rightarrow \\ &\qquad (((|((P(x+\delta, y, z, t) - P(x, y, z, t)) - (\partial_1 P(x, y, z, t) \times \delta))| < \epsilon)).\end{aligned}$$

The Navier-Stokes equations can then be implemented by writing them as sentences of the theory.

The inputs and outputs of the system can each be described by a set of sentences of the form $F_i(a, b, c, 0) = d$, where $a, b, c, d \in \mathbb{Q}$. As the fluid flow is continuous only a countable number of these are needed. The rigid boundaries of the system can be similarly defined in terms of rational approximations.

Infinite Time Turing Machines

Infinite time Turing machines (ITT machines) [12] generalise the concept of a Turing machine by allowing a computation to take an ordinal number of time steps. At successor time steps they behave like a normal Turing machine, whereas at limit ordinal time steps the contents of each cell becomes the limit supremum of the previous contents (the tape alphabet of an ITT machine is $\{0, 1\}$). At limit times the head of the machine is placed back to where it started, and the internal state goes to a fixed limit state.

Typically ITT machines are not viewed as being physically implementable, however we can describe any ITT machine by a second-order theory machine. To do this we take the theory machine corresponding to a multi-tape Turing machine in Subsect. 3.1 with alphabet $\{0, 1, \mathbf{L}\}$ and replace the induction axiom of PA with the second-order well-foundedness axiom, which is satisfied by all ordinal structures [20]. With this any model of the machine will be an expansion of an ordinal structure, and what occurs at limit ordinal stages can then be

[3] Unlike BSS machines, if such a device requires only finite precision to be implemented correctly then the second order least upper bound axiom in RA is not required.

simply stated. Indeed, to define the tape contents at any limit step x we can invoke the limit supremum via $\forall y \forall z (C(x, y, z) = 0) \vee (C(x, y, z) = 1)$ and:

$$\forall y \forall z ((C(x, y, z) = 0) \leftrightarrow \exists p((p < x) \wedge \forall q(((p < q) \wedge (q < x)) \rightarrow (C(q, y, z) = 0)))).$$

4 Finite First-Order Theory Machines

As is clear from the above examples, theory machines have the potential for super-Turing computation. Whilst this power could be explained by noting that super-Turing computations always require an infinite space, the same is also true for quantum computers, whose computations can be simulated by a Turing machine [16]. Additionally quantum computers are generally believed to be physically realisable and are hence viewed as an example of physical computation, despite their infinite space usage.

So why are some infinite computation systems "physical" when others are "unphysical"? Well we argue that a device's computational power can be characterised by the class of the logical systems required to finitely describe the device. We also assert that a device can be physically computed with only if its computational aspects can be described by a finite first-order theory machine.

Definition 4.1. *A* ***finite first-order theory (FFOT) machine*** *is a $FO_{\mathcal{V}}$-theory machine $\mathcal{M} = (\mathcal{T}, \mathcal{I}, \mathcal{O})$ such that $\mathcal{T}$ is a finite set of sentences.*

Example 4.1. Let N be a multi-tape Turing machine as in Subsect. 3.1. We can describe N via the FFOT machine $\mathcal{N}^- = (\mathcal{TM}_{\mathcal{R}}, \hat{\Sigma}^*_{\mathcal{X}}, \hat{\Gamma}^*_{\mathcal{Y}})$, in the same vocabulary $\mathcal{V}_{TM}$ as $\mathcal{N}$ and with the same input and output sets, the only difference is that the theory is $\mathcal{TM}^-_{\mathcal{R}} = EQ_{\mathcal{V}_{TM}} \cup PA^- \cup IT_{\mathbf{B}} \cup ET_{\mathcal{R}} \cup HT_{s_1}$.

Where $EQ_{\mathcal{V}_{TM}}$ consists of the axioms of equality for the vocabulary $\mathcal{V}_{TM}$ as in Remark 2.1, and PA^- is the set of first-order Peano arithmetic axioms [15], which excludes the second-order induction axiom. Now while there do exist models of PA^- which are not isomorphic to the usual structure of the natural numbers, every model of PA^- has an initial segment which is isomorphic to the natural numbers [15], which is referred to as the standard part of the model. Hence each model $\mathfrak{D}$ of $\mathcal{TM}^-_{\mathcal{R}}$ contains a standard part.

By our reasoning in Subsect. 3.1 we know that in the standard part of $\mathfrak{D}$ the computation progresses as it should do. If it halts then it does so at the standard part time step h, from which the output is defined finitely until a blank symbol is found on the tape. Crucially this output cannot depend on what happens at non-standard time steps, nor can h be reached early by the contents of the non-standard tape cells, as the function H will never map to them at standard time steps. Therefore the same output is still true in any model of $\mathcal{TM}^-_{\mathcal{R}} \cup \Phi^w_{\mathcal{X}}$.

The output of N is undefined if N does not halt on input w. In which case h cannot lie in the standard part of any model of $\mathcal{TM}^-_{\mathcal{R}} \cup \Phi^w_{\mathcal{X}}$. At such a time the configuration of the machine is independent of what occurs during the standard part of the computation, which means that it could be anything, and is hence not unique. Therefore $\mathcal{N}^-(\Phi^w_{\mathcal{X}})$ must also be undefined.

The standard way of formulating a computational problem is to view it as a function from a set of words to a set of words. Thus in order to compare the computational power of FFOT machines with other forms of computation we require a reasonable way of stating that a word function can be computed by a theory machine.

Definition 4.2. *Let Σ and Γ be sets of symbols, and let $f :\subseteq \Sigma^a \to \Gamma^b$ for $a, b \in \{*, \omega\}$ be a word function problem.*

We say that an $\mathfrak{LS}_{\mathcal{V}}$-theory machine $\mathcal{M} = (\mathcal{T}, \mathcal{I}, \mathcal{O})$ ***is able to compute*** *f if there exist sets of distinct constants $\Sigma, \Gamma \subseteq \mathcal{V}$ and simple sequences $\mathcal{X}, \mathcal{Y}$ such that $\hat{\Sigma}^a_{\mathcal{X}} \subseteq \mathcal{I}$, $\hat{\Gamma}^b_{\mathcal{Y}} \subseteq \mathcal{O}$ and for every $u \in dom(f)$ we have $\mathcal{M}(\Upsilon^u_{\mathcal{X},a}) = \Upsilon^{f(u)}_{\mathcal{Y},b}$, where $\Upsilon^v_{\mathcal{Z},c} = \Phi^v_{\mathcal{Z}}$ if $c = *$ and $\Upsilon^v_{\mathcal{Z},c} = \Psi^v_{\mathcal{Z}}$ if $c = \omega$.*

So a theory machine is able to compute a word function if there exists a simple way for a user to configure each admissable input word into the machine, such that the function's output can be simply read off from the machine. Note that for a theory machine to be able to compute a function the whole of its domain and co-domain must be contained within the input and the output sets. So we cannot compute a partial function by removing the undefined elements from the input or output sets, they must be undefined by the computation as well.

4.1 FFOT Machines and the Church-Turing Thesis

We now come to our main theorem.

Theorem 1. *A finite word function problem $f :\subseteq \Sigma^* \to \Gamma^*$ is computable by some Turing machine if and only if there exists a finite first-order theory machine that is able to compute f.*

Proof. ($\Rightarrow$) Let $f :\subseteq \Sigma^* \to \Gamma^*$ be computable, and let N be a deterministic Turing machine that computes f. We are then able to compute f with the FFOT machine $\mathcal{N}^-$ in Example 4.1.

($\Leftarrow$) Conversely, suppose that $f :\subseteq \Sigma^* \to \Gamma^*$ is computable by the FFOT machine $\mathcal{M} = (\mathcal{T}, \mathcal{I}, \mathcal{O})$ with vocabulary $\mathcal{V}$. By definition, for any $w \in \mathrm{dom}(f)$ we have $\mathcal{T} \cup \Phi^w_{\mathcal{X}} \models_{FO_{\mathcal{V}}} \Phi^{f(w)}_{\mathcal{Y}}$ and this is true for no other element of $\hat{\Gamma}^*_{\mathcal{Y}}$. Hence by Gödel's completeness theorem [13] we have $\mathcal{T} \cup \Phi^w_{\mathcal{X}} \vdash_{FO_{\mathcal{V}}} \Phi^{f(w)}_{\mathcal{Y}}$, and there exists a finite formal proof of the truth of $\Phi^{f(w)}_{\mathcal{Y}}$ given $\mathcal{T} \cup \Phi^w_{\mathcal{X}}$.

As $\mathcal{T} \cup \Phi^w_{\mathcal{X}}$ is a finite set of first-order sentences the set of all first-order sentences provable from it is computably enumerable. Similarly as $\mathcal{Y}$ is a simple sequence, the sentences of $\hat{\Gamma}^*_{\mathcal{Y}}$ are computably enumerable. We can therefore construct a Turing machine M that, on an input of w, enumerates all sentences provable from $\mathcal{T} \cup \Phi^w_{\mathcal{X}}$, while concurrently enumerating $\hat{\Gamma}^*_{\mathcal{Y}}$. By our above reasoning we know that $\Phi^{f(w)}_{\mathcal{Y}}$ and only $\Phi^{f(w)}_{\mathcal{Y}}$ is in both sets of sentences, so by checking whether each enumerated sentence appears in both sets, M will eventually find $\Phi^{f(w)}_{\mathcal{Y}}$ and know what $f(w)$ is. Therefore M is able to compute f by searching for $\Phi^{f(w)}_{\mathcal{Y}}$ and outputting $f(w)$ upon finding it. $\square$

A consequence of this result is that the CT thesis can be reformulated as:

Every effectively calculable function is computable by a finite first-order theory machine (1)

Therefore, if the CT thesis is true and applies to physical calculations, then it must be the case that the computational aspects of any obtainable physical system is describable by a finite first-order theory machine.

We do not in fact need to limit ourselves to first-order logic, as the proof of Theorem 1 relies on two facts; the fact that we can describe any Turing machine by a finite first-order theory machine, and the fact that first-order logic is complete. Neither of these properties are held solely by first-order logic, we therefore have the following generalisation.

Corollary 1. *Let $\mathfrak{LS}$ be a complete logical system, let $\mathcal{V}$ be finite, and let $\mathcal{C}$ be a class of $\mathfrak{LS}_{\mathcal{V}}$-theory machines with finite theories. If we are able to compute any Turing machine computable problem by a theory machine in $\mathcal{C}$ then $\mathcal{C}$ is computationally equivalent to the class of Turing machines.*

4.2 FFOT Machines and Type-2 Machines

Type-2 machines can also be simulated by FFOT machines.

Example 4.2. Let TN be the type-2 machine from Subsect. 3.2, we can describe TN via the FFOT machine $\mathcal{TN}^- = (\mathcal{TTM}^-_{\mathcal{U}}, \hat{\Sigma}^\omega_{\mathcal{X}}, \hat{\Gamma}^\omega_{\mathcal{Y}})$, in the same vocabulary $\mathcal{V}_{TM}$ as $\mathcal{TN}$ and with the same input and output sets, the only difference is that the theory is $\mathcal{TTM}^-_{\mathcal{U}} = EQ_{\mathcal{V}_{TM}} \cup PA^- \cup IT_{\mathbf{B}} \cup TET_{\mathcal{U}}$.

As in Example 4.1 every model $\mathfrak{C}$ of $\mathcal{TTM}^-_{\mathcal{U}}$ has a standard part which is isomorphic to the natural numbers. From Subsect. 3.2 we know that the output of $\mathfrak{C}$ is entirely defined by what occurs in the standard part of $\mathfrak{C}$. Thus the output cannot be influenced by what happens at the non-standard time steps of $\mathfrak{C}$, as if it was this would lead to a contradiction. Therefore if $g :\subseteq \Sigma^\omega \to \Gamma^\omega$ is computed by TN then $\mathcal{TN}^-$ is able to compute g. As if $g(u)$ is undefined then TN on input u must only ever write on finitely many cells of the output tape. In which case the rest of the values of $C(h, y, 1)$ could be anything and $\mathcal{TN}^-(\Psi^u_{\mathcal{X}})$ is undefined.

Theorem 2. *An infinite word function problem $g :\subseteq \Sigma^\omega \to \Gamma^\omega$ is computable by a type-2 machine if and only if there exists a finite first-order theory machine that is able to compute g.*

Proof. ($\Rightarrow$) This follows from Example 4.2.

($\Leftarrow$) Suppose that $g :\subseteq \Sigma^\omega \to \Gamma^\omega$ is computable by some FFOT machine $\mathcal{M} = (\mathcal{T}, \mathcal{I}, \mathcal{O})$ with vocabulary $\mathcal{V}$. As in the proof of Theorem 1 it must be the case that for any $u \in \mathrm{dom}(g)$ we have $\mathcal{T} \cup \Psi^u_{\mathcal{X}} \models_{FO_{\mathcal{V}}} \Psi^{g(u)}_{\mathcal{Y}}$. By the compactness theorem [9], for any finite subset $\Omega \subset \Psi^{g(u)}_{\mathcal{Y}}$, there exists a finite subset $\Delta \subset \Psi^u_{\mathcal{X}}$, such that $\mathcal{T} \cup \Delta \models_{FO_{\mathcal{V}}} \Omega$. As otherwise by compactness there would exist a model of $\mathcal{T} \cup \Psi_{\mathcal{X}}^{u}$ in which $\Psi_{\mathcal{Y}}^{g(u)}$ is false. Hence by Gödel's completeness theorem [13] there must exist a finite formal proof of the truth of Ω given $\mathcal{T} \cup \Delta$.

Hence we can construct a type-2 machine TM, that on input u enumerates the elements of $\Psi_{\mathcal{X}}^{u}$, and from which it enumerates all sentences provable from $\mathcal{T} \cup \Psi_{\mathcal{X}}^{u}$. Hence as in the proof of Theorem 1 TM is able to obtain the elements of $\Psi_{\mathcal{Y}}^{g(u)}$ and sequentially output $g(u)$. □

In the original CT thesis an effectively calculable function is intended to be finite. However if we were to assume that it could also be an infinite function, then Theorem 2 and our reformulation of the CT thesis (1) collectively imply that every effectively calculable infinite function is computable by a type-2 machine.

5 Conclusion and Future Work

As we did with a Turing machine we can convert the quantum computer in Subsect. 3.4 into a FFOT machine by simply removing the least upper-bound axiom from its theory. Any quantum computation can be finitely approximated, which means that the completion of the complex numbers is not required for such a theory machine to obtain the correct output.

Indeed it is this ability to approximate which separates the more physically reasonable quantum computers and type-2 machines from the less physically reasonable BSS and ITT machines, as we cannot obtain the outputs of the latter pair through finite approximation. We are therefore unable to formulate them in in any logic in which the compactness theorem holds [9], as $\Phi \models_{FO_{\mathcal{V}}} \Psi$ cannot be true if $\Delta \models_{FO_{\mathcal{V}}} \Psi$ is false for every finite $\Delta \subseteq \Phi$. Hence whilst the former "physical" computers can be described using first-order logic, the latter "unphysical" computers cannot.

In future work we hope to develop a concept of atemporal complexity for theory machines, ideally in a manner which is consistent with both Turing machine and quantum complexity. We also intend to look into finite second-order systems whose computational capabilities we suspect may be similar to that of ordinal Turing machines [17]. Beyond this, we suspect that a relationship can be found between a general class of theory machines and abstract transition systems [5].

In general we believe that the computational capabilities of arbitrary systems (both physical and unphysical) can be characterised and separated by the logical systems capable of describing them. For example it can be shown that a BSS machine acting on $\mathbb{R}$ can be described by a "real" first-order logical system whose structures are necessarily expansions of $\langle \mathbb{R}; <, \leqslant, +, \times, 0, 1 \rangle$. However, such a logical system is unable to describe ITT machines, which is consistent with the fact that there are ITT-computable problems which are not BSS-computable. We therefore believe that by studying models of computation via theory machines we should be able to gain a clearer understanding of how and why distinct systems differ in their ability to compute and their ability to efficiently compute.

References

1. Apelian, C., Surace, S.: Real and Complex Analysis. CRC Press, Boca Raton (2009)
2. Baumeler, Ä., Wolf, S.: Computational tameness of classical non-causal models. Proc. R. Soc. A **474**(2209) (2018). https://doi.org/10.1098/rspa.2017.0698
3. Blackburn, P., De Rijke, M., Venema, Y.: Modal Logic. Graph. Darst, vol. 53. Cambridge University Press, Cambridge (2002)
4. Blum, L., Shub, M., Smale, S.: On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. Bull. Am. Math. Soc. **21**, 1–46 (1989). https://doi.org/10.1090/S0273-0979-1989-15750-9
5. Bournez, O., Dershowitz, N., Néron, P.: Axiomatizing analog algorithms. In: Beckmann, A., Bienvenu, L., Jonoska, N. (eds.) CiE 2016. LNCS, vol. 9709, pp. 215–224. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40189-8_22
6. Barry Cooper, S.: Computability Theory. CRC Press, Boca Raton (2003)
7. Deutsch, D.: Quantum theory, the Church-Turing principle and the universal quantum computer. Proc. R. Soc. A **400**(1818), 97–117 (1985). https://doi.org/10.1098/rspa.1985.0070
8. Trillas, E., Eciolaza, L.: Fuzzy Logic. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14203-6
9. Epstein, R.L.: Classical Mathematical Logic (2011)
10. Grigorieff, S., Valarcher, P.: Evolving MultiAlgebras unify all usual sequential computation models. Leibniz Int. Proc. Inform. (2010). https://doi.org/10.4230/LIPIcs.STACS.2010.2473
11. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. 77–111 (2000). https://doi.org/10.1145/343369.343384
12. Hamkins, J.D., Lewis, A.: Infinite time Turing machines. J. Symb. Logic **65**, 567–604 (2000). https://doi.org/10.2307/2586556
13. Henkin, L.: The completeness of the first-order functional calculus. J. Symb. Logic **14**(3) (1949). https://doi.org/10.2307/2267044
14. Horsman, C., Stepney, S., Wagner, R.C., Kendon, V.: When does a physical system compute? Proc. R. Soc. A **470**(2169) (2014). https://doi.org/10.1098/rspa.2014.0182
15. Kaye, R.: Models of Peano arithmetic (1991)
16. Kitaev, A.Y., Shen, A., Vyalyi, M.: Classical and Quantum Computation. American Mathematical Society, Boston (2002)
17. Koepke, P., Koerwien, M.: Ordinal computations. Math. Struct. Comput. Sci. **16**, 867–884 (2006). https://doi.org/10.1017/S0960129506005615
18. Landau, L.D., Lifshitz, E.M.: Fluid Mechanics. Course of Theoretical Physics. Pergamon, Oxford (1987)
19. Newell, A., Simon, H.: The logic theory machine-a complex information processing system. IRE Trans. Info. Theor. **2**, 61–79 (1956)
20. Sierpiński, W.: Cardinal and ordinal numbers. Państwowe Wydawn. Naukowe, vol. 34 (1958)
21. Smullyan, R.M.: Theory of Formal Systems. Princeton University Press, Princeton (1961)
22. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **2**, 230–265 (1937)
23. Vakil, N.: Real Analysis Through Modern Infinitesimals. Cambridge University Press, Cambridge (2011)
24. Weihrauch, K.: Computable Analysis. Springer, Berlin (2000)

Author Index

Alhazov, Artiom 51

Bednárová, Zuzana 105
Bonchiş, Cosmin 69

Cao, Jiannong 34

Fernau, Henning 88
Freund, Rudolf 1, 51

Geffert, Viliam 105

Istrate, Gabriel 69
Ivanov, Sergiu 51

Kuppusamy, Lakshmanan 88

Nagy, Benedek 137

Oladele, Rufus O. 88

Perrot, Kévin 121
Perrotin, Pacôme 121

Raman, Indhumathi 88
Raynal, Michel 34
Rochian, Vlad 69

Sené, Sylvain 121

Vályi, Sándor 137

Whyman, Richard 153