EPTCS 409

Proceedings of the

Fifteenth International Symposium on
Games, Automata, Logics, and Formal Verification

Reykjavik, Iceland, 19-21 June 2024

Edited by: Antonis Achilleos and Adrian Francalanza

Published: 30th October 2024
DOI: 10.4204/EPTCS.409
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface iii

Invited Presentation: Hyperproperties: the Exciting World Beyond k-Hypersafety 1
Bernd Finkbeiner

Invited Presentation: Shielded Reinforcement Learning for Safe and Optimal Cyber Physical Systems 2
Kim Guldstrand Larsen

Invited Presentation: Mechanizing Session-Types: Challenges and Lessons Learned 3
Brigitte Pientka

Invited Presentation: Bug Detection at Scale 4
Azalea Raad

Synthesis of Timeline-Based Planning Strategies Avoiding Determinization 5
Renato Acampora, Dario Della Monica, Luca Geatti, Nicola Gigante, Angelo Montanari and Pietro Sala

Jumping Automata Must Pay 19
Shaull Almagor and Ishai Salgado

Reactive Synthesis for Expected Impacts 35
Emanuele Chini, Pietro Sala, Andrea Simonetti and Omid Zare

Towards the Usage of Window Counting Constraints in the Synthesis of Reactive Systems to Reduce State Space Explosion 53
Linda Feeken and Martin Fränzle

Deterministic Suffix-reading Automata 70
R Keerthan, B Srivathsan, R Venkatesh and Sagar Verma

Adding Reconfiguration to Zielonka's Asynchronous Automata 88
Mathieu Lehaut and Nir Piterman

A Game-Theoretic Approach for Security Control Selection 103
Dylan Léveillé and Jason Jaskolka

Epistemic Skills: Logical Dynamics of Knowing and Forgetting 120
Xiaolong Liang and Yì N. Wáng

An Evaluation of Massively Parallel Algorithms for DFA Minimization 138
Jan Martens and Anton Wijs

Reachability and Safety Games under TSO Semantics 154
Stephan Spengler

Preface

Antonis Achilleos   Adrian Francalanza

This volume contains the proceedings of GandALF 2024, the Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification. The symposium was held in Reykjavik, Iceland, on June 19–21, 2024.

The GandALF symposium was established to provide an opportunity for researchers interested in logic for computer science, automata theory, and game theory, to gather and discuss the application of formal methods to the specification, design, and verification of complex systems. Previous editions of GandALF were held in Udine, Italy (2024); Madrid, Spain (2022); Padova, Italy (2021); Brussels, Belgium (2020); Bordeaux, France (2019); Saarbrücken, Germany (2018); Rome, Italy (2017); Catania, Italy (2016); Genoa, Italy (2015); Verona, Italy (2014); Borca di Cadore, Italy (2013); Napoli, Italy (2012); and Minori, Italy (2011 and 2010). The symposium provides an international forum where people from different areas, backgrounds, and countries, can fruitfully interact, as witnessed by the composition of the program and steering committees and by the country distribution of the submitted papers.

The program committee selected 10 papers (out of 20 submissions) for presentation at the symposium. Each paper was reviewed by at least three referees, and the selection was based on originality, quality, and relevance to the topics of the call for papers. The scientific program included presentations on automata, logics for computer science and verification, learning for verification, formal methods and specification languages, games for security and verification, session types, and synthesis. The program included four invited talks, given by Bernd Finkbeiner (CISPA Helmholtz Center for Information Security), Kim G. Larsen (Aalborg University), Brigitte Pientka (McGill University), and Azalea Raad (Imperial College London). We are deeply grateful to them for contributing to this year's edition of GandALF.

We would like to thank the authors who submitted papers, the speakers, the program committee members, and the additional reviewers for their excellent work. We also thank EPTCS and arXiv for hosting the proceedings; in particular, we thank Rob van Glabbeek for the precise and prompt technical support with issues related to the proceeding publication procedure.

Finally, we would like to thank the local organisers: Angeliki Chalki, Jana Wagemaker, Vasiliki Kyriakou, and Jasmine Xuereb for ensuring the event ran smoothly.

Antonis Achilleos and Adrian Francalanza

Program Chairs

- Antonis Achilleos, Reykjavik University (Iceland)
- Andrian Francalanza, University of Malta (Malta)

Program Committee

- Parosh Aziz Abdulla, Uppsala University (Sweden)
- Valentina Castiglioni, Eindhoven University of Technology (Netherlands)

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. iii–iv, doi:10.4204/EPTCS.409.0

- Aggeliki Chalki, Reykjavik University (Iceland)
- Laure Daviaud, University of East Anglia (UK)
- Dario Della Monica, Università degli Studi di Udine (Italy)
- Giorgio Delzanno, Università degli Studi di Genova (Italy)
- Léo Exibard, Université Gustave Eiffel (France)
- Nicola Gigante, Free University of Bozen-Bolzano (Italy)
- Julian Gutierrez, Monash University (Australia)
- Jonas Kastberg Hinrichsen, Aarhus University (Denmark)
- Ryan Kavanagh, Université du Québec à Montréal (Canada)
- Orna Kupferman, Hebrew University (Israel)
- Martin Leucker, University of Luebeck (Germany)
- Jakub Michaliszyn, University of Wroclaw (Poland)
- Laura Nenzi, University of Trieste (Italy)
- Paweł Parys, University of Warsaw (Poland)
- Guillermo Perez, University of Antwerp (Belgium)
- Jakob Piribauer, TU Dresden (Germany)
- Ocan Sankur, Université Rennes, CNRS/Irisa (France)
- Felix Stutz, University of Luxembourg (Luxembourg)
- Patrick Totzke, University of Liverpool (UK)
- Tomoyuki Yamakami, University of Fukui (Japan)
- Matteo Zavatteri, University of Padova (Italy)
- Martin Zimmermann, Aalborg University (Denmark)

Steering Committee

- Luca Aceto, Reykjavik University (Iceland)
- Javier Esparza, University of Munich (Germany)
- Salvatore La Torre, University of Salerno (Italy)
- Angelo Montanari, University of Udine (Italy)
- Mimmo Parente, University of Salerno (Italy)
- Jean-François Raskin, Université libre de Bruxelles (Belgium)
- Martin Zimmermann, Aalborg University (Denmark)

External Reviewers

Luca Geatti, Karam Kharraz, Paolo Marrone, Piotr Polesiuk, Nicola Prezza, Christian Schilling, Daniel Thoma, Alexander Weinert.

Hyperproperties: the Exciting World Beyond k-Hypersafety

Bernd Finkbeiner
CISPA Helmholtz Center for Information Security
finkbeiner@cispa.de

System requirements related to concepts like information flow, knowledge, and robustness cannot be judged in terms of individual system executions, but rather require an analysis of the relationship between multiple executions. Such requirements belong to the class of hyperproperties, which generalize classic trace properties to properties of sets of traces.

A key idea in the verification of hyperproperties has been to analyze self-compositions of programs. Hyperproperties that need to hold for all possible combinations of k traces, such as k-hypersafety properties, can be analyzed as standard trace properties of the k-fold self-composition. The implicit universal quantification over the traces is, however, an inherent limitation of this paradigm, which makes it difficult to abstract in an existential manner from phenomena like scheduling in concurrent programs, nondeterministic choice, or speed of execution in asynchronous computations. Alternations between quantifiers are furthermore essential for counterfactual reasoning about causation and blame.

In this talk, we will explore the exciting world of hyperproperties beyond k-hypersafety. I will discuss the decidability and complexity of reasoning about such hyperproperties and present algorithmic techniques for the effective resolution of quantifier alternations. I will also give an overview of recently introduced logics for the specification of hyperproperties beyond k-hypersafety, including logics that combine hyperproperties with reasoning over strategies, and logics for second-order hyperproperties.

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 1–1, doi:10.4204/EPTCS.409.1

Shielded Reinforcement Learning for Safe and Optimal Cyber Physical Systems

Kim Guldstrand Larsen
Aalborg University
kgl@cs.aau.dk

I will present recent advances and applications of the tool UPPAAL Stratego (`www.uppaal.org`) supporting automatic synthesis of guaranteed safe and near-optimal control strategies for cyber physical systems. UPPAAL Stratego support reinforcement learning methods to construct near-optimal controllers. However, their behavior is not guaranteed to be safe, even when it is encouraged by reward engineering. One way of imposing safety to a learned controller is to use a safety shield, synthesized using symbolic methods from checking, and hence correct by design. To make synthesis of shields for hybrid environments tractable UPPAAL Stratego are using various abstraction techniques for hybrids systems.

We study the impact of the synthesized shield when applied as either a pre-shield (applied before learning a controller) or a post-shield (only applied after learning a controller). In addition trade-offs between efficiency of strategy representation and degree of optimality subject to safety constraints will be discussed, as well as successful on-going applications (water-management, heating systems, and traffic control).

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 2–2, doi:10.4204/EPTCS.409.2

Mechanizing Session-Types: Challenges and Lessons Learned

Brigitte Pientka
McGill University
bpientka@cs.mcgill.ca

Process calculi provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent processes. Session types allow us to statically verify that processes communicate according to prescribed protocols. Hence, they rule out a wide class of communication-related bugs before executing a given process. They also statically guarantee safety properties such as session fidelity and deadlock freedom, analogous to preservation and progress in the simply typed lambda-calculus.

Although there have been many efforts to mechanize process calculi such as the pi-calculi in proof assistants, mechanizing these systems remains an art. Process calculi use channel or action names to specify process interactions, and they often feature rich binding structures and semantics such as channel mobility. Both of these features can be challenging to mechanize, for we must track names to avoid conflicts, ensure that α-equivalence and renaming are well-defined, etc. Moreover, session types employ a linear type system, where variables cannot be implicitly copied or dropped, and therefore, many mechanizations of these systems require modeling the context and carefully ensuring that its variables are handled linearly.

In this talk, I give an introduction to the challenges that arise when mechanizing session types, and showcase two different kinds of solutions focusing on a session typed system based on classical linear logic: first, I show how to mechanize the a session tysystem directly using explicit contexts identifying key context operations to manage linear contexts; then I show a technique to localize linearity conditions as additional predicates embedded within type judgments, which allows us to use unrestricted typing contexts instead of linear ones. This latter technique is especially relevant when leveraging (weak) higher-order abstract syntax to defer the intricate channel mobility and bindings that arise in a session typed system.

From this mechanization, we discuss key design decisions and draw some key lessons for mechanizing substructural systems, such as session types. The goal of this talk is to engage the community in discussions on what support in proof environments is needed to support the mechanization of substructural systems.

This is joint work with Chuta Sano, Daniel Zackon, Ryan Kavanagh, and Alberto Momigliano.

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 3–3, doi:10.4204/EPTCS.409.3

Bug Detection at Scale

Azalea Raad
Imperial College London
azalea@imperial.ac.uk

Incorrectness Logic (IL) has recently been advanced as a logical under-approximate theory for proving the presence of bugs—dual to Hoare Logic, which is an over-approximate theory for proving the absence of bugs. To facilitate scalable bug detection, we developed incorrectness separation logic (ISL) by marrying the under-approximate reasoning of IL with the local reasoning of separation logic. This locality leads to techniques that are compositional both in code (concentrating on a program component) and in the resources accessed (spatial locality), without tracking the entire global state or the global program within which a component sits. This enables reasoning to scale to large teams and codebases: reasoning can be done even when a global program is not present. To demonstrate this, we developed Pulse, an automatic program analysis for catching memory safety errors, underpinned by ISL. Using Pulse, deployed at Meta, we found a number of real bugs in large codebases such as OpenSSL.

Inspired by this success, we later studied the power of under-approximation for detecting non-termination bugs. Program termination is a classic non-safety property that cannot in general be witnessed by a finite trace. This makes testing for non-termination challenging, and also makes it a natural target for symbolic proof. Discovering non-termination is an under-approximate problem. We thus developed an under-approximate logic for proving non-termination, resulting in a compositional proof method. We prototyped this in an automated tool, Pulse∞ (an extension of Pulse), which has already discovered a number of non-termination bugs in large open-source libraries.

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 4–4, doi:10.4204/EPTCS.409.4

Synthesis of Timeline-Based Planning Strategies Avoiding Determinization*

Renato Acampora
University of Udine, Italy
renato.acampora@uniud.it

Dario Della Monica
University of Udine, Italy
dario.dellamonica@uniud.it

Luca Geatti
University of Udine, Italy
luca.geatti@uniud.it

Nicola Gigante
Free University of Bozen-Bolzano, Italy
nicola.gigante@unibz.it

Angelo Montanari
University of Udine, Italy
angelo.montanari@uniud.it

Pietro Sala
University of Verona, Italy
pietro.sala@univr.it

Qualitative timeline-based planning models domains as sets of independent, but interacting, components whose behaviors over time, the timelines, are governed by sets of qualitative temporal constraints (ordering relations), called synchronization rules. Its plan-existence problem has been shown to be PSPACE-complete; in particular, PSPACE-membership has been proved via reduction to the nonemptiness problem for nondeterministic finite automata. However, nondeterministic automata cannot be directly used to synthesize planning strategies as a costly determinization step is needed. In this paper, we identify a large fragment of qualitative timeline-based planning whose plan-existence problem can be directly mapped into the nonemptiness problem of deterministic finite automata, which can then be exploited to synthesize strategies. In addition, we identify a maximal subset of Allen's relations that fits into such a deterministic fragment.

1 Introduction

Timeline-based planning is an approach that originally emerged and developed in the context of planning and scheduling of *space* operations [16]. In contrast to common action-based formalisms, such as PDDL [9], timeline-based languages do not make a distinction between actions, states, and goals. Rather, the domain is modeled as a set of independent, but interacting, components whose behavior over time, the timelines, is governed by a set of temporal constraints. It is worth pointing out that timeline-based planning was born with an application-oriented flavor, with various successful stories, and only relatively recently some foundational work about its expressiveness and complexity has been produced. The present paper aims at bringing back theory to practice by identifying expressive enough and computationally well-behaved fragments.

Timeline-based planning has been successfully employed by planning systems developed at NASA [5, 6] and at ESA [10] for both short- to long-term mission planning and on-board autonomy. More recently, timeline-based planning systems such as PLATINUm [18] are being employed in collaborative robotics applications [19]. All these applications share a deep reliance on *temporal reasoning* and the need for a tight integration of planning with *execution*, both features of the timeline-based framework. The latter feature is usually achieved by the use of *flexible timelines*, which represent a set of possible executions of the system that differ in the precise timing of the events, hence handling the intrinsic *temporal uncertainty* of the environment. A formal account of timeline-based planning with uncertainty

*This work is partially supported by the INdAM-GNCS Project *Analisi simbolica e numerica di sistemi ciberfisici* (project n. CUP_E53C22001930001).

A. Achilleos and A. Francalanza (Eds.): Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 5–18, doi:10.4204/EPTCS.409.5

© Acampora et al.
This work is licensed under the Creative Commons Attribution License.

has been provided by [7], and much theoretical research followed, including *complexity* [3, 4, 12] and *expressiveness* [14, 11] analyses, based on such a formalization, which is the one we use here as well.

To extend the reactivity and adaptability of timeline-based systems beyond temporal uncertainty, the framework of *timeline-based games* has been recently proposed. In timeline-based games, the system player tries to build a set of timelines satisfying the constraints independently from the choices of the environment player. This framework allows one to handle general nondeterministic environments in the timeline-based setting. However, this expressive power comes at the cost of increasing the complexity of the problem. While the plan-existence problem for timeline-based planning is EXPTIME-complete [12], deciding the existence of strategies for timeline-based games is 2EXPTIME-complete [13], and a controller synthesis algorithm exists that runs in doubly exponential time [1].

Such a high complexity motivates the search for simpler fragments that can nevertheless be useful in practical scenarios. One of these is the *qualitative* fragment, where temporal constraints only concern the relative order between events and not their distance. The qualitative fragment already proved itself to be easier for the plan-existence problem, being PSPACE-complete [8], and this makes it a natural candidate for the search of a good fragment for the strategy existence problem.

A *deterministic* arena is crucial to synthesize a non-clairvoyant strategy in *reactive synthesis* problems (see, for instance, [17]). However, determinizing the nondeterministic (exponentially sized) automaton built for the qualitative case in [8] would cause an exponential blowup, thus resulting in a procedure of doubly-exponential complexity. In this paper, we show that, by imposing some natural restrictions on the set of temporal constraints of the qualitative fragment, it is possible to lower the complexity of the strategy existence problem to EXPTIME. We show that, on the one hand, these restrictions are *sufficient* to directly synthesize a *deterministic* finite automaton (DFA) of singly-exponential size, thus usable as an arena to play the game in an asymptotically optimal way, and, on the other hand, the resulting fragment is expressive enough to capture a large subset of Allen's relations [2], defined in Section 7.

The rest of the paper is organized as follows. Section 2 recalls some background knowledge on timeline-based planning. Section 3 defines the considered fragment, that directly maps into a DFA of singly exponential size. Section 4 gives a word encoding of timelines, and vice versa. Section 5 builds an automaton to recognize plans, and Section 6 shows how to construct an automaton that accepts solution plans. Section 7 identifies the maximal subset of Allen's relations which is captured by the fragment of Section 3. Finally, Section 8 summarizes the main contributions of the work and discusses possible future developments.

2 Background

In this section, we recall the basic notions of timeline-based planning and of its qualitative variant.

2.1 Timeline-Based Planning

The key element of the framework is the notion of *state variable*. Let $\mathbb{N}^+$ be the set of positive natural numbers.

Definition 1 (State variable). *A* state variable *is a tuple $x = (V_x, T_x, D_x)$, where:*

- V_x *is the* finite domain *of the variable;*
- $T_x : V_x \to 2^{V_x}$ *is the* value transition function*, which maps each value $v \in V_x$ to the set of values that can (immediately) follow it;*

- $D_x : V_x \rightarrow \mathbb{N}^+ \times (\mathbb{N}^+ \cup \{+\infty\})$ *is a function that maps each* $v \in V_x$ *to the pair* $(d^{x=v}_{min}, d^{x=v}_{max})$ *of minimum and maximum durations allowed for intervals where* $x = v$.

A *timeline* is a finite sequence of *tokens*, each denoting a value v and (the duration of) a time interval d, that describes how a state variable x behaves over time.

Definition 2 (Tokens and timelines)**.** *A* token *for x is a tuple* $\tau = (x, v, d)$*, where x is a state variable,* $v \in V_x$ *is the value held by the variable, and* $d \in \mathbb{N}^+$ *is the* duration *of the token, with* $D_x(v) = (d^{x=v}_{min}, d^{x=v}_{max})$ *and* $d^{x=v}_{min} \leq d \leq d^{x=v}_{max}$. *A* timeline *for a state variable x is a finite sequence* $\mathsf{T} = \langle \tau_1, \ldots, \tau_k \rangle$ *of tokens for x, for some* $k \in \mathbb{N}$*, such that, for any* $1 \leq i < k$*, if* $\tau_i = (x, v_i, d_i)$*, then* $v_{i+1} \in T_x(v_i)$.

For any timeline $\mathsf{T} = \langle \tau_1, \ldots, \tau_k \rangle$ and any token $\tau_i = (x, v_i, d_i)$ in T, we define the functions $\text{start}(\mathsf{T}, i) = \sum_{j=1}^{i-1} d_j$ and $\text{end}(\mathsf{T}, i) = \text{start}(\mathsf{T}, i) + d_i$. We call the *horizon* of T the end time of the last token in T, that is, $\text{end}(\mathsf{T}, k)$. We write $\text{start}(\tau_i)$ and $\text{end}(\tau_i)$ to indicate $\text{start}(\mathsf{T}, i)$ and $\text{end}(\mathsf{T}, i)$, respectively, when there is no ambiguity.

The overall behavior of state variables is subject to a set of temporal constraints known as *synchronization rules* (or simply *rules*). We start by defining their basic building blocks. Let $\mathscr{N}$ be a finite set of *token names*. *Atoms* are formulas of the following form:

$$\begin{aligned} atom &:= term \leq_{l,u} term \mid term <_{l,u} term \\ term &:= \text{start}(a) \mid \text{end}(a) \mid t \end{aligned}$$

where $a \in \mathscr{N}$, $l, t \in \mathbb{N}$, and $u \in \mathbb{N} \cup \{+\infty\}$. As an example, atom $\text{start}(a) \leq_{l,u} \text{end}(b)$ (resp., $\text{start}(a) <_{l,u} \text{end}(b)$) relates tokens a and b by stating that the end of b cannot precede (resp., must succeed) the beginning of a, and the distance between these two endpoints must be at least l and at most u. An atom $term \leq_{l,u} term$, with $l = 0$ and $u = +\infty$, is *qualitative* (the subscript is usually omitted in this case).

An *existential statement* $\mathscr{E}$ is a constraint of the form:

$$\exists a_1[x_1 = v_1] a_2[x_2 = v_2] \ldots a_n[x_n = v_n]. \, \mathscr{C}$$

where $x_1, \ldots, x_n$ are state variables, $v_1, \ldots, v_n$ are values, with $v_i \in V_{x_i}$, $a_1, \ldots, a_n$ are symbols from the set $\mathscr{N}$ of token names, and $\mathscr{C}$ is a finite conjunction of *atoms*, involving only tokens $a_1, \ldots, a_n$, plus, possibly, the *trigger token* (usually denoted by a_0) of the *synchronization rule* in which the existential statement is embedded, as described below.[1]

Intuitively, an existential statement asks for the existence of tokens $a_1, a_2, \ldots, a_n$ whose state variables take the corresponding values $v_1, v_2, \ldots, v_n$ and are such that their start and end times satisfy the atoms in $\mathscr{C}$.

Synchronization rules are clauses of one of the following forms:

$$\begin{aligned} a_0[x_0 = v_0] &\rightarrow \mathscr{E}_1 \vee \mathscr{E}_2 \vee \ldots \vee \mathscr{E}_k \\ \top &\rightarrow \mathscr{E}_1 \vee \mathscr{E}_2 \vee \ldots \vee \mathscr{E}_k \end{aligned}$$

where $a_0 \in \mathscr{N}$, x_0 is a state variable, $v_0 \in V_{x_0}$, and $\mathscr{E}_i$ is an existential statement, for each $1 \leq i \leq k$. In the former case, $a_0[x_0 = v_0]$ is called *trigger* and a_0 is the *trigger token*, and the rule is considered *satisfied* if *for all* the tokens a_0 for which the variable x_0 takes the value v_0, at least one of the existential statements is satisfied. In the latter case, the rule is said to be *triggerless*, and it states the truth of the body without any precondition.[2] We refer the reader to [7] for a formal account of the semantics of the rules.

---

[1] W.l.o.g., we assume that if a token a appears in the quantification prefix $\exists a_1[x_1 = v_1] a_2[x_2 = v_2] \ldots a_n[x_n = v_n]$ of $\mathscr{E}$, then at least one among $\text{start}(a)$ and $\text{end}(a)$ occurs in one of its atoms.

[2] W.l.o.g., for non-triggerless rules, we assume that both $\text{start}(a_0)$ and $\text{end}(a_0)$ occur in all of its existential statements.

A *timeline-based planning problem* consists of a set of state variables and a set of rules that represent the problem domain and the goal.

Definition 3 (Timeline-based planning problem)**.** *A* timeline-based planning problem *is defined as a pair* $P = (\mathsf{SV}, S)$*, where* SV *is a set of state variables and* S *is a set of synchronization rules involving state variables in* SV.

A *solution plan* for a given timeline-based planning problem is a set of timelines, one for each state variable, that satisfies all the synchronization rules.

Definition 4 (Plan and solution plan)**.** *A* plan *over a set of state variables* SV *is a finite set of timelines with the same horizon, one for each state variable* $x \in \mathsf{SV}$*. A* solution plan *for a timeline-based planning problem* $P = (\mathsf{SV}, S)$ *is a plan over* SV *such that all the rules in* S *are satisfied.*

The problem of determining whether a solution plan exists for a given timeline-based planning problem is EXPSPACE-complete [12].

Definition 5 (Qualitative timeline-based planning)**.** *A timeline-based planning problem* $P = (\mathsf{SV}, S)$ *is said to be* qualitative *if the following conditions hold:*

1. $D_x(v) = (1, +\infty)$*, for all state variables* $x \in \mathsf{SV}$ *and* $v \in V_x$.
2. *all synchronization rules in* S *involve only* qualitative *atoms.*

Unlike timeline-based planning, such a qualitative variant is PSPACE-complete [8]. A reduction of qualitative timeline-based planning to the nonemptiness problem for non-deterministic finite automata (NFA) has been provided in [8].

3 A Well-Behaved Fragment

In this section, we introduce a meaningful fragment of qualitative timeline-based planning for which we will show that it is possible to construct DFAs of singly exponential size.

The fragment is characterized by means of some conditions on the admissible patterns of synchronization rules (eager rules). The distinctive feature of eager rules is that they can be checked using an eager/greedy strategy, that is, when a relevant event (start/end of a token involved in some atom) occurs, we are guaranteed that the starting/ending point of such a token is useful for rule satisfaction. Instead, in case of non-eager rules, it may happen that a relevant event happens that is not useful for rule satisfaction: some analogous event in the future will be.

W.l.o.g., we assume that no constraint of the forms $\mathrm{start}(a) \leq \mathrm{end}(a)$ and $\mathrm{start}(a) < \mathrm{end}(a)$ occurs explicitly in synchronization rules, even though they hold tacitly, as they follow from the definition of token (Definition 2).

As a preliminary step, we define a sort of transitive closure of a clause. First, by slightly abusing the notation, we identify a clause $\mathscr{C}$ with the finite set of atoms occurring in it. Let t, t_1, t_2, t_3 be terms of the form $\mathrm{start}(a)$ or $\mathrm{end}(a)$, with $a \in \mathscr{N}$. We denote by $\hat{\mathscr{C}}$ the *transitive closure* of $\mathscr{C}$, defined as the smallest set of atoms including $\mathscr{C}$ and such that: *(i)* if term t occurs in $\mathscr{C}$, then atom $t \leq t$ belongs to $\hat{\mathscr{C}}$, *(ii)* if terms $\mathrm{start}(a)$ and $\mathrm{end}(a)$ both occur in $\mathscr{C}$ for some token name a, then atom $\mathrm{start}(a) < \mathrm{end}(a)$ belongs to $\hat{\mathscr{C}}$, *(iii)* if atom $t_1 < t_2$ belongs to $\hat{\mathscr{C}}$, then atom $t_1 \leq t_2$ belongs to $\hat{\mathscr{C}}$ as well, *(iv)* if atoms $t_1 \leq t_2$ and $t_2 \leq t_3$ belong to $\hat{\mathscr{C}}$, then atom $t_1 \leq t_3$ belongs to $\hat{\mathscr{C}}$ as well, *(v)* if atoms $t_1 < t_2$ and $t_2 \leq t_3$ belong to $\hat{\mathscr{C}}$, then atom $t_1 < t_3$ belongs to $\hat{\mathscr{C}}$ as well, *(vi)* if atoms $t_1 \leq t_2$ and $t_2 < t_3$ belong to $\hat{\mathscr{C}}$, then atom $t_1 < t_3$ belongs to $\hat{\mathscr{C}}$ as well.[3]

[3] W.l.o.g., we assume that $\hat{\mathscr{C}}$ is consistent, *i.e.*, it admits at least a solution. We point out that this check can be done in polynomial time, since it is an instance of linear programming.

Notice that, in some particular cases, condition *(ii)* may introduce in the closure of a clause atoms of the form $\text{start}(a) < \text{end}(a)$, which, according to our assumption, do not belong to any clause.

Let us now define the core notion of *eager rule*.

Definition 6 (Eager rules). *Let $\mathscr{R}$ be a synchronization rule and let $\mathscr{C}_1, \ldots, \mathscr{C}_k$ be the clauses occurring in its existential statements. We say that $\mathscr{R}$ is* eager *if and only if, for all $\mathscr{C} \in \{\mathscr{C}_1, \ldots, \mathscr{C}_k\}$ and $a_1, a_2 \in \mathscr{N}$ appearing in $\mathscr{C}$, the following conditions hold:*

1. *if both a_1 and a_2 are non-trigger tokens and $\{\text{start}(a_2) \leq \text{end}(a_1), \text{end}(a_1) \leq \text{end}(a_2)\} \subseteq \hat{\mathscr{C}}$, then $\text{end}(a_1) \leq \text{start}(a_2) \in \hat{\mathscr{C}}$ (i.e., the end of a_1 and the start of a_2 coincide),*
2. *if a_1 is either a trigger token or a non-trigger one, a_2 is a non-trigger token, and $\{\text{start}(a_2) \leq \text{start}(a_1), \text{start}(a_1) \leq \text{end}(a_2)\} \subseteq \hat{\mathscr{C}}$, then $\text{start}(a_1) \leq \text{start}(a_2) \in \hat{\mathscr{C}}$ (i.e., a_1 and a_2 start together), and*
3. *if a_1 is a trigger token, a_2 is a non-trigger one, and $\{\text{start}(a_1) \leq \text{start}(a_2), \text{end}(a_1) \leq \text{end}(a_2)\} \subseteq \hat{\mathscr{C}}$, then $\text{start}(a_2) \leq \text{start}(a_1) \in \hat{\mathscr{C}}$ (i.e., a_1 and a_2 start together).*

We define the *eager fragment* of a qualitative timeline-based planning problem as the set of qualitative timeline-based planning problems $P = (\mathsf{SV}, S)$ such that S contains only eager rules.

An explanation of the restrictions in Definition 6 is due. Given a non-trigger token a_2, Condition 1 forces any other non-trigger token a_1 ending during a_2 (that is, such that $\text{start}(a_2) \leq \text{end}(a_1) \leq \text{end}(a_2)$) to end exactly when a_2 starts, while Condition 2 forces any other (trigger or non-trigger) token a_1 starting during a_2 (that is, such that $\text{start}(a_2) \leq \text{start}(a_1) \leq \text{end}(a_2)$) to start simultaneously to a_2. Finally, whenever a non-trigger token a_2 starts during a trigger token a_1 and ends not before the end of a_1, Condition 3 forces the two tokens to start at the same time.

Conditions 1, 2, and 3 suffice to obtain a singly exponential DFA, whose construction will be illustrated in the next sections. We give here a short intuitive account of the rationale of the above conditions.

Consider the following rule:

$$a_0[x_0 = v_0] \rightarrow \exists a_1[x_1 = v_1]. \\ (\text{start}(a_0) = \text{start}(a_1) \wedge \text{end}(a_0) \leq \text{end}(a_1)),$$

where $\text{start}(a_0) = \text{start}(a_1)$ is an abbreviation for $\text{start}(a_0) \leq \text{start}(a_1) \wedge \text{start}(a_1) \leq \text{start}(a_0)$. This rule is eager because Conditions 1, 2, and 3 are fulfilled; in particular, we have that $\text{start}(a_0) = \text{start}(a_1)$. This is crucial for any DFA $\mathscr{A}$ recognizing solution plans, because, when $\mathscr{A}$ reads the event $\text{start}(a_0)$, it can *eagerly* and *deterministically* go to a state representing the fact that both $\text{start}(a_0)$ and $\text{start}(a_1)$ have happened. Moreover, if later it reads the event $\text{end}(a_1)$, but it has not read $\text{end}(a_0)$ yet, then it transitions to a rejecting state, that is, a state from which it cannot accept any plan.

Let us provide now an example of a non-eager rule that cannot be checked in an eager/greedy fashion. Consider the rule obtained from the above one by replacing $=$ with $\leq$:

$$a_0[x_0 = v_0] \rightarrow \exists a_1[x_1 = v_1]. \\ (\text{start}(a_0) \leq \text{start}(a_1) \wedge \text{end}(a_0) \leq \text{end}(a_1)).$$

This rule is *not* eager, because atom $\text{start}(a_1) \leq \text{start}(a_0)$ does not belong to $\hat{\mathscr{C}}$ (Condition 3 is violated). Indeed, for this rule, a DFA $\mathscr{A}$ that first reads event $\text{start}(a_0)$, but not $\text{start}(a_1)$, and then, strictly after, reads event $\text{start}(a_1)$ has to *nondeterministically* guess the order between the end of such a token a_1 and the end of a_0, making the construction of an automaton of singly exponential size impossible in the

general case. Indeed, if token a_1 ends before token a_0, the rule is not satisfied, but we cannot exclude the existence of another token for $x_1 = v_1$ that starts after that one and ends after the end of a_0, thus satisfying the rule.

We conclude by showing that excluding constraints of the forms $\mathsf{start}(a) \leq \mathsf{end}(a)$ and $\mathsf{start}(a) < \mathsf{end}(a)$ from clauses makes it sometimes possible to turn an otherwise non-eager rule into an eager one. As an example, rule $a_0[x_0 = v_0] \rightarrow \exists a_1[x_1 = v_1].(\mathsf{start}(a_1) < \mathsf{end}(a_1) \wedge \mathsf{start}(a_0) = \mathsf{end}(a_1))$ is not eager (Condition 2 is violated); however, it can be rewritten as $a_0[x_0 = v_0] \rightarrow \exists a_1[x_1 = v_1].\,\mathsf{start}(a_0) = \mathsf{end}(a_1)$, which is eager.

In what follows, we give a reduction from the plan-existence problem for the eager fragment of the qualitative timeline-based planning problem to the nonemptiness problem of DFAs of *singly exponential* size with respect to the original problem. The approach is inspired by those in [8, 14] for non-eager timeline-based planning problems, where an NFA of exponential size is built for any timeline-based planning problem. However, the reductions presented there use nondeterministic automata, which cannot be used as arenas to solve timeline-based games without a previous determinization step that would cause a second exponential blowup.

First, we show how to encode timelines and plans as finite words, and vice versa (Section 4). Then, given a planning problem P, we show how to build a DFA whose language encodes the set of solution plans for P. The DFA consists of the intersection of two DFAs: one aims at verifying the constraint on the alternation of token values expressed by functions T_x, for $x \in \mathsf{SV}$, as well as that the word correctly encodes a plan over SV (Section 5); the other one verifies that the encoded plan is indeed a solution plan for P (Section 6).

From now on, we consider only qualitative timeline-based planning problems belonging to the eager fragment and, for the sake of brevity, we sometimes refer to them simply as *planning problems*.

4 From Plans to Finite Words and *Vice Versa*

In this section, as a first step in the construction of the DFA corresponding to an eager qualitative timeline-based planning problem, we show how to encode timelines and plans as *words* that can be recognized by an automaton, and *vice versa*.

Let $P = (\mathsf{SV}, S)$ be an eager qualitative timeline-based planning problem, and let $V = \cup_{x \in \mathsf{SV}} V_x$. We define the *initial alphabet* Σ^I_{SV} as $(\{-\} \times V)^{\mathsf{SV}}$, that is the set of functions from SV to $(\{-\} \times V)$.[4] Similarly, we define the *non-initial alphabet* Σ^N_{SV} as $((V \times V) \cup \{\circlearrowleft\})^{\mathsf{SV}}$, where the pairs $(v, v') \in V \times V$ are supposed to represent the value v of the token that just ended and the value v' of the token that has just started, and $\circlearrowleft$ represents the fact that the value for the state variable has not changed. The *input alphabet* (or, simply, *alphabet*) associated with SV and denoted by Σ_{SV} is the union $\Sigma^I_{\mathsf{SV}} \cup \Sigma^N_{\mathsf{SV}}$. Observe that the size of the alphabet Σ_{SV} is at most exponential in the size of SV, precisely $|\Sigma_{\mathsf{SV}}| = |\Sigma^I_{\mathsf{SV}}| + |\Sigma^N_{\mathsf{SV}}| = |V|^{|\mathsf{SV}|} + (|V|^2 + 1)^{|\mathsf{SV}|}$.

We now show how to encode the basic structure[5] underlying each plan over SV as a word in $\Sigma^I_{\mathsf{SV}} \cdot (\Sigma^N_{\mathsf{SV}})^* \cup \{\varepsilon\}$, where ε is the empty word (and corresponds to the empty plan), $(\Sigma^N_{\mathsf{SV}})^*$ is the Kleene's closure of Σ^N_{SV}, and $\cdot$ denotes the concatenation symbol. Intuitively, let ν be the symbol at position i of a word $\sigma \in \Sigma^I_{\mathsf{SV}} \cdot (\Sigma^N_{\mathsf{SV}})^* \cup \{\varepsilon\}$. Then, if $\nu(x) = (v, v')$ for some $x \in \mathsf{SV}$, then at time i a new token begins in the timeline for x with value v'; instead, if $\nu(x) = \circlearrowleft$, then no change happens at time i in the timeline

[4] The symbol $\{-\}$ is a technicality that allows us to consider pairs instead of just values in V.

[5] With "basic structure" we refer to the fact that, in this section, we neither take into account the transition functions T_x of state variables nor their domains V_x (*cf.* Definition 1), which will be dealt with in Section 5.

for x, meaning that no token ends at that time point in the timeline for x. The value v of the token ending at time i will be used later in the construction of the automata.

We remark that not all words in $\Sigma^I_{\mathsf{SV}} \cdot (\Sigma^N_{\mathsf{SV}})^* \cup \{\varepsilon\}$ correspond to plans over SV: for a word to correctly encode a plan, the information carried by the word about the value of a starting token and the one associated to the end of the same token must coincide. Formally, given a word $\sigma = \langle \sigma_0, \ldots, \sigma_{|\sigma|-1} \rangle \in \Sigma^I_{\mathsf{SV}} \cdot (\Sigma^N_{\mathsf{SV}})^* \cup \{\varepsilon\}$ and a state variable $x \in \mathsf{SV}$, let $changes(x) = (i^x_0, i^x_1, \ldots, i^x_{k^x-1})$, for some $k^x \in \mathbb{N}$, be the increasing sequence of positions where x changes, *i.e.*, $\sigma_i(x) \neq \circlearrowleft$ if and only if $i \in changes(x)$, for all $i \in \{0, \ldots, |\sigma|-1\}$. We denote by v^x_i and $\hat{v}^x_i$ the first and the second component of $\sigma_i(x)$, respectively, for all $x \in \mathsf{SV}$ and $i \in changes(x)$. We omit superscripts ^{x} when there is no risk of ambiguity.

Definition 7 (Words weakly-encoding plans)**.** *Let* $\sigma \in \Sigma^I_{\mathsf{SV}} \cdot (\Sigma^N_{\mathsf{SV}})^*$ *and let* $changes(x) = (i_0, i_1, \ldots, i_{k-1})$. *We say that* σ weakly-encodes a plan over SV *if* $\hat{v}_{i_{h-1}} = v_{i_h}$ *for all* $x \in \mathsf{SV}$ *and* $h \in \{1, \ldots, k-1\}$. *If this is the case, then the* plan induced by σ *is the set* $\{\mathsf{T}_x \mid x \in \mathsf{SV}\}$, *where* $\mathsf{T}_x = \langle (x, \hat{v}_{i_0}, i_1 - i_0), (x, \hat{v}_{i_1}, i_2 - i_1), \ldots, (x, \hat{v}_{i_{k-1}}, i_k - i_{k-1}) \rangle$ *and* $i_k = |\sigma|$, *for all* $x \in \mathsf{SV}$.

Intuitively, if a word weakly-encodes a plan, then it captures the dynamics of a state variable modulo its domain and its transition function, which will be taken care of in the next section. A converse correspondence from plans to words can be defined accordingly.

Before concluding the section, we introduce another notation that will come handy later. We denote by $events(\sigma)$ the set of events (beginning/ending of a token) occurring at a given time, encoded in the alphabet symbol σ. Formally, $events(\sigma)$ is the smallest set such that:

- if $\sigma(x) = (v, v')$ for some x, then $\{end(x, v), start(x, v')\} \subseteq events(\sigma)$, and
- if $\sigma(x) = (-, v')$ for some x, then $start(x, v') \in events(\sigma)$.

5 DFA Accepting Plans

Given an eager qualitative timeline-based planning problem $P = (\mathsf{SV}, S)$, we show how to build a DFA $\mathscr{T}_{\mathsf{SV}}$, of size at most exponential in the size of P, accepting words that correctly encode plans over SV, that is, words that weakly-encode plans (*cf.* Definition 7) *and* comply with the constraints on the alternation of token values expressed by functions T_x, for $x \in \mathsf{SV}$. In the next section, we show how to obtain a DFA, of size at most exponential in the size of P, that accepts exactly the *solution plans* for P.

For every planning problem $P = (\mathsf{SV}, S)$, the DFA $\mathscr{T}_{\mathsf{SV}}$ is the tuple $\langle Q_{\mathsf{SV}}, \Sigma_{\mathsf{SV}}, \delta_{\mathsf{SV}}, q^0_{\mathsf{SV}}, F_{\mathsf{SV}} \rangle$, whose components are defined as follows.

- Q_{SV} is the set of *states* of $\mathscr{T}_{\mathsf{SV}}$. Intuitively, a state of $\mathscr{T}_{\mathsf{SV}}$ keeps track of the token values of the timelines at the current and the previous step of the run. Therefore, a state is a function mapping each state variable x into a pair (v, v'), where v' (resp., v) denotes the token value of timeline x at the current (resp., previous) step. To formally define Q_{SV}, we exploit the definition of alphabet Σ_{SV} from Section 4. Mostly, states are alphabet symbols, except for those functions $\sigma \in \Sigma_{\mathsf{SV}}$ assigning to at least one state variable $x \in \mathsf{SV}$ value $\circlearrowleft$. For technical reasons, we also need a fresh *initial state* q^0_{SV} and a fresh *rejecting sink state* $\mathbf{s}_{\mathsf{SV}}$.

  Formally, $Q_{\mathsf{SV}} = \big(\Sigma_{\mathsf{SV}} \setminus \overline{Q}_{\mathsf{SV}}\big) \cup \{q^0_{\mathsf{SV}}, \mathbf{s}_{\mathsf{SV}}\}$, where $\overline{Q}_{\mathsf{SV}} = \{\sigma \in \Sigma_{\mathsf{SV}} \mid \sigma(x) = \circlearrowleft \text{ for some } x \in \mathsf{SV}\}$. Clearly, the size of Q_{SV} is at most as the size of Σ_{SV}, which is in turn at most exponential in the size of P.

- Σ_{SV} is the *input alphabet*, defined as in Section 4.

- $\delta_{\mathsf{SV}} : Q_{\mathsf{SV}} \times \Sigma_{\mathsf{SV}} \to Q_{\mathsf{SV}}$ is the *transition function*. Towards a definition of δ_{SV}, we say that an alphabet symbol $\sigma \in \Sigma_{\mathsf{SV}}$ is *compatible* with a state $\sigma_1 \in Q_{\mathsf{SV}}$ (we use for states the same symbols as for the alphabet, i.e., $\sigma, \sigma_1, \sigma_2, \ldots$, to stress the fact that states are closely related to alphabet symbols) if one of the following holds: *(i)* $\sigma_1 = q^0_{\mathsf{SV}}$ is the initial state and $\sigma \in \Sigma^I_{\mathsf{SV}}$ is an initial symbol such that for each $x \in \mathsf{SV}$ it holds that $\sigma(x) = (-, v)$ with $v \in V_x$; *(ii)* $\sigma_1 = (v, v') \in \Sigma_{\mathsf{SV}} \setminus \overline{Q}_{\mathsf{SV}}$ and $\sigma \in \Sigma^N_{\mathsf{SV}}$ is a non-initial symbol such that for each $x \in \mathsf{SV}$ either $\sigma(x) = \circlearrowleft$ or $\sigma(x) = (v', v'')$ with $v'' \in T_x(v') \cap V_x$.

  Now, $\delta_{\mathsf{SV}} : Q_{\mathsf{SV}} \times \Sigma_{\mathsf{SV}} \to Q_{\mathsf{SV}}$ is defined as follows. For all $\sigma_1 \in Q_{\mathsf{SV}}$ and $\sigma \in \Sigma_{\mathsf{SV}}$, if σ is not compatible with σ_1 or σ_1 is the sink state (i.e., $\sigma_1 = \mathbf{s}_{\mathsf{SV}}$), then $\delta(\sigma_1, \sigma) = \mathbf{s}_{\mathsf{SV}}$; otherwise

  - if σ_1 is the initial state (i.e., $\sigma_1 = q^0_{\mathsf{SV}}$), then $\delta(\sigma_1, \sigma) = \sigma$; in other words, in this case the automaton transitions to the state represented by the input letter;
  - if $\sigma_1 \in \Sigma_{\mathsf{SV}} \setminus \overline{Q}_{\mathsf{SV}}$, then $\delta(\sigma_1, \sigma) = \sigma_2$, where $\sigma_2(x) = \sigma_1(x)$ if $\sigma(x) = \circlearrowleft$, and $\sigma_2(x) = \sigma(x)$ otherwise, for all $x \in \mathsf{SV}$; intuitively, the automaton transitions into a state keeping track of the updated information about which tokens have changed value and which ones have not.

  We point out that, in both cases, the automaton transitions to the next state in a deterministic fashion.

- $F_{\mathsf{SV}} = Q_{\mathsf{SV}} \setminus \{\mathbf{s}_{\mathsf{SV}}\}$ is the set of *final states*.

Correctness of the DFA $\mathscr{T}_{\mathsf{SV}}$ is proved by the next lemma.

Lemma 1. *Let $P = (\mathsf{SV}, S)$ be an eager qualitative timeline-based planning problem. Then, words accepted by $\mathscr{T}_{\mathsf{SV}}$ are exactly those encoding plans over* SV. *Moreover the size of $\mathscr{T}_{\mathsf{SV}}$ is at most exponential in the size of P.*

6 DFA Accepting Solution Plans

In this section, we go through the construction of an automaton recognizing solution plans for a planning problem. Towards that, it will come in handy to define some auxiliary structures, namely *blueprints*, *snapshots* and *viewpoints*; moreover, we will define how these structures evolve and give a high-level intuition for each of them.

Let $P = (\mathsf{SV}, S)$ be an eager qualitative timeline-based planning problem, and let $V = \cup_{x \in \mathsf{SV}} V_x$. We first show how to build a DFA $\mathscr{A}_P$, whose size is at most exponential in the size of P, that accepts exactly those words encoding solutions plans for P when restricted to words encoding plans over SV. In different terms, if a word encodes a plan over SV, then it is accepted by $\mathscr{A}_P$ if and only if it encodes a solution plan for P. However, $\mathscr{A}_P$ may also accept words that do not encode a plan over SV. Therefore, we need the intersection of such a DFA $\mathscr{A}_P$ with DFA $\mathscr{T}_{\mathsf{SV}}$ from the previous section.

In the following, we use *preorders* to represent the ordering relation imposed by synchronization rules. Each existential statement of the form $\exists a_1[x_1 = v_1] a_2[x_2 = v_2] \ldots a_n[x_n = v_n].\mathscr{C}$, with $\mathscr{C}$ conjunction of atoms, identifies a *preorder* whose domain is the set of terms $\mathrm{start}(a)/\mathrm{end}(a)$ occurring in $\mathscr{C}$, and where term t_1 precedes term t_2 in the preorder whenever $t_1 \leq t_2$ belongs to $\hat{\mathscr{C}}$.

For a preorder $\mathscr{P}$, we denote by $dom(\mathscr{P})$ its domain and by $\preceq_{\mathscr{P}}$ the ordering relation. Moreover, we use $x \equiv_{\mathscr{P}} y$ to denote the fact that both $x \preceq_{\mathscr{P}} y$ and $y \preceq_{\mathscr{P}} x$ hold, and $x \prec_{\mathscr{P}} y$ to denote the fact that $x \preceq_{\mathscr{P}} y$ holds but $y \preceq_{\mathscr{P}} x$ does not. Finally, we denote by $[x]_{\equiv_{\mathscr{P}}}$ the equivalence class of x with respect to $\equiv_{\mathscr{P}}$ for every $x \in dom(\mathscr{P})$, that is, $[x]_{\equiv_{\mathscr{P}}} = \{y \in dom(\mathscr{P}) \mid y \equiv_{\mathscr{P}} x\}$. We omit the subscript $\mathscr{P}$ when it is clear from the context. A preorder $\mathscr{P}$ induces a directed acyclic graph (DAG) $G = (V, A)$, where V is the set of equivalence classes, that is, $V = \{[x]_{\equiv} \mid x \in dom(\mathscr{P})\}$, and, for every $x, y \in dom(\mathscr{P})$ there is an arc

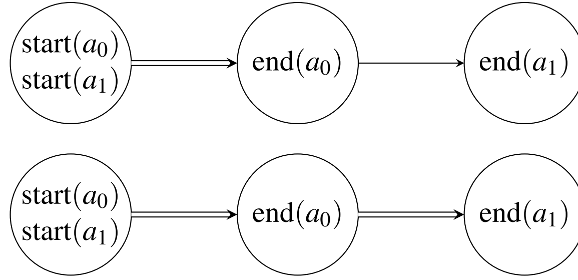


Figure 1: Above, we show the blueprint for the unique existential statement in the rule $a_0[x_0 = v_0] \rightarrow \exists a_1[x_1 = v_1].(\text{start}(a_0) = \text{start}(a_1) \wedge \text{end}(a_0) \leq \text{end}(a_1))$, from Section 3. It forces token a_0 to either be a prefix of or coincide with token a_1. Below, the blueprint obtained replacing $\text{end}(a_0) \leq \text{end}(a_1)$ with $\text{end}(a_0) < \text{end}(a_1)$, that forces a_1 to be a strict prefix of a_1.

from $[x]_\equiv$ to $[y]_\equiv$ in A (denoted by $([x]_\equiv, [y]_\equiv) \in A$ or $[x]_\equiv \rightarrow [y]_\equiv$ when set A is clear from the context) if and only if $x \prec y$ and there is no $w \in dom(\mathscr{P})$ such that $x \prec w$ and $w \prec y$. Clearly, there is a path from $[x]_\equiv$ to $[y]_\equiv$ (denoted by $[x]_\equiv \rightarrow^* [y]_\equiv$) if and only if $x \preceq y$. Therefore, given an existential statement $\mathscr{E}$ occurring in a synchronization rule $\mathscr{R}$, we refer to the associated preorder and DAG as, respectively, $\mathscr{P}_\mathscr{E}$ and $G_\mathscr{E}$.

It is important to observe that a conjunction of atoms $\mathscr{C}$ within an existential statement $\mathscr{E}$ contains atoms of both forms $t_1 \leq t_2$ and $t_1 < t_2$. To keep track of these different constraints in DAG $G_\mathscr{E} = (V, A)$ associated with $\mathscr{E}$, we identify the subset $A_< \subseteq A$ of arcs of $G_\mathscr{E}$ as the set $A_< = \{([x]_\equiv, [y]_\equiv) \in A \mid x < y \in \hat{\mathscr{C}}\}$. We sometimes write $[x]_\equiv \Rightarrow [y]_\equiv$ for $([x]_\equiv, [y]_\equiv) \in A_<$, when A is clear from the context. Figure 1 shows such a difference.

Let $\mathscr{E}$ be an existential statement occurring in a rule $\mathscr{R}$ and $G_\mathscr{E}$ the DAG associated with $\mathscr{E}$. The set of *events associated with a vertex* $[x]_\equiv$ of $G_\mathscr{E}$, denoted by $events_{G_\mathscr{E}}([x]_\equiv)$, is the smallest set such that if $\text{start}(a) \in [x]_\equiv$ (resp., $\text{end}(a) \in [x]_\equiv$) and $a[y = v]$ either occurs in $\mathscr{E}$ or is the trigger of $\mathscr{R}$, then $start(y, v) \in events_{G_\mathscr{E}}([x]_\equiv)$ (resp., $end(y, v) \in events_{G_\mathscr{E}}([x]_\equiv)$). The set of *events associated with a subset* V' *of vertices* of $G_\mathscr{E}$, denoted by $events_{G_\mathscr{E}}(V')$, is the set $\bigcup_{v \in V'} events_{G_\mathscr{E}}(v)$.

6.1 Blueprints, Snapshots, and Viewpoints

A DAG associated with an existential statement $\mathscr{E}$ is also called a *blueprint* for $\mathscr{E}$. A *snapshot* for an existential statement $\mathscr{E}$ is a pair (G, K), where $G = (V, A)$ is a blueprint for $\mathscr{E}$ and $K \subseteq V$ is a *downward closed* subset of vertices of G, that is, $v \in K$ implies $v' \in K$ for all $v' \in V$ with $v' \rightarrow^* v$. The number of different snapshots for $\mathscr{E}$ is at most $2^{|V|}$, hence at most exponential in the size of P, denoted by $|P|$. A *viewpoint* $\mathbb{V}$ for a rule $\mathscr{R}$ is a set of snapshots for existential statements in $\mathscr{R}$, at most one for each statement. Let $n_\mathscr{R}$ be the number of existential statements in $\mathscr{R}$; then, it is easy to see that the number of different viewpoints for $\mathscr{R}$ is at most $(2^{|P|})^{n_\mathscr{R}}$, hence exponential in the size of P. If $K = \varnothing$ for all $(G, K) \in \mathbb{V}$, then $\mathbb{V}$ is the *initial* viewpoint of $\mathscr{R}$; analogously, if K is the entire set of vertices of G, for some $(G, K) \in \mathbb{V}$, then $\mathbb{V}$ is a *final* viewpoint of $\mathscr{R}$.

Intuitively, a viewpoint checks the satisfaction of a rule $\mathscr{R}$ by recognizing when at least one existential statement has been fulfilled. This check works by collecting, for each existential statement, information about the tokens seen so far along the plan into snapshots, which are downward closed and accurately represent all relevant symbols read. How information is collected, thus how viewpoints and snapshots evolve, is explained in the following.

States of automata $\mathscr{A}_P$ are sets of viewpoints containing at least one viewpoint for each rule of P (besides a fresh rejecting sink state $\mathbf{s}_P$); recall that viewpoints are in turn sets of snapshots. Therefore, to define automata runs, we first show how snapshots and viewpoints evolve upon reading an alphabet symbol. To this end, we need the following notions.

For a snapshot (G,K), we set $next(G,K) = K'$, where K' is the largest downward closed subset of vertices of G for which there is no pair of vertices $v, v' \in K' \setminus K$ with $v \Rightarrow v'$. Moreover, given an alphabet symbol $\sigma \in \Sigma_{\mathsf{SV}}$, we define $next((G,K),\sigma) = K'$, where K' is the largest downward closed subset of vertices of $next(G,K)$ such that $events_G(K' \setminus K) \subseteq events(\sigma)$. We say that snapshot (G,K) is *compatible* with symbol σ if for all $start(x,v) \in events_G(K)$ and $end(x,v) \in events(\sigma) \cap events_G(V \setminus K)$, it holds that $end(x,v) \in events_G(next((G,K),\sigma))$.

Intuitively, during a run of the automaton, a snapshot (G,K) evolves by suitably extending K. $next(G,K)$ identifies the only vertices that can appear in such an extension independently from the alphabet symbol read, that is, vertices in $V \setminus K$ reachable (from K) without crossing arcs in $A_<$. The exact extension, however, depends on the actual symbol σ read by the automaton: K cannot be extended with events that are not included in σ. Therefore, $next((G,K),\sigma)$ identifies precisely how a snapshot evolves. At last, observe that for a snapshot to be allowed to evolve upon reading a symbol, it must be guaranteed that no token ending is overlooked, which is formalized by the notion of compatibility of a snapshot with a symbol.

We can now characterize the evolution of snapshots and viewpoints when reading an alphabet symbol $\sigma \in \Sigma_{\mathsf{SV}}$. The *evolution of a snapshot* (G,K) when reading σ, denoted $evol((G,K),\sigma)$, is snapshot $(G, next((G,K),\sigma))$, if (G,K) is compatible with σ; it is undefined otherwise. The *evolution of a viewpoint* $\mathbb{V}$ when reading σ, denoted $evol(\mathbb{V},\sigma)$, is viewpoint $\mathbb{V}'$, defined as the smallest set such that for all $(G,K) \in \mathbb{V}$, if $evol((G,K),\sigma)$ is defined, then $evol((G,K),\sigma) \in \mathbb{V}'$.

6.2 States, Initial State, and Final States of $\mathscr{A}_P$

We have already mentioned that *states* of $\mathscr{A}_P$ are sets of viewpoints containing at least one viewpoint for each rule $\mathscr{R} \in S$ (recall that S is the set of rules in planning problem P), besides a fresh rejecting sink state $\mathbf{s}_P$. However, since it is crucial for us to bound the size of $\mathscr{A}_P$ to be at most exponential in the one of P, we impose the *linearity condition*, formalized in what follows.

First, recall that, given a rule $\mathscr{R}$, featuring existential statements $\mathscr{E}_1,\dots,\mathscr{E}_{n_\mathscr{R}}$, a viewpoint $\mathbb{V}$ for $\mathscr{R}$ only contains at most one snapshot for each existential statement in $\mathscr{R}$; therefore, it holds that $|\mathbb{V}| \leq n_\mathscr{R}$ and there is a partial surjective function $f_\mathbb{V} : \{\mathscr{E}_1,\dots,\mathscr{E}_{n_\mathscr{R}}\} \to \mathbb{V}$, where $f_\mathbb{V}(\mathscr{E})$ is the only snapshot for $\mathscr{E}$ in $\mathbb{V}$, if any, for all $\mathscr{E} \in \{\mathscr{E}_1,\dots,\mathscr{E}_{n_\mathscr{R}}\}$.

Now, for all rules $\mathscr{R} \in S$, let $\Upsilon_\mathscr{R}$ be the set of viewpoints for $\mathscr{R}$, and let $\Upsilon_P = \bigcup_{\mathscr{R}\in S} \Upsilon_\mathscr{R}$. We define an ordering relation $\preceq$ between viewpoints: for all $\mathbb{V},\mathbb{V}' \in \Upsilon_P$, it holds that $\mathbb{V} \preceq \mathbb{V}'$ if and only if *(i)* $\mathbb{V},\mathbb{V}' \in \Upsilon_\mathscr{R}$ for some $\mathscr{R} \in S$, *(ii)* $dom(f_{\mathbb{V}'}) \subseteq dom(f_\mathbb{V})$,[6] and *(iii)* for all $\mathscr{E} \in dom(f_{\mathbb{V}'})$, we have that $f_\mathbb{V}(\mathscr{E}) = (G,K)$, $f_{\mathbb{V}'}(\mathscr{E}) = (G,K')$, and $K \subseteq K'$. Intuitively, $\mathbb{V} \preceq \mathbb{V}'$ captures the fact that $\mathbb{V}'$ has gone further than $\mathbb{V}$ in matching input symbols to satisfy a rule. Therefore, a snapshot in $\mathbb{V}$ either evolved into one in $\mathbb{V}'$, according to the symbols read, or has disappeared because it is not compatible with some of the symbols read, and thus it cannot be used anymore to satisfy the rule.

At this point, we can formalize the linearity condition, crucial to constrain the size of $\mathscr{A}_P$ (Lemma 2).

Definition 8 (Linearity condition)**.** *A set of viewpoints Υ satisfies the* linearity condition *if for all viewpoints $\mathbb{V},\mathbb{V}' \in \Upsilon$ and rules $\mathscr{R} \in S$, if $\mathbb{V},\mathbb{V}' \in \Upsilon_\mathscr{R}$, then $\mathbb{V} \preceq \mathbb{V}'$ or $\mathbb{V}' \preceq \mathbb{V}$ holds.*

[6] For a partial function f, we denote by $dom(f)$ the set of elements where f is defined.

Intuitively, we impose all viewpoints for the same rule in a state of $\mathscr{A}_P$ to be linearly ordered.

We are now ready to formally characterize the set of *states* of $\mathscr{A}_P$, consisting of the sets $\Upsilon \subseteq \Upsilon_P$ of viewpoints that contain at least one viewpoint for each rule $\mathscr{R} \in S$ and that satisfy the linearity condition, and including, in addition, a fresh *rejecting sink state* $\mathbf{s}_P$. We denote it by Q_P.

The *initial state* q_P^0 of $\mathscr{A}_P$ is the set $\{\mathbb{V}^0_{\mathscr{R}} \mid \mathscr{R} \in S\}$, where $\mathbb{V}^0_{\mathscr{R}}$ is the initial viewpoint of rule $\mathscr{R}$.

Towards a definition of the set F_P of *final states* of $\mathscr{A}_P$, we introduce the notion of *enabled viewpoints*. A viewpoint $\mathbb{V}$ for rule $\mathscr{R} \in S$ is *enabled* if either $\mathscr{R}$ is triggerless or $\mathscr{R}$ has trigger token a_0 and $\mathrm{start}(a_0) \in K$ for some $(G,K) \in \mathbb{V}$. A state q of $\mathscr{A}_P$ is *final* if every enabled viewpoint therein is final.

6.3 Transition Function of $\mathscr{A}_P$

The last step of our construction is the definition of the *transition function* δ_P for automaton $\mathscr{A}_P$.

To this end, we introduce the notion of alphabet symbol *enabling* a viewpoint $\mathbb{V}$ along with the one of states of $\mathscr{A}_P$ *compatible* with an alphabet symbol. Let $\mathbb{V}$ be a viewpoint for a non-triggerless rule $\mathscr{R}$ with trigger token a_0 and $\sigma \in \Sigma_{\mathsf{SV}}$ an alphabet symbol. We say that σ *enables* $\mathbb{V}$ if there is $(G,K) \in \mathbb{V}$ with $\mathrm{start}(a_0) \in \mathit{next}((G,K),\sigma)$. Moreover, we say that a state $q \in Q_P \setminus \{\mathbf{s}_P\}$ is *compatible* with σ if for all non-triggerless rules $\mathscr{R} \in S$, with trigger token $a_0[x_0 = v_0]$, if $\mathit{start}(x_0, v_0) \in \mathit{events}(\sigma)$, then there is a viewpoint $\mathbb{V} \in q$ such that σ enables $\mathbb{V}$.

We are now ready to define the *transition function* δ_P of $\mathscr{A}_P$. For all $q \in Q_P$ and alphabet symbol $\sigma \in \Sigma_{\mathsf{SV}}$:

- if $q = \mathbf{s}_P$ or q is not compatible with σ, then $\delta(q,\sigma) = \mathbf{s}_P$;
- otherwise, $\delta(q,\sigma) = q'$, where q' is the smallest set such that for all $\mathbb{V} \in q$
  - $\mathit{evol}(\mathbb{V},\sigma) \in q'$ and
  - if σ enables $\mathbb{V}$, then $\mathbb{V} \in q'$.

Lemma 2. *Let $P = (\mathsf{SV}, S)$ be an eager qualitative timeline-based planning problem. Then, each finite word over Σ_{SV} that encodes a plan over SV is accepted by $\mathscr{A}_P$ if and only if it encodes a solution plan for P. Moreover, the size of $\mathscr{A}_P$ is at most exponential in the size of P.*

Proof. For lack of space, we omit the proof of soundness showing that the automaton accepts the correct language as claimed. Instead, we show that the size of $\mathscr{A}_P$ is indeed at most exponential in the size of P.

Let k be the largest number of existential statements in a rule of P and k' the largest number of atoms in an existential statement of P. Thanks to the linearity rule enjoyed by states of P, it is not difficult to convince oneself that the number of different viewpoints for the same rule in a state $q \in Q_P$ to be at most $k \times k'$. Thus, each state in Q_P contains at most $|S| \times k \times k'$ different viewpoints (the product of the number of rules in P by the number of different viewpoints for the same rule).

Therefore, the size of Q_P is at most $|\Upsilon_P|^{(|S| \times k \times k')}$. Clearly, $(|S| \times k \times k')$ is at most polynomial in the size of P. Since $|\Upsilon_P| \leq \sum_{\mathscr{R} \in S} |\Upsilon_{\mathscr{R}}|$ and, as already pointed out, $|\Upsilon_{\mathscr{R}}|$ is at most exponential in the size of P, we can conclude that the size of Q_P is at most exponential in the size of P. □

Theorem 1. *Let $P = (\mathsf{SV}, S)$ be an eager qualitative timeline-based planning problem. Then, the words accepted by the intersection automaton of $\mathscr{A}_P$ and $\mathscr{T}_{\mathsf{SV}}$ are exactly those encoding solution plans for P. Moreover, the size of the intersection automaton of $\mathscr{A}_P$ and $\mathscr{T}_{\mathsf{SV}}$ is at most* exponential *in the size of P.*

7 A Maximal Subset of Allen's Relations

Allen's interval algebra is a formalism for temporal reasoning introduced in [2]. It identifies all possible relations between pairs of time intervals over a linear order and specifies a machinery to reason about them. In this section, we isolate the maximal subset of Allen's relations captured by the eager fragment of qualitative timeline-based planning. To this end, we show how to map Allen's relations over tokens in terms of their endpoints, that is, as conjunctions of atoms over terms $\text{start}(a), \text{start}(b), \text{end}(a), \text{end}(b)$, for token names a and b. Then, we check which relation encoding satisfies the conditions of Definition 6. Let $a, b \in \mathcal{N}$.

- a before b (b after a) can be defined as $\text{end}(a) < \text{start}(b)$.
- a meets b (b is-met-by a) can be defined as $\text{end}(a) = \text{start}(b)$.
- a ends b (b is-ended-by a) can be defined as $\text{start}(b) < \text{start}(a) \wedge \text{end}(a) = \text{end}(b)$.
- a starts b (b is-started-by a) can be defined as $\text{start}(a) = \text{start}(b) \wedge \text{end}(a) < \text{end}(b)$.
- a overlaps b (b is-overlapped-by a) can be defined as $\text{start}(a) < \text{start}(b) \wedge \text{start}(b) < \text{end}(a) \wedge \text{end}(a) < \text{end}(b)$.
- a during b (b contains a) can be defined as $\text{start}(b) < \text{start}(a) \wedge \text{end}(a) < \text{end}(b)$.
- $a = b$ can be defined as $\text{start}(a) = \text{start}(b) \wedge \text{end}(a) = \text{end}(b)$.

It is not difficult to see that, if one of the tokens, let's say a, is the trigger token, then the encodings not complying with Definition 6 are the ones for Allen's relations ends, is-ended-by, overlaps, is-overlapped-by, and during. Thus, the maximal subset of Allen's relations that can be captured by an instance of the eager fragment of the timeline-based planning problem consists of relations before, after, meets, is-met-by, starts, is-started-by, contains, and $=$.

As an example, consider relation overlaps and let $\mathscr{C} = \{\text{start}(a) < \text{start}(b), \text{start}(b) < \text{end}(a), \text{end}(a) < \text{end}(b)\}$ be its encoding. Clearly, the transitive closure $\hat{\mathscr{C}}$ of $\mathscr{C}$ (cf. Section 3) includes also $\text{start}(a) \leq \text{start}(b)$ and $\text{end}(a) \leq \text{end}(b)$ but it does not include $\text{start}(b) \leq \text{start}(a)$, thus violating Condition 2 of Definition 6. A similar argument can be used for relations ends, is-ended-by, is-overlapped-by, and during.

If, instead, none of the token is a trigger token, then the only Allen's relations not violating any of the conditions of Definition 6 are before, after, meets, and is-met-by. We omit the details.

8 Conclusions

In this paper, we identified a meaningful fragment of timeline-based planning whose solutions can be recognized by DFAs of singly exponential size. Specifically, we identified restrictions on the allowed synchronization rules, which we called *eager rules*, for which we showed how to build the corresponding deterministic automaton of exponential size, that can then be directly exploited to synthesize strategies. Moreover, we isolated a maximal subset of Allen's relations captured by such a fragment.

Whether the fragment of timeline-based planning identified by the eager rules is maximal or not is an open question currently under study. Further research directions include a parametrized complexity analysis over the number of synchronization rules and a characterization in terms of temporal logics, like the one in [15].

References

[1] Renato Acampora, Luca Geatti, Nicola Gigante, Angelo Montanari & Valentino Picotti (2022): *Controller Synthesis for Timeline-based Games*. In Pierre Ganty & Dario Della Monica, editors: *Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022*, *EPTCS* 370, pp. 131–146, doi:10.4204/EPTCS.370.9.

[2] James F. Allen (1983): *Maintaining Knowledge about Temporal Intervals*. *Commun. ACM* 26(11), pp. 832–843, doi:10.1145/182.358434.

[3] Laura Bozzelli, Alberto Molinari, Angelo Montanari & Adriano Peron (2018): *Complexity of Timeline-Based Planning over Dense Temporal Domains: Exploring the Middle Ground*. In Andrea Orlandini & Martin Zimmermann, editors: *Proceedings of the 9th International Symposium on Games, Automata, Logics, and Formal Verification*, *EPTCS* 277, pp. 191–205, doi:10.4204/EPTCS.277.14.

[4] Laura Bozzelli, Alberto Molinari, Angelo Montanari & Adriano Peron (2018): *Decidability and Complexity of Timeline-Based Planning over Dense Temporal Domains*. In Michael Thielscher, Francesca Toni & Frank Wolter, editors: *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, AAAI Press, pp. 627–628. Available at https://aaai.org/ocs/index.php/KR/KR18/paper/view/17995.

[5] Steve Chien, Gregg Rabideau, Russell Knight, Robert Sherwood, Barbara Engelhardt, Darren Mutz, Tara Estlin, Benjamin Smith, Forest Fisher, T Barrett et al. (2000): *ASPEN-Automating space mission operations using automated planning and scheduling*. In: *SpaceOps 2000*, AIAA Press.

[6] Steve A. Chien, Rob Sherwood, Daniel Tran, Benjamin Cichy, Gregg Rabideau, Rebecca Castaño, Ashley Davies, Rachel Lee, Dan Mandl, Stuart Frye, Bruce Trout, Jerry Hengemihle, Jeff D'Agostino, Seth Shulman, Stephen G. Ungar, Thomas Brakke, Darrell Boyer, Jim Van Gaasbeck, Ronald Greeley, Thomas Doggett, Victor R. Baker, James M. Dohm & Felipe Ip (2004): *The EO-1 Autonomous Science Agent*. In: *3rd International Joint Conference on Autonomous Agents and Multiagent Systems*, IEEE Computer Society, pp. 420–427, doi:10.1109/AAMAS.2004.10022.

[7] Marta Cialdea Mayer, Andrea Orlandini & Alessandro Umbrico (2016): *Planning and execution with flexible timelines: a formal account*. *Acta Informatica* 53(6-8), pp. 649–680, doi:10.1007/s00236-015-0252-z.

[8] Dario Della Monica, Nicola Gigante, Salvatore La Torre & Angelo Montanari (2020): *Complexity of Qualitative Timeline-Based Planning*. In Emilio Muñoz-Velasco, Ana Ozaki & Martin Theobald, editors: *27th International Symposium on Temporal Representation and Reasoning, TIME 2020, September 23-25, 2020, Bozen-Bolzano, Italy*, *LIPIcs* 178, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 16:1–16:13, doi:10.4230/LIPICS.TIME.2020.16.

[9] Maria Fox & Derek Long (2003): *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. *J. Artif. Intell. Res.* 20, pp. 61–124, doi:10.1613/jair.1129.

[10] Simone Fratini, Amedeo Cesta, Andrea Orlandini, Riccardo Rasconi & Riccardo De Benedictis (2011): *APSI-based Deliberation in Goal Oriented Autonomous Controllers*. In: *ASTRA 2011*, 11, ESA.

[11] Nicola Gigante, Angelo Montanari, Marta Cialdea Mayer & Andrea Orlandini (2016): *Timelines Are Expressive Enough to Capture Action-Based Temporal Planning*. In Curtis E. Dyreson, Michael R. Hansen & Luke Hunsberger, editors: *23rd International Symposium on Temporal Representation and Reasoning*,, IEEE Computer Society, pp. 100–109, doi:10.1109/TIME.2016.18.

[12] Nicola Gigante, Angelo Montanari, Marta Cialdea Mayer & Andrea Orlandini (2017): *Complexity of Timeline-Based Planning*. In Laura Barbulescu, Jeremy Frank, Mausam & Stephen F. Smith, editors: *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017*, AAAI Press, pp. 116–124. Available at https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/view/15758.

[13] Nicola Gigante, Angelo Montanari, Andrea Orlandini, Marta Cialdea Mayer & Mark Reynolds (2020): *On timeline-based games and their complexity*. *Theoretical Computer Science* 815, pp. 247–269, doi:10.1016/j.tcs.2020.02.011.

[14] Dario Della Monica, Nicola Gigante, Angelo Montanari & Pietro Sala (2018): *A Novel Automata-Theoretic Approach to Timeline-Based Planning*. In Michael Thielscher, Francesca Toni & Frank Wolter, editors: *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, AAAI Press, pp. 541–550. Available at https://aaai.org/ocs/index.php/KR/KR18/paper/view/18024.

[15] Dario Della Monica, Nicola Gigante, Angelo Montanari, Pietro Sala & Guido Sciavicco (2017): *Bounded Timed Propositional Temporal Logic with Past Captures Timeline-based Planning with Bounded Constraints*. In Carles Sierra, editor: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, ijcai.org, pp. 1008–1014, doi:10.24963/IJCAI.2017/140.

[16] Nicola Muscettola (1994): *HSTS: Integrating Planning and Scheduling*. In Monte Zweben & Mark S. Fox, editors: *Intelligent Scheduling*, chapter 6, Morgan Kaufmann, pp. 169–212.

[17] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of an Asynchronous Reactive Module*. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini & Simona Ronchi Della Rocca, editors: *16th International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science* 372, Springer, pp. 652–671, doi:10.1007/BFB0035790.

[18] Alessandro Umbrico, Amedeo Cesta, Marta Cialdea Mayer & Andrea Orlandini (2017): *PLATINUm: A New Framework for Planning and Acting*. In Floriana Esposito, Roberto Basili, Stefano Ferilli & Francesca A. Lisi, editors: *Proceedings of the 16th International Conference of the Italian Association for Artificial Intelligence*, *LNCS* 10640, Springer, pp. 498–512, doi:10.1007/978-3-319-70169-1_37.

[19] Alessandro Umbrico, Amedeo Cesta & Andrea Orlandini (2023): *Human-Aware Goal-Oriented Autonomy through ROS-Integrated Timeline-based Planning and Execution*. In: *32nd IEEE International Conference on Robot and Human Interactive Communication*, IEEE, pp. 1164–1169, doi:10.1109/RO-MAN57019.2023.10309516.

Jumping Automata Must Pay*

Shaull Almagor
Technion, Israel
shaull@technion.ac.il †

Ishai Salgado
Technion, Israel
ishaisalgado@campus.technion.ac.il

Jumping automata are finite automata that read their input in a non-sequential manner, by allowing a reading head to "jump" between positions on the input, consuming a permutation of the input word. We argue that allowing the head to jump should incur some cost. To this end, we propose three quantitative semantics for jumping automata, whereby the jumps of the head in an accepting run define the cost of the run. The three semantics correspond to different interpretations of jumps: the *absolute distance* semantics counts the distance the head jumps, the *reversal* semantics counts the number of times the head changes direction, and the *Hamming distance* measures the number of letter-swaps the run makes.

We study these measures, with the main focus being the *boundedness problem*: given a jumping automaton, decide whether its (quantitative) languages is bounded by some given number k. We establish the decidability and complexity for this problem under several variants.

1 Introduction

Traditional automata read their input sequentially. Indeed, this is the case for most state-based computational models. In some settings, however, we wish to abstract away the order of the input letters. For example, when the input represents available resources, and we only wish to reason about their *quantity*. From a more language-theoretic perspective, this amounts to looking at the *commutative closure* of languages, a.k.a. their *Parikh image*. To capture this notion in a computation model, *Jumping Automata (*JFAs*)* were introduced in [19]. A jumping automaton may read its input in a non-sequential manner, jumping from letter to letter, as long as every letter is read exactly once. Several works have studied the algorithmic properties and expressive power of these automata [11, 12, 21, 10, 17, 4].

While JFAs are an attractive and simple model, they present a shortcoming when thought of as model for systems, namely that the abstraction of the order may be too coarse. More precisely, the movement of the head can be thought of as a physical process of accessing the input storage of the JFA. Then, sequential access is the most basic form of access and can be considered "cheap", but allowing the head to jump around is physically more difficult and therefore should incur a cost.

To address this, we present three *quantitative semantics* for JFAs, whereby a JFA represents a function from words to costs, which captures how expensive it is to accept a given word with respect to the head jumps. The different semantics capture different aspects of the cost of jumps, as follows.

Consider a JFA $\mathscr{A}$ and a word w, and let ρ be an accepting run of $\mathscr{A}$ on w. The run ρ specifies the sequence of states and indices visited in w. We first define the cost of individual runs.

- In the *Absolute Distance* semantics (ABS), the cost of ρ is the sum of the lengths of jumps it makes.
- In the *Reversal* semantics (REV), the cost of ρ is the number of times the reading head changes its direction (i.e., moving from right to left or from left to right).

*The full version can be found at the authors' homepages.

†This research was supported by the ISRAEL SCIENCE FOUNDATION (grant No. 989/22). We also thank Michaël Cadilhac for fruitful discussions about this submission.

A. Achilleos and A. Francalanza (Eds.): Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 19–34, doi:10.4204/EPTCS.409.6

© S. Almagor & I. Salgado
This work is licensed under the Creative Commons Attribution License.

- In the *Hamming* semantics (HAM), we consider the word w' induced by ρ, i.e., the order of letters that ρ reads. Then, the cost of ρ is the number of letters where w' differs from w.

We then define the cost of the word w in $\mathcal{A}$ according to each semantics, by taking the run that minimizes the cost.

Thus, we lift JFAs from a Boolean automata model to the rich setting of *quantitative* models [5, 3, 8, 9]. Unlike other quantitative automata, however, the semantics in this setting arise naturally from the model, without an external domain. Moreover, the definitions are naturally motivated by different types of memory access, as we now demonstrate. First, consider a system whose memory is laid out in an array (i.e., a tape), with a reading head that can move along the tape. Moving the head requires some energy, and therefore the total energy spent reading the input corresponds to the ABS semantics. Next, consider a system whose memory is a spinning disk (or a sliding tape), so that the head stays in place and the movement is of the memory medium. Then, it is cheap to continue spinning in the same direction[1], and the main cost is in reversing the direction, which requires stopping and reversing a motor. Then, the REV semantics best captures the cost. Finally, consider a system that reads its input sequentially, but is allowed to edit its input by replacing one letter with another, such that at the end the obtained word is a permutation of the original word. This is akin to *edit-distance automata* [20, 13] under a restriction of maintaining the amount of resources. Then, the minimal edits required correspond to the HAM semantics.

Example 1. *Consider a (standard)* NFA $\mathcal{A}$ *for the language given by the regular expression* $(ab)^*$ *(the concrete automaton chosen is irrelevant, see Remark 6). As a* JFA*,* $\mathcal{A}$ *accepts a word* w *if and only if* w *has an equal number of* a*'s and* b*'s. To illustrate the different semantics, consider the words* $w_1 = ababbaab$ *and* $w_2 = ababbaba$*, obtained from* $(ab)^4$ *by flipping the third* ab *pair (*w_1*) and the third and fourth pairs (*w_2*). As we define in Section 3, we think of runs of the* JFA $\mathcal{A}$ *as if the input is given between end markers at indices* 0 *and* $n+1$*, and the jumping run must start at* 0 *and end in* $n+1$*.*

- *In the* ABS *semantics, the cost of* w_1*, denoted* $\mathcal{A}_{\text{ABS}}(w_1)$*, is* 2*: the indices read by the head are* $0,1,2,3,4,6,5,7,8,9$*, so there are two jumps of cost* 1*: from* 4 *to* 6 *and from* 5 *to* 7*. Similarly, we have* $\mathcal{A}_{\text{ABS}}(w_2) = 4$*, e.g., by the sequence* $0,1,2,3,4,6,5,8,7,9$*, which has two jumps of cost* 1 *(*4 *to* 6 *and* 7 *to* 9*), and one jump of cost* 2 *(*5 *to* 8*). (formally, we need to prove that there is no better run, but this is not hard to see).*
- *In the* REV *semantics, we have* $\mathcal{A}_{\text{REV}}(w_1) = 2$ *by the same sequence of indices as above, as the head performs two "turns", one at index* 6 *(from* $\rightarrow$ *to* $\leftarrow$*) and one at* 5 *(from* $\leftarrow$ *to* $\rightarrow$*). Here, however, we also have* $\mathcal{A}_{\text{REV}}(w_2) = 2$*, using the sequence* $0,1,2,3,4,6,8,7,5,9$*, whose turning points are* 8 *and* 5*.*
- *In the* HAM *semantics we have* $\mathcal{A}_{\text{HAM}}(w_1) = 2$ *and* $\mathcal{A}_{\text{HAM}}(w_2) = 4$*, since we must change the letters in all the flipped pairs for the words to be accepted.*

Example 2. *Consider now an* NFA $\mathcal{B}$ *for the language given by the regular expression* a^*b^**. Note that as a* JFA*,* $\mathcal{B}$ *accepts* $\{a,b\}^*$*, since every word can be reordered to the form* a^*b^**.*

Observe that in the REV *semantics, for every word* w *we have* $\mathcal{B}_{\text{REV}}(w) \leq 2$*, since at the worst case* $\mathcal{B}$ *makes one left-to-right pass to read all the* a*'s, then a right-to-left pass to read all the* b*'s, and then jump to the right end marker, and thus it has two turning points. In particular,* $\mathcal{B}_{\text{REV}}$ *is bounded.*

However, in the ABS *and* HAM *semantics, the costs can become unbounded. Indeed, in order to accept words of the form* $b^n a^n$*, in the* ABS *semantics the head must first jump over all the* b^n*, incurring a high cost, and for the* HAM *semantics, all the letters must be changed, again incurring a high cost.*

[1] We assume that the head does not return back to the start by continuing the spin, but rather reaches some end.

Related work Jumping automata were introduced in [19]. We remark that [19] contains some erroneous proofs (e.g., closure under intersection and complement, also pointed out in [12]). The works in [11, 12] establish several expressiveness results on jumping automata, as well as some complexity results. In [21] many additional closure properties are established. An extension of jumping automata with a two-way tape was studied in [10], and jumping automata over infinite words were studied by the first author in [4].

When viewed as the commutative image of a language, jumping automata are closely related to Parikh Automata [16, 7, 6, 14], which read their input and accept if a certain Parikh image relating to the run belongs to a given semilinear set (indeed, we utilize the latter in our proofs). Another related model is that of symmetric transducers – automata equipped with outputs, such that permutations in the input correspond to permutations in the output. These were studied in [2] in a jumping-flavour, and in [1] in a quantitative k-window flavour.

More broadly, quantitative semantics have received much attention in the past two decades, with many motivations and different flavors of technicalities. We refer the reader to [5, 9] and the references therein.

Contribution and paper organization Our contribution consists of the introduction of the three jumping semantics, and the study of decision problems pertaining to them (defined in Section 3). Our main focus is the boundedness problem: given a JFA $\mathcal{A}$, decide whether the function described by it under each of the semantics bounded by some constant k. We establish the decidability of this problem for all the semantics, and consider the complexity of some fragments. More precisely, we consider several variants of boundedness, depending on whether the bound is a fixed constant, or an input to the problem, and on whether the jumping language of $\mathcal{A}$ is universal. Our complexity results are summarized in Table 1.

Our paper is organized as follows: the preliminaries and definitions are given in Sections 2 and 3. Then, each of Sections 4 to 6 studies one of the semantics, and follows the same structure: we initially establish that the membership problem for the semantics is NP-complete. Then we characterize the set of words whose cost is at most k using a construction of an NFA. These constructions differ according to the semantics, and involve some nice tricks with automata, but are technically not hard to understand. We note that these constructions are preceded by crucial observations regarding the semantics, which allow us to establish their correctness. Next, in Section 7 we give a complete picture of the interplay between the different semantics (using some of the results established beforehand). Finally, in Section 8 we discuss some exciting open problems. Due to lack of space, some proofs appear in the full version.

	k-BND	PARAM-BND	UNIV-k-BND	UNIV-PARAM-BND	
				unary	binary
ABS	Decidable PSPACE-h	Decidable PSPACE-h	PSPACE-c	EXPSPACE PSPACE-h	2-EXPSPACE PSPACE-h
REV	Decidable PSPACE-h	Decidable PSPACE-h	PSPACE-c	EXPSPACE PSPACE-h	2-EXPSPACE PSPACE-h
HAM	Decidable PSPACE-h	Decidable PSPACE-h	PSPACE-c	PSPACE-c	EXPSPACE PSPACE-h

Table 1: Complexity results of the various boundedness problems for the three semantics. The complexity of membership is NP-complete for all the semantics. The “Decidable” entries depend on the complexity of the containment problem for Parikh Automata.

2 Preliminaries and Definitions

For a finite alphabet Σ we denote by Σ^* the set of finite words over Σ. For $w \in \Sigma^*$ we denote its letters by $w = w_1 \cdots w_n$, and its length by $|w| = n$. In the following, when discussing sets of numbers, we define $\min \emptyset = \infty$

Automata A *nondeterministic finite automaton (*NFA*)* is a 5-tuple $\mathscr{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$ where Σ is a finite alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a nondeterministic transition function, $Q_0 \subseteq Q$ is a set of initial states, and $\alpha \subseteq Q$ is a set of accepting states. A *run* of $\mathscr{A}$ on a word $w = w_1 w_2 \dots w_n$ is a sequence $\rho = q_0, q_1, \dots, q_n$ such that $q_0 \in Q_0$ and for every $0 \le i < n$ it holds that $q_{i+1} \in \delta(q_i, w_{i+1})$. The run ρ is *accepting* if $q_n \in \alpha$. A word w is *accepted* by $\mathscr{A}$ if there exists an accepting run of $\mathscr{A}$ on w. The *language* of $\mathscr{A}$, denoted $\mathfrak{L}(\mathscr{A})$, is the set of words accepted by $\mathscr{A}$.

Permutations Let $n \in \mathbb{N}$. The *permutation group* S_n is the set of bijections (i.e. *permutations*) from $\{1,...,n\}$ to itself. S_n forms a group with the function-composition operation and the identity permutation as a neutral element. Given a word $w = w_1 \cdots w_n$ and a permutation $\pi \in S_n$, we define $\pi(w) = w_{\pi(1)} \cdots w_{\pi(n)}$. For example, if $w = abcd$ and $\pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix}$ then $\pi(w) = cdba$. We usually denote permutations in *one-line form*, e.g., π is represented as $(3,4,2,1)$. We say that a word y is a *permutation* of x, and we write $x \sim y$ if there exists a permutation $\pi \in S_{|x|}$ such that $\pi(x) = y$.

Jumping Automata A jumping automaton is syntactically identical to an NFA, with the semantic difference that it has a reading head that can "jump" between indices of the input word. An equivalent view is that a jumping automaton reads a (nondeterministically chosen) permutation of the input word.

Formally, consider an NFA $\mathscr{A}$. We view $\mathscr{A}$ as a *jumping finite automaton* (JFA) by defining its *jumping language* $\mathfrak{J}(\mathscr{A}) = \{w \in \Sigma^* \mid \exists u \in \Sigma^*.\ w \sim u \wedge u \in \mathfrak{L}(\mathscr{A})\}$.

Since our aim is to reason about the manner with which the head of a JFA jumps, we introduce a notion to track the head along a run. Consider a word w of length n and a JFA $\mathscr{A}$. A *jump sequence* is a vector $\mathbf{a} = (a_0, a_1, a_2, \dots, a_n, a_{n+1})$ where $a_0 = 0$, $a_{n+1} = n+1$ and $(a_1, a_2, \dots, a_n) \in S_n$. We denote by J_n the set of all jump sequences of size $n+2$.

Intuitively, a jump sequence $\mathbf{a}$ represents the order in which a JFA visits a given word of length n. First it visits the letter at index a_1, then the letter at index a_2 and so on. To capture this, we define $w_{\mathbf{a}} = w_{a_1} w_{a_2} \cdots w_{a_n}$. Observe that jump sequences enforce that the head starts at position 0 and ends at position $n+1$, which can be thought of as left and right markers, as is common in e.g., two-way automata.

An alternative view of jumping automata is via *Parikh Automata (PA)* [16, 6]. The standard definition of PA is an automaton whose acceptance condition includes a semilinear set over the transitions. To simplify things, and to avoid defining unnecessary concepts (e.g., semilinear sets), for our purposes, a PA is a pair $(\mathscr{A}, \mathscr{C})$ where $\mathscr{A}$ is an NFA over alphabet Σ, and $\mathscr{C}$ is a JFA over Σ. Then, the PA $(\mathscr{A}, \mathscr{C})$ accepts a word w if $w \in \mathfrak{L}(\mathscr{A}) \cap \mathfrak{J}(\mathscr{C})$. Note that when $\mathfrak{L}(\mathscr{A}) = \Sigma^*$, then the PA coincides with $\mathfrak{J}(\mathscr{C})$. Our usage of PA is to obtain the decidability of certain problems. Specifically, from [16] we have that emptiness of PA is decidable.

3 Quantitative Semantics for JFAs

In this section we present and demonstrate the three quantitative semantics for JFAs. We then define the relevant decision problems, and lay down some general outlines to solving them, which are used in later sections. For the remainder of the section fix a JFA $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$.

3.1 The Semantics

The Absolute-Distance Semantics In the absolute-distance semantics, the cost of a run (given as a jump sequence) is the sum of the sizes of the jumps made by the head. Since we want to think of a sequential run as a run with 0 jumps, we measure a jump over k letters as distance $k-1$ (either to the left or to the right). This is captured as follows.

For $k \in \mathbb{Z}$, define $[\![k]\!] = |k|-1$. Consider a word $w \in \Sigma^*$ with $|w| = n$, and let $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_n, a_{n+1})$ be a jump sequence, then we lift the notation above and write $[\![\mathbf{a}]\!] = \sum_{i=1}^{n+1} [\![a_i - a_{i-1}]\!]$.

Definition 3 (Absolute-Distance Semantics)**.** *For a word $w \in \Sigma^*$ with $|w| = n$ we define*

$$\mathcal{A}_{\text{ABS}}(w) = \min\{[\![\mathbf{a}]\!] \mid \mathbf{a} \textit{ is a jump sequence, and } w_{\mathbf{a}} \in \mathfrak{L}(\mathcal{A})\}$$

(recall that $\min \emptyset = \infty$ *by definition).*

The Reversal Semantics In the reversal semantics, the cost of a run is the number of times the head changes direction in the corresponding jump sequence. Consider a word $w \in \Sigma^*$ with $|w| = n$, and let $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_n, a_{n+1})$ be a jump sequence, we define

$$\#_{\text{REV}}(\mathbf{a}) = |\{i \in \{1, \ldots, n\} \mid (a_i > a_{i-1} \wedge a_i > a_{i+1}) \vee (a_i < a_{i-1} \wedge a_i < a_{i+1})\}|$$

Definition 4 (Reversal Semantics)**.** *For a word $w \in \Sigma^*$ with $|w| = n$ we define*

$$\mathcal{A}_{\text{REV}}(w) = \min\{\#_{\text{REV}}(\mathbf{a}) \mid \mathbf{a} \textit{ is a jump sequence, and } w_{\mathbf{a}} \in \mathfrak{L}(\mathcal{A})\}$$

The Hamming Semantics In the Hamming measure, the cost of a word is the minimal number of coordinates of w that need to be changed in order for the obtained word to be accepted by $\mathcal{A}$ (sequentially, as an NFA), so that the changed word is a permutation of w.

Consider two words $x, y \in \Sigma^*$ with $|x| = |y| = n$ such that $x \sim y$, we define the *Hamming Distance* between x and y as $d_H(x,y) = |\{i \mid x_i \neq y_i\}|$.

Definition 5 (Hamming Semantics)**.** *For a word $w \in \Sigma^*$ we define*

$$\mathcal{A}_{\text{HAM}}(w) = \min\{d_H(w', w) \mid w' \in L(\mathcal{A}), w' \sim w\}$$

Remark 6. *Note that the definitions of the three semantics are independent of the* NFA*, and only refer to its language. We can therefore refer to the cost of a word in a language according to each semantics, rather than the cost of a word in a concrete automaton.*

3.2 Quantitative Decision Problems

In the remainder of the paper we focus on quantitative variants of the standard Boolean decision problems pertaining to the jumping semantics. Specifically, we consider the following problems for each semantics $\textsc{sem} \in \{\textsc{abs}, \textsc{ham}, \textsc{rev}\}$.

- MEMBERSHIP: Given a JFA $\mathscr{A}$, $k \in \mathbb{N}$ and a word w, decide whether $\mathscr{A}_{\textsc{sem}}(w) \leq k$.
- k-BND (for a fixed k): Given a JFA $\mathscr{A}$, decide whether $\forall w \in \mathfrak{J}(\mathscr{A})\ \mathscr{A}_{\textsc{sem}}(w) \leq k$.
- PARAM-BND: Given a JFA $\mathscr{A}$ and $k \in \mathbb{N}$, decide whether $\forall w \in \mathfrak{J}(\mathscr{A})\ \mathscr{A}_{\textsc{sem}}(w) \leq k$.

We also pay special attention to the setting where $\mathfrak{J}(\mathscr{A}) = \Sigma^*$, in which case we refer to these problems as UNIV-k-BND and UNIV-PARAM-BND. For example, in UNIV-PARAM-BND we are given a JFA $\mathscr{A}$ and $k \in \mathbb{N}$ and the problem is to decide whether $\mathscr{A}_{\textsc{sem}}(w) \leq k$ for all words $w \in \Sigma^*$.

The boundedness problems can be thought of as quantitative variants of Boolean universality (i.e., is the language equal to Σ^*). Observe that the problems above are not fully specified, as the encoding of k (binary or unary) when it is part of the input may effect the complexity. We remark on this when it is relevant. Note that the emptiness problem is absent from the list above. Indeed, a natural quantitative variant would be: is there a word w such that $\mathscr{A}_{\textsc{sem}}(w) \leq k$. This, however, is identical to Boolean emptiness, since $\mathfrak{L}(\mathscr{A}) \neq \emptyset$ if and only if there exists w such that $\mathscr{A}_{\textsc{sem}}(w) = 0$. We therefore do not consider this problem. Another problem to consider is boundedness when k is existentially quantified. We elaborate on this problem in Section 8.

4 The Absolute-Distance Semantics

The first semantics we investigate is $\textsc{abs}$, and we start by showing that (the decision version of) computing its value for a given word is NP-complete. This is based on bounding the distance with which a word can be accepted.

Lemma 7. *Consider a JFA $\mathscr{A}$ and $w \in \mathfrak{J}(\mathscr{A})$ with $|w| = n$, then $\mathscr{A}_{\textsc{abs}}(w) < n^2$.*

Proof. Since $w \in \mathfrak{J}(\mathscr{A})$, there exists a jump sequence $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_n, a_{n+1})$ such that $w_{\mathbf{a}} \in \mathfrak{L}(\mathscr{A})$. Therefore, $\mathscr{A}_{\textsc{abs}}(w) \leq [\![\mathbf{a}]\!]$. Observe that $|a_i - a_{i-1}| \leq n$ for all $i \in \{1, \ldots, n+1\}$, since there is no jump from 0 to $n+1$ (since $a_0 = 0$ and $a_n = n+1$). The following concludes the proof:

$$[\![\mathbf{a}]\!] = \sum_{i=1}^{n+1} [\![a_i - a_{i-1}]\!] = \sum_{i=1}^{n+1} |a_i - a_{i-1}| - 1 \leq \sum_{i=1}^{n+1} n - 1 = (n+1)(n-1) < n^2$$

□

We can now prove the complexity bound for computing the absolute distance, as follows.

Theorem 8 (Absolute-Distance MEMBERSHIP is NP-complete)**.** *The problem of deciding, given $\mathscr{A}$, w and $k \in \mathbb{N}$, whether $\mathscr{A}_{\textsc{abs}}(w) \leq k$, is NP-complete.*

Proof. In order to establish membership in NP, note that by Lemma 7, we can assume $k \leq n^2$, as otherwise we can set $k = n^2$. Then, it is sufficient to nondeterministically guess a jump sequence $\mathbf{a}$ and to check that $w_{a_1} \cdots w_{a_n} \in \mathfrak{L}(\mathscr{A})$ and that $[\![\mathbf{a}]\!] \leq k$. Both conditions are easily checked in polynomial time, since k is polynomially bounded.

Hardness in NP follows by reduction from (Boolean) membership in JFA: it is shown in [12] that deciding whether $w \in \mathfrak{J}(\mathscr{A})$ is NP-hard. We reduce this problem by outputting, given $\mathscr{A}$ and w, the same $\mathscr{A}$ and w with the bound $k = n^2$. The reduction is correct by Lemma 7 and the fact that if $w \notin \mathfrak{J}(\mathscr{A})$ then $\mathscr{A}_{\textsc{abs}}(w) = \infty$. □

4.1 Decidability of Boundedness Problems for ABS

We now turn our attention to the boundedness problems. Consider a JFA $\mathcal{A}$ and $k \in \mathbb{N}$. Intuitively, our approach is to construct an NFA $\mathcal{B}$ that simulates, while reading a word $w \in \Sigma^*$, every jump sequence of $\mathcal{A}$ on w whose absolute distance is at most k. The crux of the proof is to show that we can indeed bound the size of $\mathcal{B}$ as a function of k. At a glance, the main idea here is to claim that since the absolute distance is bounded by k, then $\mathcal{A}$ cannot make large jumps, nor many small jumps. Then, if we track a sequential head going from left to right, then the jumping head must always be within a bounded distance from it. We now turn to the formal arguments. Fix a JFA $\mathcal{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$.

To understand the next lemma, imagine $\mathcal{A}$'s jumping head while taking the jth step in a run on w according to a jump sequence $\mathbf{a} = (a_0, a_1, a_2, \dots, a_n, a_{n+1})$. Thus, the jumping head points to the letter at index a_j. Concurrently, imagine a "sequential" head (reading from left to right), which points to the jth letter in w. Note that these two heads start and finish reading the word at the same indices $a_0 = 0$ and $a_{n+1} = n+1$. It stands to reason that if at any step while reading w the distance between these two heads is large, the cost of reading w according to $\mathbf{a}$ would also be large, as there would need to be jumps that bridge the gaps between the heads. The following lemma formalizes this idea.

Lemma 9. *Consider a jump sequence $\mathbf{a} = (a_0, a_1, a_2, \dots, a_n, a_{n+1})$. For every $1 \le j \le n$ it holds that $[\![\mathbf{a}]\!] \ge |a_j - j|$.*

Proof. Let $1 \le j \le n+1$. First, assume that $a_j \ge j$ and consider the sum $\sum_{i=1}^{j} [\![a_i - a_{i-1}]\!] \le [\![\mathbf{a}]\!]$. From the definition of $[\![\cdot]\!]$ we have $\sum_{i=1}^{j} [\![a_i - a_{i-1}]\!] = \left(\sum_{i=1}^{j} |a_i - a_{i-1}|\right) - j$, and we conclude that in this case $[\![\mathbf{a}]\!] \ge |a_j - j|$ by the following:

$$\left(\sum_{i=1}^{j} |a_i - a_{i-1}|\right) - j \underset{\text{triangle inequality}}{\ge} \left|\sum_{i=1}^{j} a_i - a_{i-1}\right| - j \underset{\text{telescopic sum}}{=} |a_j - a_0| - j \underset{a_0=0}{=} a_j - j \underset{a_j \ge j}{=} |a_j - j|$$

The direction $a_j < j$ is proved by looking at the sum of the *last* j elements: assume $a_j < j$, and consider the sum $\sum_{i=j+1}^{n+1} [\![a_i - a_{i-1}]\!] \le [\![\mathbf{a}]\!]$. From the definition of $[\![\cdot]\!]$ we have

$$\sum_{i=j+1}^{n+1} [\![a_i - a_{i-1}]\!] = \left(\sum_{i=j+1}^{n+1} |a_i - a_{i-1}|\right) - (n+1-(j+1)+1) = \left(\sum_{i=j+1}^{n+1} |a_i - a_{i-1}|\right) - (n+1-j)$$

Similarly to the previous case, from the triangle inequality we have

$$\left(\sum_{i=j+1}^{n+1} |a_i - a_{i-1}|\right) - (n+1-j) \ge |a_{n+1} - a_j| - (n+1-j) = n+1-a_j-(n+1-j) = j - a_j = |a_j - j|$$

where we use the fact that $a_{n+1} = n+1 > a_j$, and our assumption that $a_j < j$. This again concludes that $[\![\mathbf{a}]\!] \ge |a_j - j|$. □

From Lemma 9 we get that in order for a word w to attain a small cost, it must be accepted with a jumping sequence that stays close to the sequential head. More precisely:

Corollary 10. *Let $k \in \mathbb{N}$ and consider a word w such that $\mathcal{A}_{\text{ABS}}(w) \le k$, then there exists a jumping sequence $\mathbf{a} = (a_0, a_1, a_2, \dots, a_n, a_{n+1})$ such that $w_{\mathbf{a}} \in \mathfrak{L}(\mathcal{A})$ and for all $1 \le j \le n$ we have $|a_j - j| \le k$.*

We now turn to the construction of an NFA that recognizes the words whose cost is at most k.

Lemma 11. *Let $k \in \mathbb{N}$. We can effectively construct an* NFA $\mathscr{B}$ *such that* $\mathfrak{L}(\mathscr{B}) = \{w \in \Sigma^* | \mathscr{A}_{\text{ABS}}(w) \leq k\}$.

Proof. Let $k \in \mathbb{N}$. Intuitively, $\mathscr{B}$ works as follows: it remembers in its states a window of size $2k+1$ centered around the current letter (recall that as an NFA, it reads its input sequentially). The window is constructed by nondeterministically guessing (and then verifying) the next k letters, and remembering the last k letters.

$\mathscr{B}$ then nondeterministically simulates a jumping sequence of $\mathscr{A}$ on the given word, with the property that the jumping head stays within distance k from the sequential head. This is done by marking for each letter in the window whether it has already been read in the jumping sequence, and nondeterministically guessing the next letter to read, while keeping track of the current jumping head location, as well as the total cost incurred so far. After reading a letter, the window is shifted by one to the right. If at any point the window is shifted so that a letter that has not been read by the jumping head shifts out of the $2k+1$ scope, the run rejects. Similarly, if the word ends but the guessed run tried to read a letter beyond the length of the word, the run rejects. The correctness of the construction follows from Corollary 10. We now turn to the formal details. Recall that $\mathscr{A} = \langle \Sigma, Q, \delta, Q_0, \alpha \rangle$. We define $\mathscr{B} = \langle \Sigma, Q', \delta', Q'_0, \beta \rangle$ as follows.

The state space of $\mathscr{B}$ is $Q' = Q \times (\Sigma \times \{?, \checkmark\})^{-k,\dots,k} \times \{-k,\dots,k\} \times \{0,\dots,k\}$. We denote a state of $\mathscr{B}$ as (q, f, j, c) where $q \in Q$ is a state of $\mathscr{A}$, $f : \{-k,\dots,k\} \to \Sigma \times \{?, \checkmark\}$ represents a window of size $2k+1$ around the sequential head, where $\checkmark$ marks letters that have already been read by $\mathscr{A}$ (and ? marks the others), j represents the index of the head of $\mathscr{A}$ relative to the sequential head, and c represents the cost incurred thus far in the run. We refer to the components of f as $f(j) = (f(j)_1, f(j)_2)$ with $f(j)_1 \in \Sigma$ and $f(j)_2 \in \{?, \checkmark\}$.

The initial states of $\mathscr{B}$ are $Q'_0 = \{(q, f, j, j-1) \mid q \in Q_0 \wedge j > 0 \wedge (f(i)_2 = \checkmark \iff i \leq 0)\}$. That is, all states where the state of $\mathscr{A}$ is initial, the location of the jumping head is some $j > 0$ incurring a cost of $j-1$ (i.e., the initial jump $\mathscr{A}$ makes), and the window is guessed so that everything left of the first letter is marked as already-read (to simulate the fact that $\mathscr{A}$ cannot jump to the left of the first letter).

The transitions of $\mathscr{B}$ are defined as follows. Consider a state (q, f, j, c) and a letter $\sigma \in \Sigma$, then $(q', f', j', c') \in \delta'((q, f, j, c), \sigma)$ if and only if the following hold (see Fig. 1 for an illustration):

- $f(1)_1 = \sigma$. That is, we verify that the next letter in the guessed window is indeed correct.
- $f(-k)_2 = \checkmark$. That is, the leftmost letter has been read. Otherwise by Corollary 10 the cost of continuing the run must be greater than k.
- $f(j)_2 \neq \checkmark$ and $f'(j-1) = \checkmark$ (if $j > -k$). That is, the current letter has not been previously read, and will be read from now on (note that index j before the transition corresponds to index $j-1$ after).
- $q' = \delta(q, f(j)_1)$, i.e. the state of $\mathscr{A}$ is updated according to the current letter.
- $c' = c + |j' + 1 - j| - 1$, since j' represents the index in the shifted window, so in the "pre-shifted" tape this is actually index $j+1$. We demonstrate this in Fig. 1. Also, $c' \leq k$ by the definition of Q.
- $f'(i) = f(i+1)$ for $i < k$. That is, the window is shifted and the index $f'(k)$ is nondeterministically guessed[2].

Finally, the accepting states of $\mathscr{B}$ are $\beta = \{(q, f, 1, c) \mid q \in \alpha \wedge f(j)_2 =? \text{ for all } j > 0\}$. That is, the state of $\mathscr{A}$ is accepting, the overall cost is at most k, the location of the jumping head matches the sequential head (intuitively, location $n+1$), and no letter beyond the end of the tape has been used.

[2]The guess could potentially be $\checkmark$, but this is clearly useless.

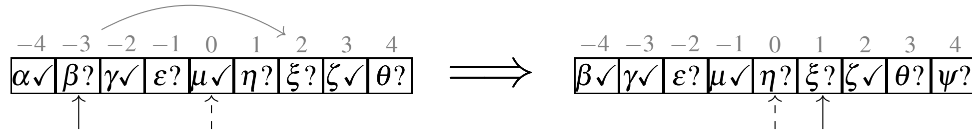


Figure 1: A single transition in the construction of Lemma 11. The dashed arrow signifies the sequential head, the full arrow is the "imaginary" jumping head. Here, the head jumps from -3 to 2, incurring a cost of 4, but in the indexing after the transition ξ is at index 1, thus the expression given for c' in the construction. Note that the letter being read must be μ, and that α must be checked, otherwise the run has failed.

It is easy to verify that $\mathscr{B}$ indeed guesses a jump sequence and a corresponding run of $\mathscr{A}$ on the given word, provided that the jumping head stays within distance k of the sequential head. By Corollary 10, this restriction is complete, in the sense that if $\mathscr{A}_{\text{ABS}}(w) \leq k$ then there is a suitable jump sequence under this restriction with which w is accepted. ☐

We can now readily conclude the decidability of the boundedness problems for the ABS semantics. The proof (in the full version) makes use of the decidability of emptiness for Parikh Automata [16].

Theorem 12. *The following problems are decidable for the* ABS *semantics: k*-BND*,* PARAM-BND*,* UNIV-k-BND *and* UNIV-PARAM-BND.

With further scrutiny, we see that the size of $\mathscr{B}$ constructed as per Lemma 11 is polynomial in the size of $\mathscr{A}$ and single-exponential in k. Thus, UNIV-k-BND is in fact decidable in PSPACE, whereas UNIV-PARAM-BND is in EXPSPACE and 2-EXPSPACE for k given in unary and binary, respectively. For the non-universal problems we do not supply upper complexity bounds, as these depend on the decidability for PA containment, for which we only derive decidability from [16].

4.2 PSPACE-Hardness of Boundedness for ABS

In the following, we complement the decidability result of Theorem 12 by showing that already UNIV-k-BND is PSPACE-hard, for every $k \in \mathbb{N}$.

We first observe that the absolute distance of every word is even. In fact, this is true for every jumping sequence.

Lemma 13. *Consider a jumping sequence* $\mathbf{a} = (a_0, a_1, \ldots, a_n, a_{n+1})$*, then* $[\![\mathbf{a}]\!]$ *is even.*

Proof. Observe that the parity of $|a_i - a_{i-1}|$ is the same as that of $a_i - a_{i-1}$. It follows that the parity of $[\![\mathbf{a}]\!] = \sum_{i=1}^{n+1} [\![a_i - a_{i-1}]\!] = \sum_{i=1}^{n+1} |a_i - a_{i-1}| - 1$ is the same as that of

$$\sum_{i=1}^{n+1} (a_i - a_{i-1} - 1) = \left(\sum_{i=1}^{n+1} a_i - a_{i-1} \right) - (n+1) = n+1-(n+1) = 0$$

and is therefore even (the penultimate equality is due to the telescopic sum). ☐

We say that $\mathscr{A}_{\text{ABS}}$ is *k-bounded* if $\mathscr{A}_{\text{ABS}}(w) \leq k$ for all $w \in \Sigma^*$. We are now ready to prove the hardness of UNIV-k-BND. Observe that for a word $w \in \Sigma^*$ we have that $\mathscr{A}_{\text{ABS}}(w) = 0$ if and only if $w \in \mathfrak{L}(\mathscr{A})$ (indeed, a cost of 0 implies that an accepting jump sequence is the sequential run $0, 1, \ldots, |w|+1$). In particular, we have that $\mathscr{A}_{\text{ABS}}$ is 0-bounded if and only if $\mathfrak{L}(\mathscr{A}) = \Sigma^*$. Since the universality problem

for NFAs is PSPACE-complete, this readily proves that UNIV-0-BND is PSPACE-hard. Note, however, that this does *not* imply that UNIV-k-BND is also PSPACE-hard for other values of k, and that the same argument fails for $k > 0$. We therefore need a slightly more elaborate reduction.

Lemma 14. *For* ABS *the* UNIV-k-BND *and* k-BND *problems are PSPACE-hard for every* $k \in \mathbb{N}$.

Proof. We sketch the proof for UNIV-k-BND. The case of k-BND requires slightly more effort and is delegated to the full version. By Lemma 13, we can assume without loss of generality that k is even. Indeed, if there exists $m \in \mathbb{N}$ such that $\mathscr{A}_{\text{ABS}}(w) \leq 2m+1$ for every $w \in \Sigma^*$, then by Lemma 13 we also have $\mathscr{A}_{\text{ABS}}(w) \leq 2m$. Therefore, we assume $k = 2m$ for some $m \in \mathbb{N}$.

We reduce the universality problem for NFAs to the UNIV-$2m$-BND problem. Consider an NFA $\mathscr{A} = \langle Q, \Sigma, \delta, Q_0, \alpha \rangle$, and let $\heartsuit \notin \Sigma$ be a fresh symbol. Intuitively, we obtain from $\mathscr{A}$ an NFA $\mathscr{B}$ over the alphabet $\Sigma \cup \{\heartsuit\}$ such that $w \in \mathfrak{L}(\mathscr{B})$ if and only if the following hold:

1. Either w does not contain exactly m occurrences of $\heartsuit$, or
2. w contains exactly m occurrences of $\heartsuit$, but does not start with $\heartsuit$, and $w|_\Sigma \in \mathfrak{L}(\mathscr{A})$ (where $w|_\Sigma$ is obtained from w by removing all occurrences of $\heartsuit$).

We then have the following: if $\mathfrak{L}(\mathscr{A}) = \Sigma^*$, then for every $w \in (\Sigma \cup \{\heartsuit\})^*$ if $w \in \mathfrak{L}(\mathscr{B})$ then $\mathscr{B}_{\text{ABS}}(w) = 0 \leq 2m$, and if $w \notin \mathfrak{L}(\mathscr{B})$ then w starts with $\heartsuit$ but has exactly m occurrences of $\heartsuit$. Thus, jumping to the first occurrence of a letter in Σ incurs a cost of at most m, and reading the skipped $\heartsuit$ symbols raises the cost to at most $2m$. From there, w can be read consecutively and be accepted since $w|_\Sigma \in \mathfrak{L}(\mathscr{A})$. So again $\mathscr{B}_{\text{ABS}}(w) \leq 2m$, and $\mathscr{B}$ is $2m$-bounded.

Conversely, if $\mathfrak{L}(\mathscr{A}) \neq \Sigma^*$, take $x \notin \mathfrak{L}(\mathscr{A})$ such that $x \neq \varepsilon$ (see the full version for details regarding this assumption), and consider the word $w = \heartsuit^m x$. We then have $w \notin \mathfrak{L}(\mathscr{B})$, and moreover – in order to accept w (if at all possible), $\mathscr{B}$ first needs to jump over the initial $\heartsuit^m$, guaranteeing a cost of at least $2m$ (m for the jump and another m to later read the $\heartsuit^m$ prefix), and needs at least one more jump to accept x, since $x \notin \mathfrak{L}(\mathscr{A})$. Thus, $\mathscr{B}_{\text{ABS}}(w) > 2m$, so $\mathscr{B}$ is not $2m$-bounded. The precise construction and correctness are given in the full version. ☐

Lemma 14 shows hardness for fixed k, and in particular when k is part of the input. Thus, UNIV-PARAM-BND and PARAM-BND are also PSPACE-hard, and UNIV-k-BND is PSPACE-complete. Also, UNIV-PARAM-BND is in EXPSPACE and 2-EXPSPACE for k given in unary and binary, respectively.

5 The Reversal Semantics

We now study the reversal semantics. Recall from Definition 4 that for a JFA $\mathscr{A}$ and a word w, the cost $\mathscr{A}_{\text{REV}}(w)$ is the minimal number of times the jumping head changes "direction" in a jump sequence for which w is accepted.

Consider a word w with $|w| = n$ and a jump sequence $\mathbf{a} = (a_0, a_1, a_2, \ldots, a_n, a_{n+1})$. We say that an index $1 \leq i \leq n$ is a *turning index* if $a_i > a_{i-1}$ and $a_i > a_{i+1}$ (i.e., a right-to-left turn) or if $a_i < a_{i-1}$ and $a_i < a_{i+1}$ (i.e., a left-to-right turn). We denote by $\texttt{Turn}(\mathbf{a})$ the set of turning indices of $\mathbf{a}$.

For example, consider the jump sequence $(\overset{a_0}{0}, \overset{a_1}{2}, \overset{a_2}{3}, \overset{a_3}{5}, \overset{a_4}{7}, \overset{a_5}{4}, \overset{a_6}{1}, \overset{a_7}{6}, \overset{a_8}{8})$, then $\texttt{Turn}(\mathbf{a}) = \{4, 6\}$. Note that the cost of w is then $\mathscr{A}_{\text{REV}}(w) = \min\{|\texttt{Turn}(\mathbf{a})| \mid w_{\mathbf{a}} \in \mathfrak{L}(\mathscr{A})\}$. Viewed in this manner, we have that $\mathscr{A}_{\text{REV}}(w) \leq |w|$, and computing $\texttt{Turn}(\mathbf{a})$ can be done in polynomial time. Thus, analogously to Theorem 8 we have the following.

Theorem 15 (Reversal MEMBERSHIP is NP-complete)**.** *The problem of deciding, given $\mathscr{A}$ and k, whether $\mathscr{A}_{\text{REV}}(w) \leq k$ is NP-complete.*

Remark 16. *For every jump sequence* **a** *we have that* $|\texttt{Turn}(\mathbf{a})|$ *is even, since the head starts at position* 0 *and ends at* $n+1$*, where after an odd number of turning points the direction is right-to-left, and hence cannot reach* $n+1$.

5.1 Decidability of Boundedness Problems for REV

We begin by characterizing the words accepted using at most k reversals as a shuffle of subwords and reversed-subwords, as follows. Let $x, y \in \Sigma^*$ be words, we define their *shuffle* to be the set of words obtained by interleaving parts of x and parts of y. Formally:

$$x \sqcup\!\sqcup y = \{s_1 \cdot t_1 \cdot s_2 \cdot t_2 \cdots s_k \cdot t_k \mid \forall i\ s_i, t_i \in \Sigma^* \wedge x = s_1 \cdots s_k \wedge y = t_1 \cdots t_k\}$$

For example, if $x = aab$ and $y = cd$ then $x \sqcup\!\sqcup y$ contains the words $aabcd$, $acabd$, $caadb$, among others (the colors reflect which word each subword originated from). Note that the subwords may be empty, e.g., $caadb$ can be seen as starting with ε as a subword of x. It is easy to see that $\sqcup\!\sqcup$ is an associative operation, so it can be extended to any finite number of words.

The following lemma states that, intuitively, if $\mathscr{A}_{\text{REV}}(w) \leq k$, then w can be decomposed to a shuffle of at most $k+1$ subwords of itself, where all the even ones are reversed (representing the left-reading subwords). See the full version for the proof.

Lemma 17. *Let $k \in \mathbb{N}$. Consider an* NFA $\mathscr{A}$ *and a word $w \in \Sigma^*$. Then $\mathscr{A}_{\text{REV}}(w) \leq k$ if and only if there exist words $s_1, s_2, \ldots, s_{k+1} \in \Sigma^*$ such that the following hold.*

1. $s_1 s_2 \ldots s_{k+1} \in \mathfrak{L}(\mathscr{A})$.
2. $w \in s_1 \sqcup\!\sqcup {s_2}^R \sqcup\!\sqcup s_3 \sqcup\!\sqcup {s_4}^R \sqcup\!\sqcup \ldots \sqcup\!\sqcup s_{k+1}$ *(where s_i^R is the reverse of s_i).*

Using the characterization in Lemma 17, we can now construct a corresponding NFA, by intuitively guessing the shuffle decomposition and running copies of $\mathscr{A}$ and its reverse in parallel. See the full version for the proof.

Lemma 18. *Let $k \in \mathbb{N}$ and consider a* JFA $\mathscr{A}$*. We can effectively construct an* NFA $\mathscr{B}$ *such that* $\mathfrak{L}(\mathscr{B}) = \{w \in \Sigma^* \mid \mathscr{A}_{\text{REV}}(w) \leq k\}$.

The proof of Lemma 18 shows that the size of $\mathscr{B}$ is polynomial in the size of $\mathscr{A}$ and single-exponential in k, giving us PSPACE membership for UNIV-k-BND.

5.2 PSPACE-Hardness of Boundedness for REV

Following a similar scheme to the Absolute Distance Semantics of Section 4, observe that for a word $w \in \Sigma^*$ we have that $\mathscr{A}_{\text{ABS}}(w) = 0$ if and only if $w \in \mathfrak{L}(\mathscr{A})$, which implies that UNIV-0-BND is PSPACE-hard. Yet again, the challenge is to prove hardness of UNIV-k-BND for all values of k.

Theorem 19. *For* REV, UNIV-k-BND *is PSPACE-complete for every $k \in \mathbb{N}$.*

Proof. Membership in PSPACE follows from Lemma 18 and the discussion thereafter. For hardness, we follow the same flow as the proof of Lemma 14, but naturally the reduction itself is different. Specifically, we construct an NFA that must read an expression of the form $(\heartsuit\spadesuit)^m$ before its input. This allows us to shuffle the input to the form $\spadesuit^m\heartsuit^m$, which causes many reversals (see the full version). ☐

As in Section 4.2, it follows that UNIV-PARAM-BND, k-BND and PARAM-BND are also PSPACE-hard.

6 The Hamming Semantics

Recall from Definition 5 that for a JFA $\mathscr{A}$ and word w, the cost $\mathscr{A}_{\text{HAM}}(w)$ is the minimal Hamming distance between w and w' where $w' \sim w$ and $w' \in \mathfrak{L}(\mathscr{A})$.

Remark 20 (An alternative interpretation of the Hamming Semantics)**.** *We can think of a jumping automaton as accepting a permutation w' of the input word w. As such, a natural candidate for a quantitative measure is the "distance" of the permutation used to obtain w' from the identity (i.e. from w). The standard definition for such a distance is the number of transpositions of two indices required to move from one permutation to the other, namely the distance in the Cayley graph for the transpositions generators of S_n. It is easy to show that in fact, the Hamming distance coincides with this definition.*

In the full version we establish the complexity of computing the Hamming measure of a given word.

Theorem 21 (Hamming MEMBERSHIP is NP-complete)**.** *The problem of deciding, given $\mathscr{A}$ and $k \in \mathbb{N}$, whether $\mathscr{A}_{\text{HAM}}(w) \leq k$ is NP-complete.*

Similarly to Sections 4.1 and 5.1, in order to establish the decidability of UNIV-PARAM-BND, we start by constructing an NFA that accepts exactly the words for which $\mathscr{A}_{\text{HAM}}(w) \leq k$.

Lemma 22. *Let $k \in \mathbb{N}$. We can effectively construct an* NFA $\mathscr{B}$ *with* $\mathfrak{L}(\mathscr{B}) = \{w \in \Sigma^* \mid \mathscr{A}_{\text{HAM}}(w) \leq k\}$.

Proof. Let $k \in \mathbb{N}$. Intuitively, $\mathscr{B}$ works as follows: while reading a word w sequentially, it simulates the run of $\mathscr{A}$, but allows $\mathscr{A}$ to intuitively "swap" the current letter with a (nondeterministically chosen) different one (e.g., the current letter may be a but the run of $\mathscr{A}$ can be simulated on either a or b). Then, $\mathscr{B}$ keeps track of the swaps made by counting for each letter a how many times it was swapped by another letter, and how many times another letter was swapped to it. This is done by keeping a counter ranging from $-k$ to k, counting the difference between the number of occurrences of each letter in the simulated word versus the actual word. We refer to this value as the *balance* of the letter. $\mathscr{B}$ also keeps track of the total number of swaps. Then, a run is accepting if at the end of the simulation, the total amount of swaps does not exceed k, and if all the letters end up with 0 balance. See the full version for a detailed construction and proof. ☐

An analogous proof to Theorem 12 gives us the following.

Theorem 23. *The following problems are decidable for the* HAM *semantics: k-*BND*,* PARAM-BND*,* UNIV-k-BND*, and* UNIV-PARAM-BND.

We note that the size of $\mathscr{B}$ constructed in Lemma 22 is polynomial in k and single-exponential in $|\Sigma|$, and therefore when Σ is fixed and k is either fixed or given in unary, both UNIV-PARAM-BND and UNIV-k-BND are in PSPACE.

For a lower bound, we remark that similarly to Section 4.2, it is not hard to prove that UNIV-k-BND is also PSPACE-hard for every k, using relatively similar tricks. However, since UNIV-PARAM-BND is already PSPACE-complete, then UNIV-k-BND is somewhat redundant. We therefore make do with the trivial lower bound whereby we reduce universality of NFA to UNIV-0-BND.

Theorem 24. *For* HAM*, the* UNIV-PARAM-BND *problem is PSPACE-complete for k encoded in unary and fixed alphabet Σ.*

7 Interplay Between the Semantics

Having established some decidability results, we now turn our attention to the interplay between the different semantics, in the context of boundedness. We show that for a given JFA $\mathscr{A}$, if $\mathscr{A}_{\text{ABS}}$ is bounded, then so is $\mathscr{A}_{\text{HAM}}$, and if $\mathscr{A}_{\text{HAM}}$ is bounded, then so is $\mathscr{A}_{\text{REV}}$. We complete the picture by showing that these are the only relationships – we give examples for the remaining cases (see Table 2).

Lemma 25. *Consider a* JFA $\mathscr{A}$*. If* $\mathscr{A}_{\text{ABS}}$ *is bounded, then* $\mathscr{A}_{\text{HAM}}$ *is bounded.*

Proof. Consider a word $w \in \Sigma^*$, we show that if $\mathscr{A}_{\text{ABS}}(w) \leq k$ for some $k \in \mathbb{N}$ then $\mathscr{A}_{\text{HAM}}(w) \leq (2k+1)(k+1)$. Assume $\mathscr{A}_{\text{ABS}}(w) \leq k$, then there exists a jump sequence $\mathbf{a} = (a_0, \ldots, a_{n+1})$ such that $[\![\mathbf{a}]\!] \leq k$ and $w_{\mathbf{a}} \in \mathfrak{L}(\mathscr{A})$. In the following we show that $a_i = i$ for all but $(2k+1)(k+1)$ indices, i.e., $|\{i \mid a_i \neq i\}| \leq (2k+1)(k+1)$.

It is convenient to think of the jumping head moving according to $\mathbf{a}$ in tandem with a sequential head moving from left to right. Recall that by Lemma 9, for every index i we have that $i-k \leq a_i \leq i+k$, i.e. the jumping head stays within distance k from the sequential head.

Consider an index i such that $a_i \neq i$ (if there is no such index, we are done). we claim that within at most $2k$ steps, $\mathscr{A}$ performs a jump of cost at least 1 according to $\mathbf{a}$. More precisely, there exists $i+1 \leq j \leq i+2k$ such that $|a_j - a_{j-1}| > 1$. To show this we split to two cases:

- If $a_i > i$, then there exists some $m \leq i$ such that m has not yet been visited according to $\mathbf{a}$ (i.e., by step i). Index m must be visited by a_i within at most k steps (otherwise it becomes outside the $i-k, i+k$ window around the sequential head), and since $a_i > i$, it must perform a “ left jump” of size at least 2 (otherwise it always remains to the right of the sequential reading head).
- If $a_i < i$, then there exists some $m \geq i$ such that m has already been visited by step i according to $\mathbf{a}$. Therefore, within at most $2k$ steps, the jumping head must skip at least over this position (think of m as a hurdle coming toward the jumping head, which must stay within distance k of the sequential head and therefor has to skip over it). Such a jump incurs a cost of at least 1.

Now, let $B = \{i \mid a_i \neq i\}$ and assume by way of contradiction that $|B| > (2k+1)(k+1)$. By the above, for every $i \in B$, within $2k$ steps the run incurs a cost of at least 1. While some of these intervals of $2k$ steps may overlap, we can still find at least $k+1$ such disjoint segments (indeed, every $i \in B$ can cause an overlap with at most $2k$ other indices). More precisely, there are $i_1 < i_2 < \ldots < i_{k+1}$ in B such that $i_j > i_{j-1} + 2k$ for all j, and therefore each of the costs incurred within $2k$ steps of visiting i_j is independent of the others. This, however, implies that $[\![\mathbf{a}]\!] \geq k+1$, which is a contradiction, so $|B| \leq (2k+1)(k+1)$.

It now follows that $\mathscr{A}_{\text{HAM}}(w) = |\{i \mid w_{a_i} \neq w_i\}| \leq |\{i \mid a_i \neq i\}| \leq (2k+1)(k+1)$ ☐

Lemma 26. *Consider a* JFA $\mathscr{A}$*. If* $\mathscr{A}_{\text{HAM}}$ *is bounded, then* $\mathscr{A}_{\text{REV}}$ *is bounded.*

Proof. Consider a word $w \in \Sigma^*$, we show that if $\mathscr{A}_{\text{HAM}}(w) \leq k$ for some $k \in \mathbb{N}$ then $\mathscr{A}_{\text{REV}}(w) \leq 3k$. Assume $\mathscr{A}_{\text{HAM}}(w) \leq k$, then there exists a jump sequence $\mathbf{a} = (a_0, \ldots, a_{n+1})$ such that $w_{\mathbf{a}} \in \mathfrak{L}(\mathscr{A})$ and $w_{\mathbf{a}}$ differs from w in at most k indices. We claim that we can assume without loss of generality that for every index i such that $w_{a_i} = w_i$ we have $a_i = i$ (i.e., i is a *fixed point*). Intuitively – there is no point swapping identical letters. Indeed, assume that this is not the case, and further assume that $\mathbf{a}$ has the minimal number of fixed-points among such jump sequences. Thus, there exists some j for which $a_j \neq j$ but $w_{a_j} = w_j$. Let m be such that $a_m = j$, and consider the jump sequence $\mathbf{a}' = (a'_0, \ldots, a'_{n+1})$ obtained from $\mathbf{a}$ by composing it with the swap $(a_j\ a_m)$. Then, for every $i \notin \{j, m\}$ we have that $a'_i = a_i$. In addition, $a'_j = a_m = j$ as well as $a'_m = a_j$. In particular, $\mathbf{a}'$ has more fixed points than a (exactly those of $\mathbf{a}$ and j). However, we claim that $w_{\mathbf{a}} = w_{\mathbf{a}'}$. Indeed, the only potentially-problematic coordinates are a_j

and a_m. For j we have $w_{a_j} = w_j = w_{a'_j}$. and for m we have $w_{a'_m} = w_{a_j} = w_j = w_{a_m}$. This is a contradiction to $\mathbf{a}$ having a minimal number of fixed points, so we conclude that no such coordinate $a_j \neq j$ exists.

Next, observe that $\mathtt{Turn}(\mathbf{a}) \subseteq \{i \mid a_i \neq i \vee a_{i+1} \neq i+1 \vee a_{i-1} \neq i-1\}$. Indeed, if $a_{i-1} = i-1$, $a_i = i$ and $a_{i+1} = i+1$ then clearly i is not a turning index. By the property established above, we have that $w_{a_i} = w_i$, if and only if $a_i = i$. It follows that $\mathtt{Turn}(\mathbf{a}) \subseteq \{i \mid w_{a_i} \neq w_i \vee w_{a_{i+1}} \neq w_{i+1} \vee w_{a_{i-1}} \neq w_{i-1}\}$, so $|\mathtt{Turn}(\mathbf{a})| \leq 3k$ (since each index where $w_{\mathbf{a}} \neq w$ is counted at most 3 times[3] in the latter set). ☐

Combining Lemmas 25 and 26, we have the following.

Corollary 27. *If* ABS *is bounded, then so is* REV.

We proceed to show that no other implication holds with regard to boundedness, by demonstrating languages for each possible choice of bounded/unbounded semantics (c.f. Remark 6). The examples are summarized in Table 2, and are proved below.

ABS	HAM	REV	Language
Bounded	Bounded	Bounded	$(a+b)^*$
Unbounded	Bounded	Bounded	$(a+b)^*a$
Unbounded	Unbounded	Bounded	a^*b^*
Unbounded	Unbounded	Unbounded	$(ab)^*$

Table 2: Examples for every possible combination of bounded/unbounded semantics. The languages are given by regular expressions (e.g., $(a+b)^*a$ is the languages of words that end with a.)

Example 28. *The language $(a+b)^*$ is bounded in all semantics. This is trivial, since every word is accepted, and in particular has cost* 0 *in all semantics.*

Example 29. *The language $(a+b)^*a$ is bounded in the* HAM *and* REV *semantics, but unbounded in* ABS. *Indeed, let $\mathscr{A}$ be an* NFA *such that $\mathfrak{L}(\mathscr{A}) = (a+b)^*a$ and consider a word $w \in \mathfrak{J}(\mathscr{A})$, then w has at least one occurrence of a at some index i. Then, for the jumping sequence $\mathbf{a} = (0,1,2,\ldots,i-1,n,i+1,\ldots n-1,i,n+1)$ we have that $w_{\mathbf{a}} \in \mathfrak{L}(\mathscr{A})$. Observe that $d_H(w_{\mathbf{a}},w) \leq 2$ (since $w_{\mathbf{a}}$ differs from w only in indices i and n), and $\mathtt{Turn}(\mathbf{a}) \subseteq \{i,n\}$, so $\mathscr{A}_{\mathrm{HAM}}(w) \leq 2$ and $\mathscr{A}_{\mathrm{REV}}(w) \leq 2$.*

For ABS, *however, consider the word ab^n for every $n \in \mathbb{N}$. Since the letter a must be read last, then in any jumping sequence accepting the word, there is a point where the jumping head is at index n and the sequential head is at position* 1. *By Lemma 9, it follows that $\mathscr{A}_{\mathrm{ABS}}(w) \geq n-1$, and by increasing n, we have that $\mathscr{A}_{\mathrm{ABS}}$ is unbounded.*

Example 30. *The language a^*b^* is bounded in the* REV *semantics, but unbounded in* HAM *and* ABS. *Indeed, let $\mathscr{A}$ be an* NFA *such that $\mathfrak{L}(\mathscr{A}) = a^*b^*$ and consider a word $w \in \mathfrak{J}(\mathscr{A})$, and denote by $i_1 < i_2 \ldots < i_k$ the indices of a's in w in increasing order, and by $j_1 > j_2 > \ldots > j_{n-k}$ the indices of b's in decreasing order. Then, for the jumping sequence $\mathbf{a} = (i_1,\ldots,i_k,j_1,\ldots j_{n-k},n+1)$ we have that $w_{\mathbf{a}} \in \mathfrak{L}(\mathscr{A})$, and $\mathscr{A}_{\mathrm{REV}}(w) \leq 2$ (since the jumping head goes right reading all the a's, then left reading all the b's, then jumps to $n+1$).*

For HAM, *consider the word $w = b^na^n$ for every $n \in \mathbb{N}$. The only permutation of w that is accepted in $\mathfrak{L}(\mathscr{A})$ is $w' = a^nb^n$, and $d_H(w,w') = n$, so $\mathscr{A}_{\mathrm{HAM}}$ is unbounded. By Lemma 25 it follows that $\mathscr{A}_{\mathrm{ABS}}$ is also unbounded.*

[3] A slightly finer analysis shows that this is in fact at most $2k$, but we are only concerned with boundedness.

Example 31. *The language $(ab)^*$ is unbounded in all the semantics. Indeed, let $\mathscr{A}$ be an* NFA *such that $\mathfrak{L}(\mathscr{A}) = (ab)^*$, then by Lemma 26 and Corollary 27 it suffices to show that $\mathscr{A}_{\text{REV}}$ is unbounded.*

Consider the word $w = b^n a^n$ for every $n \in \mathbb{N}$, and let $\mathbf{a} = (a_0, a_1, \ldots, a_{2n}, a_{2n+1})$ such that $w_{\mathbf{a}} \in (ab)^*$, then for every odd $i \leq 2n$ we have $a_i \in \{n+1, \ldots, 2n\}$ and for every even $i \leq 2n$ we have $a_i \in \{1, \ldots, n\}$. In particular, every index $1 \leq i \leq 2n$ is a turning point, so $\mathscr{A}_{\text{REV}}(w) = 2n$, and $\mathscr{A}_{\text{REV}}$ is unbounded.

8 Discussion and Future Work

Quantitative semantics are often defined by externally adding some quantities (e.g., weights) to a finite-state model, usually with the intention of explicitly reasoning about some unbounded domain. It is rare and pleasing when quantitative semantics arise naturally from a Boolean model. In this work, we study three such semantics. Curiously, despite the semantics being intuitively unrelated, it turns out that they give rise to interesting interplay (see Section 7).

We argue that Boundedness is a fundamental decision problem for the semantics we introduce, as it measures whether one can make do with a certain budget for jumping. An open question left in this research is *existentially-quantified boundedness*: whether there *exists* some bound k for which $\mathscr{A}_{\text{SEM}}$ is k-bounded. This problem seems technically challenging, as in order to establish its decidability, we would need to upper-bound the minimal k for which the automaton is k-bounded, if it exists. The difficulty arises from two fronts: first, standard methods for showing such bounds involve some pumping argument. However, the presence of permutations makes existing techniques inapplicable. We expect that a new toolbox is needed to give such arguments. Second, the constructions we present for UNIV-PARAM-BND in the various semantics seem like the natural approach to take. Therefore, a sensible direction for the existential case is to analyze these constructions with a parametric k. The systems obtained this way, however, do not fall into (generally) decidable classes. For example, in the HAM semantics, using a parametric k we can construct a labelled VASS. But the latter do not admit decidable properties for the corresponding boundedness problem.

We remark on one fragment that can be shown to be decidable: consider a setting where the jumps are restricted to swapping disjoint pairs of adjacent letters, each incurring a cost of 1. Then, the JFA can be translated to a weighted automaton, whose boundedness problem is decidable by [15, 18]. We remark that the latter decidability is a very involved result. This suggests (but by no means proves) that boundedness may be a difficult problem.

References

[1] Antonio Abu Nassar & Shaull Almagor (2022): *Simulation by Rounds of Letter-To-Letter Transducers*. In: *30th EACSL Annual Conference on Computer Science Logic*, doi:10.4230/LIPIcs.CSL.2022.3.

[2] Shaull Almagor (2020): *Process Symmetry in Probabilistic Transducers*. In: *40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, doi:10.4230/LIPIcs.FSTTCS.2020.35.

[3] Shaull Almagor & Orna Kupferman (2011): *Max and sum semantics for alternating weighted automata*. In: *International Symposium on Automated Technology for Verification and Analysis*, Springer, pp. 13–27, doi:10.1007/978-3-642-24372-1_2.

[4] Shaull Almagor & Omer Yizhaq (2023): *Jumping Automata over Infinite Words*. In: *International Conference on Developments in Language Theory*, Springer, pp. 9–22, doi:10.1007/978-3-031-33264-7_2.

[5] Udi Boker (2021): *Quantitative vs. weighted automata*. In: *Reachability Problems: 15th International Conference, RP 2021, Liverpool, UK, October 25–27, 2021, Proceedings 15*, Springer, pp. 3–18, doi:10.1007/978-3-030-89716-1_1.

[6] Michaël Cadilhac, Alain Finkel & Pierre McKenzie (2012): *Affine Parikh automata*. *RAIRO-Theoretical Informatics and Applications* 46(4), pp. 511–545, doi:10.1051/ita/2012013.

[7] Michaël Cadilhac, Alain Finkel & Pierre McKenzie (2012): *Bounded parikh automata*. *International Journal of Foundations of Computer Science* 23(08), pp. 1691–1709, doi:10.1142/S0129054112400709.

[8] Krishnendu Chatterjee, Laurent Doyen & Thomas A Henzinger (2010): *Quantitative languages*. *ACM Transactions on Computational Logic (TOCL)* 11(4), pp. 1–38, doi:10.1007/978-3-540-87531-4_28.

[9] Manfred Droste, Werner Kuich & Heiko Vogler (2009): *Handbook of weighted automata*. Springer Science & Business Media, doi:10.1007/978-3-642-01492-5.

[10] Szilárd Zsolt Fazekas, Kaito Hoshi & Akihiro Yamamura (2021): *Two-way deterministic automata with jumping mode*. *Theoretical Computer Science* 864, pp. 92–102, doi:10.1016/j.tcs.2021.02.030.

[11] Henning Fernau, Meenakshi Paramasivan & Markus L Schmid (2015): *Jumping finite automata: characterizations and complexity*. In: *International Conference on Implementation and Application of Automata*, Springer, pp. 89–101, doi:10.1007/978-3-319-22360-5_8.

[12] Henning Fernau, Meenakshi Paramasivan, Markus L Schmid & Vojtěch Vorel (2017): *Characterization and complexity results on jumping finite automata*. *Theoretical Computer Science* 679, pp. 31–52, doi:10.1016/j.tcs.2016.07.006.

[13] Dana Fisman, Joshua Grogin & Gera Weiss (2023): *A Normalized Edit Distance on Infinite Words*. In: *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*, Schloss-Dagstuhl-Leibniz Zentrum für Informatik, doi:10.4230/LIPIcs.CSL.2023.20.

[14] Shibashis Guha, Ismaël Jecker, Karoliina Lehtinen & Martin Zimmermann (2022): *Parikh Automata over Infinite Words*. In: *42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, doi:10.4230/LIPIcs.FSTTCS.2022.40.

[15] Kosaburo Hashiguchi (1982): *Limitedness theorem on finite automata with distance functions*. *Journal of computer and system sciences* 24(2), pp. 233–244, doi:10.1016/0022-0000(82)90051-4.

[16] Felix Klaedtke & Harald Rueß (2003): *Monadic second-order logics with cardinalities*. In: *Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30*, Springer, pp. 681–696, doi:10.1007/3-540-45061-0_54.

[17] Giovanna J Lavado, Giovanni Pighizzini & Shinnosuke Seki (2014): *Operational state complexity under Parikh equivalence*. In: *Descriptional Complexity of Formal Systems: 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings 16*, Springer, pp. 294–305, doi:10.1007/978-3-319-09704-6_26.

[18] Hing Leung & Viktor Podolskiy (2004): *The limitedness problem on distance automata: Hashiguchi's method revisited*. *Theoretical Computer Science* 310(1-3), pp. 147–158, doi:10.1016/S0304-3975(03)00377-3.

[19] Alexander Meduna & Petr Zemek (2012): *Jumping finite automata*. *International Journal of Foundations of Computer Science* 23(07), pp. 1555–1578, doi:10.1142/S0129054112500244.

[20] Mehryar Mohri (2002): *Edit-distance of weighted automata*. In: *International Conference on Implementation and Application of Automata*, Springer, pp. 1–23, doi:10.1007/3-540-44977-9_1.

[21] Vojtěch Vorel (2018): *On basic properties of jumping finite automata*. *International Journal of Foundations of Computer Science* 29(01), pp. 1–15, doi:10.1142/S0129054118500016.

Reactive Synthesis for Expected Impacts

Emanuele Chini
Department of Computer, Control
and Management Engineering,
University "La Sapienza",
Rome, Italy.
emanuele.chini@uniroma1.it
Department of Computer Science,
University of Verona, Verona (Italy)
emanuele.chini@univr.it

Pietro Sala
Department of Computer Science,
University of Verona, Verona (Italy)
pietro.sala@univr.it

Andrea Simonetti
Department of Computer Science,
University of Verona, Verona (Italy)
andrea.simonetti@studenti.univr.it

Omid Zare
Department of Computer Science,
University of Verona, Verona (Italy)
omid.zare@univr.it

As business processes become increasingly complex, effectively modeling decision points, their likelihood, and resource consumption is crucial for optimizing operations. To address this challenge, this paper introduces a formal extension of the Business Process Model and Notation (BPMN) that incorporates choices, probabilities, and impacts, referred to as BPMN+CPI. This extension is motivated by the growing emphasis on precise control within business process management, where carefully selecting decision pathways in repeated instances is crucial for conforming to certain standards of multiple resource consumption and environmental impacts. In this context we deal with the problem of synthesizing a strategy (if any) that guarantees that the expected impacts on repeated execution of the input process are below a given threshold. We show that this problem belongs to PSPACE complexity class; moreover we provide an effective procedure for computing a strategy (if present).

1 Introduction

BPMN (Business Process Model and Notation) has emerged as a pivotal formalism in the realm of process management, offering a standardized method for detailing business processes in various sectors, including healthcare and industry. Its graphical notation facilitates the clear and precise representation of process flows, enabling stakeholders to comprehend, analyze, and improve business operations. In the healthcare sector, BPMN plays a critical role in implementing patient care guidelines [26]. Similarly, in the industrial domain, it aids in the efficient management of manufacturing and supply chain processes, ensuring timely delivery of products and services [15]. In these domains, increasing attention has arisen in the past decade on the topic of Business Processes Management, where the choice of traces on the control side is paramount. These applications demand measurement and employ, as a means for selection, notions such as *cost-awareness*[23], *energy-awareness*[5], and *resource-awareness* [11], which naturally induce scenarios where multiple measurements must be controlled.

In this paper, we proceed under the implicit assumption that all costs, energies, and resources utilized are positive and exhibit additive characteristics. This implies that our process instances solely deplete resources to fulfill their objectives without the capability to generate resources. As we will demonstrate, this restriction contributes to favorable computational properties.

Moreover, we use the probabilistic split, referred to as *nature*, which signifies a decision based on a probability distribution beyond the worker's control. For instance, in healthcare, a *nature* is the chance

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 35–52, doi:10.4204/EPTCS.409.7

© E. Chini, A. Simonetti, P. Sala, and O. Zare
This work is licensed under the
Creative Commons Attribution License.

of developing gastritis when taking Brufen 600 with a probability of 1%. Similarly, in industrial applications, machinery wear and tear may influence the production process, requiring maintenance stops during production.

Finally, time consumption for tasks is considered, as they will be equipped with specific durations.

Our approach here is twofold. First, we aim to introduce a formal BPMN extension that addresses execution in the presence of all the previously mentioned components, namely, BPMN plus Choices/Probability/Impacts (BPMN+CPI). Next, we seek to provide a dynamic control mechanism, i.e., a strategy, for BPMN execution. This is to ensure, where possible, that the expected impacts remain below a set of user-defined thresholds. To elegantly juggle all these concepts within a single framework, we enrich the standard Petri Net semantics for BPMN to capture impacts, durations, and probabilities. We call this model of computation the Simultaneous Probabilistic Impactful Network (SPIN). We then define a graph representing all possible executions of SPIN. This graph is combined with a natural modification of classical reachability games to derive the desired strategy, if any. The primary aim of this study is to determine, for a process formalized in BPMN+CPI, whether a controller exists that can accurately execute each step of the process while ensuring that the expected value of each resource, across repeated process instances, remains within predefined thresholds.

Upon establishing the computational model for BPMN+CPI, we tackle the challenge of synthesizing a strategy for a specified process in BPMN+CPI, given a set of expected value thresholds. This is achieved through the following steps:

1. *Semantics by Petri Nets*. After defining how to translate a BPMN+CPI into a SPIN, we define the semantics of both of them by giving the semantics of SPIN alone as an extension of classical Petri net semantics. This includes introducing time durations for places, probabilistic transitions, and the possibility (under certain conditions) of executing a set of enabled transitions simultaneously instead of one at a time;
2. *Computation Graph*. All possible computations for the given SPIN are represented as a graph. In this graph, each node represents a path of executions, and any edge between two computations indicates that the source computation can be extended to the target computation by firing one or more enabled transitions in the source computation;
3. *Classical Reachability Game Graph Transformation* [27]. By transforming the computation graph into a classical reachability game graph, where spoiler nodes (typically denoted by $\square$) represent choices made by nature, we assess the existence of a "good" set of final states that can "attract" the initial state. If such a set exists, we can infer the existence of our strategy.

The paper is organized as follows. In Section 2, we present and describe related work and the state-of-the-art algorithms for finding strategies in computational models that can encode BPMN+CPI through suitable translation. In Section 3, we illustrate a practical example of a BPMN process in an industrial setting, followed by a formal definition of the BPMN+CPI model, detailing the components of choices, probabilities, and impacts. Since we restrict ourselves to acyclic graphs, at the end of this section, we briefly discuss a simple way of dealing with loops within the proposed framework. In Section 4, we provide the complexity bounds for the strategy synthesis problem for BPMN+CPI. While Section 4 deals with the decision problem of establishing whether a strategy exists or not, Section 5 focuses on effectively synthesizing a strategy given a BPMN+CPI process and a bound for expected impacts. Finally, Section 6 highlights our main findings, their theoretical and practical impacts, and future research avenues.

Methods	Costs	Durations	Strategy
UPPAAL-Stratego	multiple, not considered for strategy	explicitly defined, time is continuous	•non-deterministic • state explosion due to subset construction
PRISM	multiple negative allowed	implicit via multiple states	• ϵ-approximated strategy • increases exponentially w.r.t $1/\epsilon$
MPG-MDP	multiple negative allowed	implicit via multiple states	• infinite plays • BPMN+CPI would need difficult encoding • game averages values on a per-step basis
Our method	multiple, only positive	explicitly defined	• deterministic • exact strategy by integrating rewards and probabilities • game averages values on a per-instance basis

Table 1: A summary of the features of the tool introduced in this study and the problems addressed by UPPAAL-Stratego, PRISM, and MPG-MDP, respectively.

2 Related work

The most commonly accepted semantics for BPMN processes, used for both formal tasks like monitoring, verification, and querying, and application-driven tasks like process discovery and execution forecasting, is the Petri Net semantics. In this approach, a BPMN process is mapped into a Petri Net [12]. This mapping retains several beneficial properties, including the crucial feature that the resulting net is 1-bounded [6], meaning that from an initial state with one token, all configurations will have at most one token per place. Under this 1-boundedness assumption, the Petri Net reduces to an exponentially succinct representation of a finite automaton (FA) [18], where all labelings can be represented as sets of places holding one token, making the number of states finite. If the language of this automaton is defined by its transitions, the resulting FA is deterministic (DFA). Thus, many formal problems, such as querying, emptiness checking, strategy synthesis (reachability games), and Linear Temporal Logic (LTL) model checking, can be equivalently viewed in BPMN, 1-bounded Petri Nets, or succinct DFAs, as transformations between these representations can be performed in LOGSPACE.

Incorporating resources into BPMN processes is well-explored in process optimization literature. In [23], an extension of the classical BPMN notation is proposed to evaluate the overall cost of process diagrams, comparing costs associated with tasks as single values or intervals to find the most cost-effective way to perform the intended job. Our contribution specifically focuses on the positive impacts of such integration, further allowing the specification of impacts as arrays of cost values to express monetary costs and other resources or requirements. In [9], Combi et al. outlined a method for enforcing distinctive temporal behaviors by introducing temporal patterns (e.g., minimum and/or maximum durations) linked to tasks. They proposed creating reusable, duration-aware process models using existing BPMN elements, capturing duration constraints at various abstraction levels, and checking for duration constraint violations at runtime. Duran et al. [14] introduced a rewriting logic executable specification of BPMN extended with time and probabilities, allowing stochastic expressions to specify task durations and flow delays. Herbert et al. [19] formalize an extension of the BPMN language incorporating probabilistic non-deterministic branching. Additionally, they present an algorithm for translating such models into MDPs expressed in the syntax of the PRISM model checker [22]. This facilitates precise quantitative analysis of business processes. We have adopted a similar extension of BPMN to introduce non-deterministic behaviour (for nature nodes), which is frequently observed in real-world application scenarios. Probabilities are linked to gateway branching behaviors, enabling discrete-event simulation and automatic stochastic

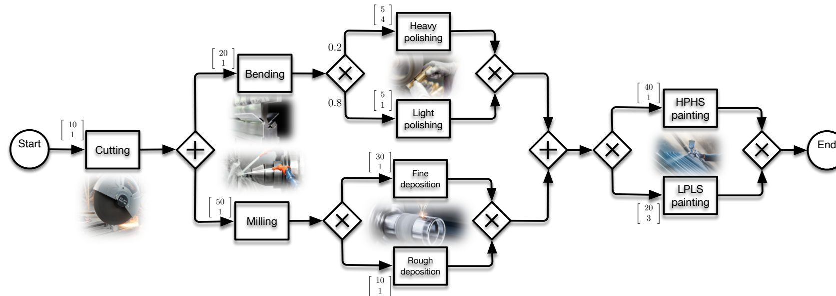


Figure 1: An example of BPMN+CPI diagram for an industrial process.

verification of various properties. Our work will consider task durations by imposing stringent time constraints, ensuring that each task extends over a time interval precisely equal to its duration, which possibly affects which choice is enabled first in a given execution. Additionally, incorporating probabilities into BPMN situates our research within the specialized domain of Markov Decision Processes (MDPs) [17], significantly enhancing the applicability of BPMN in decision-making under uncertainty.

Moving beyond BPMN, our approach primarily involves devising a strategy within an MDP enhanced with vectors of positive impacts. The objective is to ensure that the strategy's expected value does not exceed a specific threshold. The realm of strategy synthesis for MDPs has been extensively explored, leading to notable breakthroughs like the PRISM model checker [22]. PRISM has emerged as a key instrument, evolving over time to incorporate sophisticated features for strategizing within MDP contexts. Another notable development is UPPAAL-Stratego [10], an extension of the well-regarded UPPAAL-TIGA [4], which solves the strategy synthesis problem for games played on timed automata incorporating both costs and probabilities. From a theoretical perspective, albeit less focused on specific tools, our issue shares similarities with Mean Payoff Games (MPG)[28] as applied to MDP (MPG-MDP)[8]. The differences and similarities between our proposed method and the current state of the art are concisely summarized in Table 1. While we focus on a system with probabilities, we are aware of other formalisms that allow impact vectors with negative contributions, such as infinite energy games [2].

3 BPMN+CPI: Processes with Choices, Probabilities, and Impacts

In this section, we begin by informally illustrating the concept of BPMN+CPI through an intuitive example of a metal manufacturing process together with an initial, intuitive understanding of the expected impacts induced by a strategy in Section 3.1. These concepts are then formalized in Section 3.2, where we also state the core problem of this work: finding an optimal strategy that minimizes the overall impact. Finally, in Section 3.3, we discuss the advantages and drawbacks of reducing diagrams with loops to acyclic ones from the perspective of strategy synthesis.

3.1 Motivating Example

The BPMN+CPI diagram of Figure 1 depicts a metal manufacturing process that involves cutting, milling, bending, polishing, depositioning, and painting a metal piece. It consists of a single-entry-single-exit (SESE) diagram, with a choice, a nature, and an impact for each task, which is defined as a numbers vector. The bracketed numbers next to each activity represent impact vectors $\begin{bmatrix} a \\ b \end{bmatrix}$ where a = cost of the task and b = hours/men required to complete the task. For instance, cutting the metal piece costs 10 units (e.g.,

currency, resource, etc.), and requires 1 unit of time or manpower (e.g., 1 hour or 1 worker). In Figure 1, the nature's probability of each chosen path is indicated with the numbers next to decision points. For example, there's a high probability (0.8) of the process moving from bending to light polishing and a low probability (0.2) of it moving to fine heavy polishing.

Whenever the process is executed, the worker and nature make a series of choices, which result in a path executed on the BPMN with a total impact vector for that specific instance. Let's now assume that, for economic reasons, the process must stay within a certain bound. Therefore, our interest is always to stay below that bound. However, we have to consider that the path also depends on the natures within the process, of which we do not know the choice a priori, but we only have the probability of going one way or the other. Consequently, we can formulate a strategy, defined as a series of choices taken while considering the nature and a maximum expected impact, to manage to reach the end of the process with a certain impact vector.

Strategy example: after cutting the metal piece, we have two tasks after the parallel split node, so we do the bending and milling in parallel. Then, after milling we have two options to choose from, here we choose fine deposition. After bending, we have two options to choose from: we choose light polishing with the probability of 0.8. Then, we have two final tasks to choose from: we select LPLS painting. Finally, we have the maximum expected impact of $\begin{bmatrix}115\\11\end{bmatrix}\times 0.2+\begin{bmatrix}135\\8\end{bmatrix}\times 0.8=\begin{bmatrix}131\\8.6\end{bmatrix}$.

A strategy is defined as winning only if the expected impact vector is below the bound. Therefore, the goal is to find a winning strategy. Consider, for example, that you want to keep the BPMN+CPI visible in Figure 1 under the limit of $ei=\begin{bmatrix}155\\7.5\end{bmatrix}$. In this case, the strategy shown is not a winning strategy. In fact, it presents a maximum expected impact greater than the bound ei. Below we propose an example of a winning strategy.

Wining strategy example: after cutting we perform milling in parallel with bending. we have two options that come after milling; we choose fine deposition. We have two options to choose from after bending; we choose light polishing with a probability of 0.8. Then, we have two final tasks to choose from and select HPHS painting this time. Finally, we have $\begin{bmatrix}135\\9\end{bmatrix}\times 0.2+\begin{bmatrix}155\\6\end{bmatrix}\times 0.8=\begin{bmatrix}151\\6.6\end{bmatrix}\leq ei$, so this strategy successfully keeps the overall impact below the expected impact.

3.2 Problem Formulation

In this section, we formally state the BPMN+CPI semantics. First, we define the concept of Structured Single-Entry Single-Exit (SESE) BPMN, Figure 2, as follows.

Definition 1. *A* structured single-entry-single-exit diagram*, from now on simply a SESE diagram, is a directed graph $\mathcal{D}=(V,E,E_\top,\mathcal{T})$ where (V,E) is a directed graph, $E_\top\subseteq E$, $\mathcal{T}:V\to\{event, task, join, split\}$ such that:*

1. *for each $v\in V$ if $\mathcal{T}(v)=event$ then there exists at most one edge departing from v, there exists at most one edge entering v, and at least one edge departing from v or entering v, i.e., $|\{(v,v')\in E\}|\leq 1$, $|\{(v',v)\in E\}|\leq 1$, and $|\{(v',v)\in E\}\cup\{(v,v')\in E\}|>0$;*
2. *there exists exactly two distinct nodes $\hat{v},\check{v}$ in V such that $\hat{v}$ has not incoming edges and $\check{v}$ has not outgoing edges, i.e., $\{(v,\hat{v})\in E\}=\{(\check{v},v)\in E\}=\emptyset$;*
3. *for each $v\in V$ if $\mathcal{T}(v)=task$ there exists exactly one edge departing from v and one edge entering v, i.e., $|\{(v,v')\in E\}|=|\{(v',v)\in E\}|=1$;*
4. *for each $v\in V$ if $\mathcal{T}(v)=split$ there exists exactly two edges departing from v and one edge entering v, i.e., $|\{(v,v')\in E\}|=2$ and $|\{(v',v)\in E\}|=1$;*
5. *$E_\top\subseteq\{(v,v'):\mathcal{T}(v)=split\}$ and for each $v\in V$ if $\mathcal{T}(v)=split$ we have $|\{v':(v,v')\in E_\top\}|=1$;*

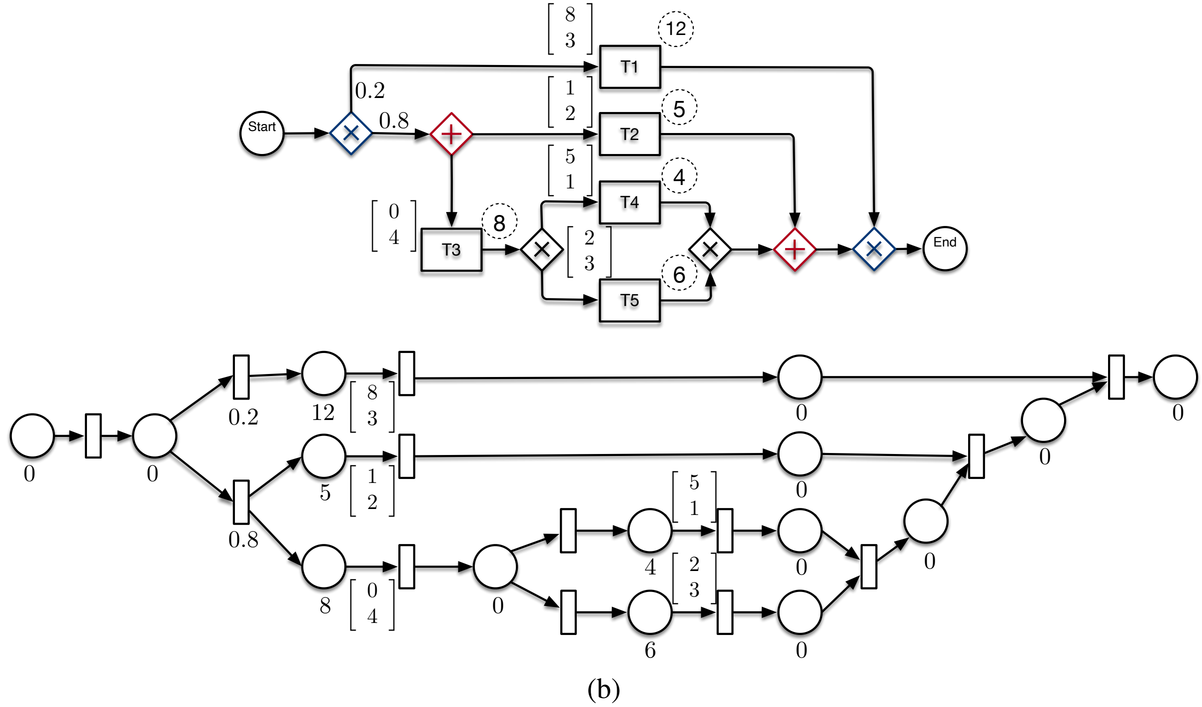


Figure 2: A BPMN+CPI utilizing all the components considered in this work (a) and its SPIN translation (b).

6. *for each $v \in V$ if $\mathcal{T}(v) = join$ there exists exactly one edge departing from v and two edges entering v, i.e., $|\{(v,v') \in E\}| = 1$ and $|\{(v',v) \in E\}| = 2$;*

Every non-SESE BPMN diagram can be translated into a SESE diagram as demonstrated in [13].

In particular, in the rest of this work, we will restrict ourselves to acyclic SESE diagrams. We will discuss this limitation and how it can be overcome in Section 3.3.

We define BPMN+CPI processes as follows.

Definition 2. *A BPMN+CPI is a tuple $Pcpi = (\mathcal{D}, \mathcal{P}, \mathcal{I}, \delta)$ where $\mathcal{D} = (V, E, \mathcal{T})$ is a SESE diagram, and $\mathcal{P} : split(V) \to \mathbb{R}_{[0,1]}$ is a* partial function, *$\mathcal{I} : task(V) \to (\mathbb{R}_{\geq 0})^k$ with $k \in \mathbb{N}$, and $\delta : task(V) \to \mathbb{N}^+$.*

Let us notice that since $\mathcal{P}$ is a partial function, it suffices to encode the natural split gateways in the diagrams, i.e., the one with associated probabilities. Then we may define V_{nature} as the set $V_{nature} = Dom(\mathcal{P})$ and, on the other hand, for the choice of the system V_{choice} as $V_{choice} = split(V) \setminus V_{nature}$.

Let us now extend the semantics of classical Petri nets [25] in order to capture the semantics of BPMN+CPI process.

Definition 3. *A Simultaneous Probabilistic Impactful Network (*SPIN*) is a tuple $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$ where P and T are finite disjoint set of places and transition, respectively, $T_p \subseteq T$, $\Delta \subseteq (P \times T) \cup (T \times P)$, $I : T \to \mathbb{N}^k$, $D : P \to \mathbb{N}$, and $Pr : T_p \to [0,1]$.*

Given a SPIN $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$ for each $pt \in PT$ let $incoming(pt) = \{pt' \in PT : (pt', pt) \in \Delta\}$ and let $outgoing(pt) = \{pt' \in PT : (pt, pt') \in \Delta\}$. Here we focus on a specific restriction of SPIN called *structured acyclic* SPIN.

Definition 4. *We say that a* SPIN $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$ *is* structured and acyclic *if and only if the directed graph (PT, Δ) is acyclic, and the following conditions hold:*

1. *for each $pt \in PT$ we have, $|incoming(pt)| \leq 2$ $|outgoing(pt)| \leq 2$ and $|incoming(pt)| + |outgoing(pt)| \leq 3$;*
2. *there exists a unique partition $\mathcal{T}_p = \{t_1, \bar{t}_1\}, \dots, \{t_m, \bar{t}_m\}$ of T_p such that $Pr(t_i) = 1 - Pr(\bar{t}_i)$, $|outgoing(t_i)| = |outgoing(\bar{t}_i)| = 1$, and $incoming(t_i) = incoming(\bar{t}_i)$;*
3. *there exists a unique set cover $PT_1, \dots PT_m$ of PT such that for each pair PT_i, PT_j of the cover $PT_i \cup PT_j$ also belongs to the cover and the following conditions hold:*
   - *$pt \in PT$ we have that there exists at most two incoming and two outgoing edges, and the cardinality of the incoming and outgoing edges is at most 3, i.e., $\{(pt', pt)\}$.*
   - *for each pair PT_i, PT_j $PT_i \cap PT_j = \emptyset$, or $PT_i \subseteq PT_j$, or $PT_j \subseteq PT_i$;*
   - *for each $PT_i \neq PT$ there exists a unique element $pt_{in(i)} \in PT_i$ (resp., $pt_{out(i)} \in PT_i$) such that $\{pt_{in(i)}\} = \{pt : (pt', pt) \in \Delta, pt' \notin PT_i, pt \in PT_i\}$ (resp., $\{pt_{out(i)}\} = \{pt : (pt, pt') \in \Delta, pt' \notin PT_i, pt \in PT_i\}$);*
   - *for each $PT_i \neq PT$ all the elements of PT_i are reachable from $pt_{in(i)}$ via Δ and all the elements of PT_i can reach $pt_{out(i)}$ via Δ.*

The class of SPIN, as captured by Definition 4, is the counterpart of acyclic BPMN+CPI. The formal translation from BPMN+CPI to SPIN provided which enriches the work [12], is not here shown for the sake of brevity. However, an example that includes the main BMPN elements is shown in Figure 2.

Let us notice that by the above definition a structured acyclic SPIN, a SPIN from now on, features exactly one place p_0 with $incoming(p_0) = \emptyset$ and a unique place p_f with $outgoing(p_f) = \emptyset$. Let us define a switch function $sw : T_p \to T_p$ such that for every $t \in T_p$ $\{sw(t), t'\} \in \mathcal{T}_p$. Basically sw act as a tool that allow us, for every probabilistic transition t, to access the unique other probabilistic $\bar{t}$ transition which shares the same incoming place of t.

Let us now formally define how computations work for SPINs. Given a SPIN $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$, a state $q : P \to \mathbb{N} \cup \{\epsilon\}$ is a function that maps places in temporal units, where ϵ states that the specific place has not been visited yet, or that it has already been visited.

Initial state q_0 and final state q_f for a SPIN are defined as follows: $q_0(p) = \begin{cases} 0 & \text{if } p = p_0 \\ \epsilon & \text{otherwise} \end{cases}$; $q_f(p) = \begin{cases} 0 & \text{if } p = p_f \\ \epsilon & \text{otherwise} \end{cases}$.

We will say that a transition $t \in T$ is enabled in a state q if and only if, for all $p \in incoming(t)$, $q(p) \geq D(p)$. Let us introduce now the concept of saturated state.

Definition 5. *Given a state q for a spin $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$ we say that q is saturated if and only if there exists at least one transition $t \in T$ which is enabled in q*

Since in a not saturated state q no transition $t \in T$ is enabled the net will be stuck in q. Then the intuition behind not saturated states is that the corresponding BPMN+CPI process is waiting for one or more tasks to terminate before going further. For getting out of such not saturated states we introduce a special transition t_w, the so called *wait transition* which encode the passing of one time units and it is enabled only in not saturated states.

Unlike classical Petri Nets, where each transition is fired one at the time here may fire either t_w or a subset of T called maximal non-conflicting enabled transition set.

Definition 6. *Given a state q for a spin $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$ and a subset $\overline{T} \subseteq T$ we say that $\overline{T}$ is a* maximal non-conflicting enabled transition set, *MNCE for short, in q if and only if the following conditions hold:*

1. *for each $t \in \overline{T}$ we have that t is enabled in q (*enabled*);*
2. *for each $t, t' \in \overline{T}$ with $t \neq t'$ we have $(incoming(t) \cup outgoing(t)) \cap (incoming(t') \cup outgoing(t')) = \emptyset$ (*non-conflicting*);*
3. *for any $t \in T \setminus \overline{T}$ we have that $\overline{T} \cup \{t\}$ violates the above two conditions (*maximal*);*

Given a set of transitions $\overline{T} \subseteq T$, let

$$OutPlaces(\overline{T}) = \bigcup_{t \in \overline{T}} outgoing(t)$$

and let

$$Places(\overline{T}) = \bigcup_{t \in \overline{T}} incoming(t) \cup OutPlaces(\overline{T}).$$

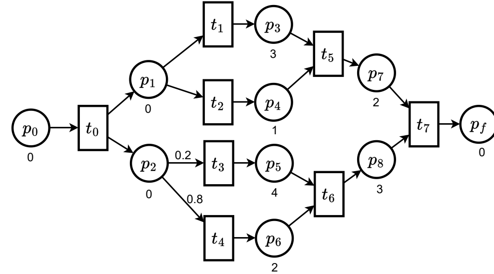


Figure 3: A SPIN for illustrating *MNCE* and probabilistic variants.

Now we are ready to define the transition relation between states in a SPIN. Let $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$ a spin for any pair of states q,q' for it we have:

$$q \overset{t_w}{\rightarrow} q' \ \textbf{iff} \ \begin{array}{c} q \text{ is not saturated} \\ \text{and} \\ q'(p) = \begin{cases} q(p)+1 & \text{if } q(p) \in \mathbb{N} \\ \epsilon & \text{otherwise} \end{cases} \end{array} \ ; \quad q \overset{\overline{T}}{\rightarrow} q' \ \textbf{iff} \ \begin{array}{c} q \text{ is saturated, } \overline{T} \text{ is an } MNCE \text{ in } q, \\ \text{and} \\ q'(p) = \begin{cases} q(p)+1 & \text{if } q(p) \in \mathbb{N} \text{ and } p \notin Places(\overline{T}) \\ 0 & \text{if } p \in OutPlaces(\overline{T}) \\ \epsilon & \text{otherwise} \end{cases} \end{array} .$$

Definition 7. *A computation $c = q_0 \overset{\overline{T_1}}{\rightarrow} \dots \overset{\overline{T_n}}{\rightarrow} q$ in a SPIN is a sequence of sets of transitions $\overline{T_i}$ where for each $1 \leq i \leq n$ we have that $\overline{T_i}$ is either t_w or an MNCE for q_{i-1}.*

A computation $c = q_0 \overset{\overline{T_1}}{\rightarrow} \dots \overset{\overline{T_n}}{\rightarrow} q$ in a SPIN is called a final computation if $q = q_f$. Stated that $I(t_w) = 0^k$ we can compute $I(c) = \sum_{t \in \bigcup_{i=1}^{n} \overline{T_i}} I(t)$ the impact associated with the computation c and $p(c) = \prod_{t \in \bigcup_{i=1}^{n} \overline{T_i} \cap T_p} Pr(t)$, the probability associated with the computation c. Let $T_{\not p} = T \setminus T_p$, that is, the set of transitions devoid of probabilistic transition, a strategy is defined as follows.

Definition 8. *Let $\mathbb{C}$ be the set of all the computations for a SPIN, we can define a strategy $S : \mathbb{C} \rightarrow 2^{T_{\not p}} \cup \{t_w\}$, a function that maps computations either into subsets of $T_{\not p}$ or into t_w.*

So, starting from a computation c in which we have reached the last state of the sequence, a strategy $S(c)$ tells us which are the next non-probabilistic transitions that are going to be *fired*. For all computations $c = q_0 \overset{\overline{T_1}}{\rightarrow} \dots \overset{\overline{T_n}}{\rightarrow} q$ we implicitly assume that $S(c)$ is t_w if q is not saturated and for and does not exists an enabled transition $t \in T_{\not p} \setminus S(c)$ such that $t \cup S(c)$ is non-conflicting, i.e., $S(c)$ may always be completed into an *MNCE* for q. Given a computation $c = q_0 \overset{\overline{T_1}}{\rightarrow} \dots \overset{\overline{T_i}}{\rightarrow} q_i \overset{\overline{T_{i+1}}}{\rightarrow} \dots \overset{\overline{T_n}}{\rightarrow} q$, we refer to the first i transitions sets of the sequence with the term sub-computation, written $c_{[0\dots i]}$.

Definition 9. *Given a strategy S, a play of S is a computation $c = q_0 \overset{\overline{T_1}}{\rightarrow} \dots \overset{\overline{T_n}}{\rightarrow} q$, such that for all sub-computations $c_{[0\dots i]}$, $S(c_{[0\dots i]}) \in \overline{T}_{i+1}$.*

Let $Games(S)$ be the set of all the *final* computations in $\mathbb{C}$ which are also plays of S.

Definition 10. *Given a vector bound $\mathbb{E}\mathbb{I} \in \mathbb{N}^k$, a strategy S is said to be winning for $\mathbb{E}\mathbb{I}$ if and only if $\sum_{c \in Games(S)} p(c) I(c) \leq \mathbb{E}\mathbb{I}$.*

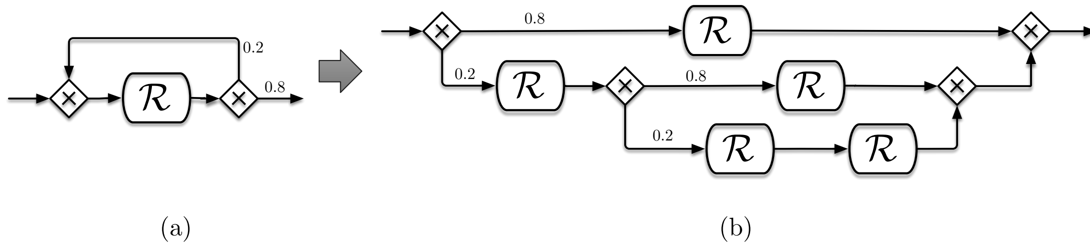


Figure 4: An example of a 2-unraveling of a loop region.

Finally, we highlight the problem we aim to resolve throughout this work.

Problem 1. *Given a structured acyclic* SPIN *and an expected vector bound* $\mathbb{E}\mathbb{I}$ *decide whether or not there exists a winning strategy* S *for* $\mathbb{E}\mathbb{I}$ *in* SPIN.

Given a generic state p, e.g., $q = \{p_1 \mapsto 0, p_2 \mapsto 0\}$ (for the sake of brevity, because the other positions are equal to ϵ are not inserted in q) from the diagram in Figure 3. Then, we are interested in the *MNCE* set of transitions and suppose we are in state q. In this case, all the *MNCE* are $\{t_1,t_3\},\{t_1,t_4\},\{t_2,t_3\},\{t_2,t_4\}$. Now, consider a transition set $\overline{T}$ where $\overline{T} = \{t_1\}$; it is clear that $\overline{T}$ is not *MNCE* because it is not maximal as it does not consider a transition that originates from p_2, e.g. can be extended to $\{t_1,t_2\}$. Let's suppose now that we have $\overline{T} = \{t_1,t_4,t_5\}$. In this case, it is not *MNCE* because it contains t_5 that is not enabled. Finally, let's suppose that we have $\overline{T} = \{t_1,t_2,t_4\}$; it is not *MNCE* because it contains t_1,t_2 that have the same origin in p_1. In fact, $incoming(t_1) = incoming(t_2) = p_1$. We now propose an example to clarify what we mean by $Pvariant(\overline{T})$ 11. Consider the *MNCE* $\{t_1,t_3\}$, the $Pvariant$ is $Pvariant(\{t_1,t_3\}) = Pvariant(\{t_1,t_4\}) = \{\{t_1,t_3\},\{t_1,t_4\}\}$. Notice that one variant of $\{t_1,t_3\}$ is $\{t_1,t_3\}$, because one variant of $\overline{T}$ is always $\overline{T}$.

3.3 Dealing with Loops

Despite the whole work being based on acyclic SESE diagrams, we are aware that an important component of such diagrams is missing, i.e., loops. We briefly discuss a simple method for handling loops in our framework, and we are interested in exploring more elegant and theoretical options in future developments. In our framework, the split node v that induces a loop must be a nature one, i.e., $v \notin V_{nature}$, to represent a more general problem. If $v \in V_{choice}$, we restrict our search to strategies that avoid further loop iterations due to the non-negative nature of impacts. We highlight a set $V_{loop} \subseteq V_{nature}$, identifying split nodes encapsulating a loop region. We introduce a function $maxloop : V_{loop} \to \mathbb{N}$ to encode the maximum loop iterations, allowing us to unravel the cyclic structure into an acyclic one. An example of this unraveling is provided in Figure 4, where $maxloop(v) = 2$ results in a chain of 2 copies of v nested into each other. Each additional iteration reduces the contribution to the expected impact by an order of magnitude. This approach is simple to understand and implement and can be parametrized by the user. However, it may result in an exponential increase in size for multiple nested loops, even if $max(Img(maxloop))$ is small. This could affect the feasibility of finding a winning strategy.

We would like to point out that the finite user-parametrized loop unraveling is one of the simplest and most common approaches adopted in the BPMN field [12] in order to deal with loops. For the time being, our tool (see the end of Section 5.2 for further) deals with loops by the method described above, which is still good for contexts that do not put to much emphasis on high numbers of iterations of the loops, for

quick experiments, or for comparison with more sophisticated methods to come.

4 Computational Complexity

In this section, we provide a complexity upper bound for Problem 1, that is PSPACE, by means of the Algorithm 2. The lower bound for the complexity, which is within NP-HARD and PSPACE (NP-HARD lower bound may be provided by a reduction similar to the one presented in Section 5 for k cost game) is still an open problem. First, we have to observe that due to the duration constraints, we may have an exponential number of wait steps if we express such durations in binary. However, this may be easily dealt with if we consider the fact that chains of wait transition by their very definition do not generate possible branching in the computation. Let Q the set of all possible states, we define a function $sat : Q \to Q$ as follows: $sat(q) = \begin{cases} q & \text{if q is saturated} \\ sat(q') & \text{with } q \xrightarrow{t_w} q' \text{otherwise} \end{cases}$.

Basically, the sat function take a state q and returns the next saturated state that can be obtained by q. Now we can provide the definition of saturating transition between two saturated states q, q':

$$q \overset{\overline{T}}{\Rightarrow} q' \textbf{ iff } \begin{array}{r} q \text{ is saturated, } \overline{T} \text{ is an } \mathit{MNCE} \text{ for } q \text{ and either } q \xrightarrow{\overline{T}} q' \text{ with } q' \text{ saturated} \\ \text{or there exists } q'' \text{ such that } q \xrightarrow{\overline{T}} q' \text{ and } sat(q'') = q' \end{array}$$

It is easy to see that a partial strategy S that is defined only on the computations c which end in a saturated state q is as good as a complete strategy since there is only one "move" allowed in a not-saturated state. The decision algorithm for Problem 1 makes use of Algorithm 1, that given a state q computes $sat(q)$ in logarithmic space by means of binary arithmetic.

Our decision procedure relies on the following notion of variant for and *MNCE* .

Definition 11. *Given an MNCE $\overline{T}$ in q an MNCE $\hat{T}$ in q is a* probabilistic variant *of $\overline{T}$ if the following conditions hold: 1. $\overline{T} \cap (T \setminus T_p) = \hat{T} \cap (T \setminus T_p)$; 2. $\forall t \in T_p$ s.t. $t, sw(t) \notin \overline{T}$ we have $t, sw(t) \notin \hat{T}$; 3. $\forall t \in (\hat{T} \cap T_p)$ either $t \in \hat{T}$ or $sw(t) \in \hat{T}$.*

Informally speaking, a probabilistic variant for an *MNCE* $\overline{T}$ in q is still an *MNCE* $\hat{T}$ in q which shares with $\overline{T}$ all the non-probabilistic transitions. Given an *MNCE* $\overline{T}$ in q, we denote with $Pvariant(\overline{T}, q)$ the set of all and only the probabilistic variants of $\overline{T}$ in q. Clearly, we have $\overline{T} \in Pvariant(\overline{T}, q)$.

Algorithm 1: Saturate(q, N)

Input: a state q of a SPIN $N = (PT = P \cup T, T_p, \Delta, I, Pr, D)$
Output: $sat(q)$
1 **if** *there exists $t \in T$ s.t. t is enabled in q* **then**
2   **return** q
3 **let** $\overline{T} \subseteq T$ s.t. for each $t \in \overline{T}$ and for each $p \in incoming(t)$ we have $q(p) \neq \epsilon$
4 **foreach** $t \in \overline{T}$ **do**
5     $k_t \leftarrow \max\{D(p) - q(p) : p \in incoming(t)\}$
6 $k \leftarrow n \min\{k_t : t \in \overline{T}\}$
7 **let** q' s.t. $\forall p \in P$
$$q'(p) = \begin{cases} \epsilon & \text{if } q(p) = \epsilon \\ q(p) + k & \text{otherwise} \end{cases}$$
8 **return** q'

Algorithm 2 employs a non-deterministic approach to ascertain the existence of a viable strategy for a given instance of Problem 1. This is achieved by dynamically enumerating all possible plays, thereby maintaining only a single play in memory at any given moment. This method ensures polynomial memory utilization while providing a comprehensive evaluation of potential strategies.

For the sake of brevity, we do not provide the full proof that Algorithm 2 works in polynomial space. However, we informally provide the key arguments of the proof:

- Algorithm 2 is non-deterministic because it guesses the correct move (if any) at line 9, where $\overline{T} \cap (T \setminus T_p)$ represents the output of the current strategy;
- $Saturate$ operates in LOGSPACE and deals with the binary representation of durations for places;
- Given that N is acyclic, we have that any transition is considered at most for one recursive call to $StrategyExists$. Therefore, the number of nested procedure calls is bounded by $|T|$ since t_w transitions are collapsed via the function $Saturate$;
- In principle, $|Pvariant(\overline{T}, q)|$ (line 10 of Algorithm 2) may be of the order of $2^{|T|}$. However, since only one element $\hat{T} \in Pvariant(\overline{T}, q)$ is needed at a time for updating $\overline{rei}$ via the recursive call in the body of the for loop (line 12 of Algorithm 2), it is possible to set up an enumeration to keep the space polynomial at each step.

Since each play may be represented in polynomial space, we have the following result.

Theorem 1. *Problem 1 is NP-HARD and belongs to the complexity class PSPACE.*

However, our primary objective is to formulate a strategy rather than merely verifying its existence. Consequently, Section 5 is dedicated to addressing the strategy synthesis problem for BPMN+CPI. This section elaborates on the proposed solution, central to the functionality of the effective prototype that we have developed and implemented.
The exact complexity of Problem 1 is still open, we know that it can be proved to be NP-HARD by means of a reduction from the Partition problem introduced in Section 5 for k-cost reachability games.

The NP-HARD lower bound may be achieved by building a game devoid of nature nodes in a way that resembles the one-player restriction of the generalized game proposed in [16], but here Partition is used instead of SAT as the NP-HARD problem we reduce from. In [16], the authors provide a QSAT reduction for the unrestricted case, thus obtaining a PSPACE-HARD lower bound. Such a reduction is not directly applicable in our setting since our winning conditions embrace all possible plays, not a single one. In other words, in [16], a faulty strategy may be detected by witnessing a faulty single play it generates, while in our setting, a faulty strategy may be detected only by considering a subset (possibly all) the plays it generates. For this reason, at this point, we cannot conjecture the exact lower bound for the complexity of Problem 1 without further analysis.

5 Synthesizing Strategies

In this section, we will take advantage of the SPIN translation which has been fully described in Section 3.2. This tree has the foundational semantics of classical Petri Nets [25] for BPMN process. These concepts serve as the mathematical and logical basis for describing a graph-game representation and how the strategy is discovered presented below.

5.1 A k-cost Reachability Game

In this section, we will introduce a graph-game representation for dealing with the synthesis of strategies given a BPMN+CPI diagram $\mathcal{D} = (V, E, E_\top, \mathcal{T})$ which decides whether there exists a strategy that guarantees that the expected impact of a diagram is dominated by a given impact vector bound $\mathbb{I}$.

Definition 12. *A k-cost game board is a tuple $\mathcal{B} = (P = P_\circ \cup P_\Box, p_0, F, C, M)$ such that $p_0 \in P$, $M \subseteq P \times P$, $F \subseteq P$ with $\{(m, m') : m \in F\} = \emptyset$ (i.e, there aren't outgoing edges from F), $C : P \to \mathbb{R}^k$, (P, M) is a directed acyclic graph.*

Definition 13. *Given k-cost game board $\mathcal{B} = (P = P_\circ \cup P_\Box, p_0, F, C, M)$ a strategy is a function $s : P^* \to P$ such that: for every $\rho \in P^*$ we have $(\rho[-1], s(\rho)) \in M$.*

Algorithm 2: Recursive Procedure for Solving Problem 1

Input: a SPIN $N = (P, PT = T \cup T_p, \Delta, I, Pr, D)$ and $\mathbb{EI} \in \mathbb{N}^k$
Output: $ei \in \mathbb{R}^k$ with $ei \le \mathbb{EI}$ if there exists a strategy with residual expected impact ei, and FAIL otherwise

```
1  let q_0 be the initial state of N;
2  return StrategyExists(Saturate (q_0, N), 0^k, 1, EI)
3  Procedure StrategyExists(q, im, cp, rei):
     Data: A saturated state q of N, the value cp of the cumulative probability of the current
           play, im ∈ R^k the current impact for the play, rei ∈ R^k the residual expected impact
           currently available for consumption.
     Result: rei ∈ (R^+)^k if there exists a strategy from the current state q that that has rei
             residual w.r.t. ei, and FAIL otherwise
4    if q is final then
5      if rei ≰ 0^k then
6        FAIL
7      return rei − (cp · im)
8    let T̄ an MNCE for q′
9    rei‾ ← rei
10   foreach T̂ ∈ Pvariant(T̄, q) do
11     let q′ s.t. q −T̂→ q̂′
12     rei‾ ← StrategyExist(Saturate(q’, N), im + Σ_{t∈T̂} I(t), cp · Π_{t∈T̂∩T_p} Pr(t), rei‾)
13     if rei ≱ 0^k then
14       FAIL
15   return rei‾
```

Definition 14. *Given a k-cost game board $\mathcal{B} = (P = P_\circ \cup P_\Box, p_0, F, C, M)$ and a strategy s, a successful play $\rho \in P^*$ is generated by s in $\mathcal{B}$ if and only if: (i) $\rho[0] = p_0$; (ii) $\rho[-1] \in F$; (iii) for every $0 < i < |\rho|$ if $\rho[i-1] \in P_\circ$ then $\rho[i] = s(\rho[0 : i])$.*

Let P_s^* be the set of all the possible plays generated by s.

Definition 15. *Given s we say that P_s^* is closed if for each $\rho \in P_s^*$ and for each $0 \le i < |\rho| - 1$ such that $\rho[i] \in P_\Box$ then for each $(\rho[i], p) \in M$ we have that there exists $\rho' \in P_s^*$ with $\rho'[0 : i] = \rho[0 : i]$ and $\rho'[i+1] = p$.*

Given a P_s^* we let $final(P_s^*)$ the set $final(P_s^*) = \{\rho[-1] : \rho \in P_s^*\}$.

Problem 2. *Given a k-cost game board $\mathcal{B} = (P = P_\circ \cup P_\Box, p_0, F, C, M)$ and a cost $c \in \mathbb{R}^k$ determine whether or not there exists a strategy s for which P_s^* is closed and $\sum_{p \in final(P_s^*)} C(p) \le c$.*

A strategy s is positional if and only if for every $\rho, \rho' \in P^*$ we have that $\rho[-1] = \rho'[-1]$ implies $s(\rho) = s(\rho')$. For the purpose of our game, w.l.o.g. a positional strategy may be redefined as $s : P_\circ \to P$.

Problem 3. *Given a k-cost game board $\mathcal{B} = (P = P_\circ \cup P_\Box, p_0, F, C, M)$ and a cost $c \in \mathbb{R}^k$ determine whether or not there exists a* ***positional*** *strategy s for which P_s^* is closed and $\sum_{p \in final(P_s^*)} C(p) \le c$.*

Theorem 2. *For every k-cost game board and each cost vector $c \in \mathbb{R}^k$ we have that $(\mathcal{B}, c)$ is a positive instance of Problem 2 if and only if $(\mathcal{B}, c)$ is a positive instance of Problem 3*

It is easy to prove that Problem 3 belongs to the complexity class NP, by simply provide a succinct certificate, that is, given an instance $(\mathcal{B} = (P = P_\circ \cup P_\Box, M, p_0, F, \mathcal{C}), c)$ of Problem 3 guess a subset $M' \subseteq M$ such that $\{(p, p') \in M : p \in P_\Box\} \subseteq M'$ and for each $p \in P_\circ$ either $\{(p, p') \in M\} = \emptyset$ or there exists a unique edge $(p, p') \in M'$. Then, let F' be the subset of F reachable from p_0 in the M'-induced sub-graph $(P_\circ \cup P_\Box, M')$ we have that M' is a solution if and only if $\sum_{p \in F'} \mathcal{C}(p) \leq c$. The NP-HARD lower bound for Problem 3, and thus for Problem 2, is proved by a reduction from the following NP-HARD problem.

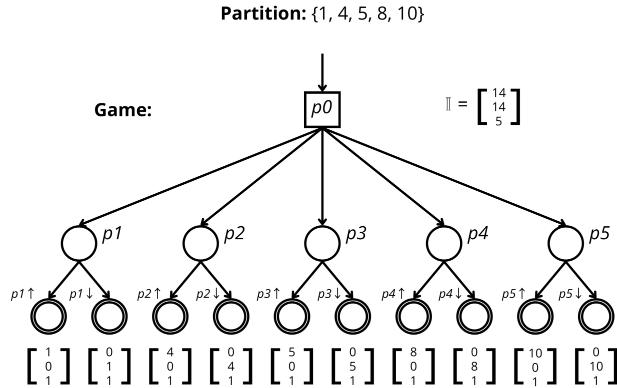


Figure 5: Reduction from Partition to k-cost game Problem.

Problem 4. (Distinct Partition) *Given a set of natural numbers $S = \{n_1, \ldots, n_m\}$ decide whether or not there exists a partition (S_1, S_2) of S such that $\sum_{n \in S_1} n = \sum_{n \in S_2} n$.*

As formulated by Korf in [21], Problem 4 is actually NP-complete. We recall this in Theorem 3.

Theorem 3. *Distinct Partition (Problem 4) is NP-Complete [21].*

There exists a simple LOG-SPACE reduction from Distinct Partition to Problem 3, and thus to Problem 2, for $k \geq 3$. The reduction is very simple, it suffices to transform the distinct partition problem $S = \{n_1, \ldots, n_m\}$ into an instance of Problem 2 $(\mathcal{B}_S = (P = P_\circ \cup P_\Box, M, p_0, F, \mathcal{C}), c_S)$ as follows:

1. $P_\circ = \{p^i, p^i_\uparrow, p^i_\downarrow : 1 \leq i \leq m\}$,
2. $P_\Box = \{p_0\}$,
3. $M = \{(p_0, p^i) : 1 \leq i \leq m\} \cup \{(p^i, p^i_\uparrow), (p^i, p^i_\downarrow) : 1 \leq i \leq m\}$,
4. $F = \{p^i_\downarrow, p^i_\uparrow : 1 \leq i \leq m\}$,
5. $\mathcal{C}(p^i_\uparrow) = [n_i, 0, 1]$ and $\mathcal{C}(p^i_\downarrow) = [0, n_i, 1]$ for each $1 \leq i \leq m$,
6. $c_S = \left[\frac{\sum_1^m n_i}{2}, \frac{\sum_1^m n_i}{2}, m \right]$.

An example of the proposed reduction is given in Figure 5. It is easy to prove that $(\mathcal{B}_S, c_S)$ is a positive instance of Problem 2 if and only if S is a positive instance of the distinct partition problem.

Theorem 4. *Problem 3 and Problem 2 for $k \geq 3$ are NP-Complete problems.*

5.2 From BPMN+CPI to k-cost Reachability Game

We conclude this section by providing the direct translation from an instance $(N, \mathbb{EI})$ of Problem 1 into a k-cost game $(\mathcal{B}, \mathbb{EI})$, which admits a solution if and only if the problem $(N, \mathbb{EI})$ admits a solution.

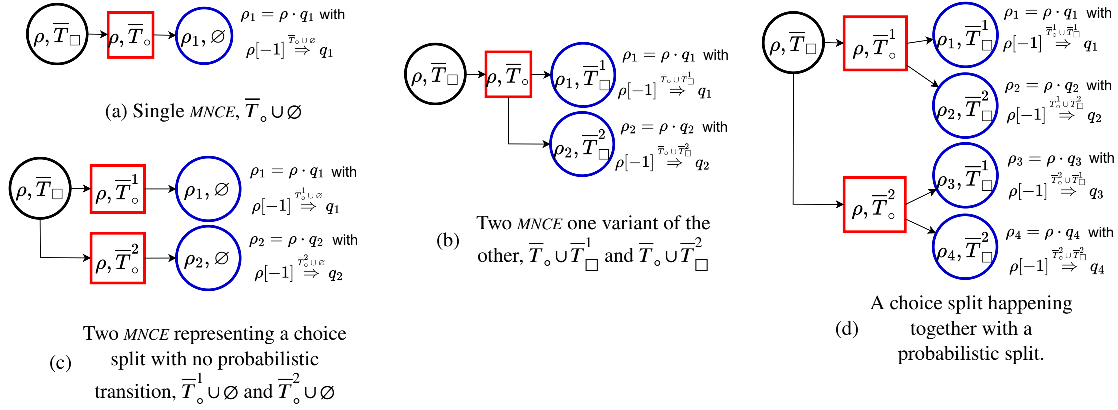


Figure 6: Different scenarios involving at most one choice and at least one probabilistic split.

Moreover, if $(\mathcal{B}, \mathbb{E}\mathbb{I})$ admits a solution, i.e., it is a positive instance of Problem 2, such a solution will effectively represent a strategy for the original problem.

Before providing this translation, we introduce a couple of useful definitions. Given an *MNCE* $\overline{T}$ for a state q, we define two sets: $\overline{T}_\square = \overline{T} \cap T_p$ and $\overline{T}_\circ = \overline{T} \setminus T_\square$. Additionally, for any $\overline{T}_\square \subseteq T_p$, let $Pr(\overline{T}_\square) = \prod_{t \in \overline{T}_\square} Pr(t)$; clearly, $Pr(\emptyset) = 1$. For any $\overline{T}_* \subseteq T$, let $\mathcal{I}(\overline{T}_*) = \sum_{t \in \overline{T}_*} \mathcal{I}(t)$; clearly, $\mathcal{I}(\emptyset) = 0$. Finally, let Q be the set of all possible saturated states on N and C^+_Q be the set of all possible non-empty combinations of elements in Q. Given a combination $\rho \in C^+_Q$, we denote its last element as $\rho[-1]$.
Given an instance $(N, \mathbb{E}\mathbb{I})$ of we define a k-cost game board $\mathcal{B}_N = (S = S_\circ \cup S_\square, s_0, F, C, M)$ as follows:

$$S_\circ = \{(\rho, \overline{T}_\square) \in C^+_Q \times 2^{T_p} \, : \, \rho[-1] \; is \; saturated\}, \quad S_\square = \left\{ (\rho, \overline{T}_\circ) \, : \, \begin{array}{c} \rho \in C^+_Q, \text{ there exists } \overline{T}_\square \subseteq T_p s.t. \\ \overline{T}_\circ \cup \overline{T}_\square \; is \; an \; MNCE \; for \; \rho[-1] \end{array} \right\},$$

$$s_0 = (sat(q_0), \emptyset), \qquad F = \{(\rho, \overline{T}_\square) \in C^+_Q \times 2^{T_p} \, : \, \rho[-1] = q_f\},$$

and $M = \{((\rho, \overline{T}_\square), (\rho, \overline{T}_\circ)) \, : \, (\rho, \overline{T}_\square) \in P_\circ, (\rho, P_\square) \in P_\square\} \cup \{((\rho, \overline{T}_\circ), (\rho q, \overline{T}_\square)) \, : \, \rho[-1] \overset{\overline{T}_\circ \cup \overline{T}_\square}{\Rightarrow} q\}$.

Graphical examples of how the relation M is build in the case when the *MNCE* $\overline{T} = \overline{T}_\circ \cup \overline{T}_\square$ satisfies $|\overline{T}_\circ| \leq 1$ and $|\overline{T}_\square| \leq 1$ are provided in Figure 6.
Lastly, for the cost function, let M^* denote the reflexive and transitive closure of M. For any $s \in F$, the cost function $C(s)$ is defined as:

$$C(s) = \left(\prod_{(\rho, \overline{T}_\square) \in S_\circ : (s_0, (\rho, \overline{T}_\square)), ((\rho, \overline{T}_\square), s) \in M^*} Pr\left(\overline{T}_\square\right) \right) \cdot \left(\sum_{(\rho, \overline{T}_*) \in S_\circ \cup S_\square : (s_0, (\rho, \overline{T}_*)), ((\rho, \overline{T}_*), s) \in M^*} \mathcal{I}\left(\overline{T}_*\right) \right)$$

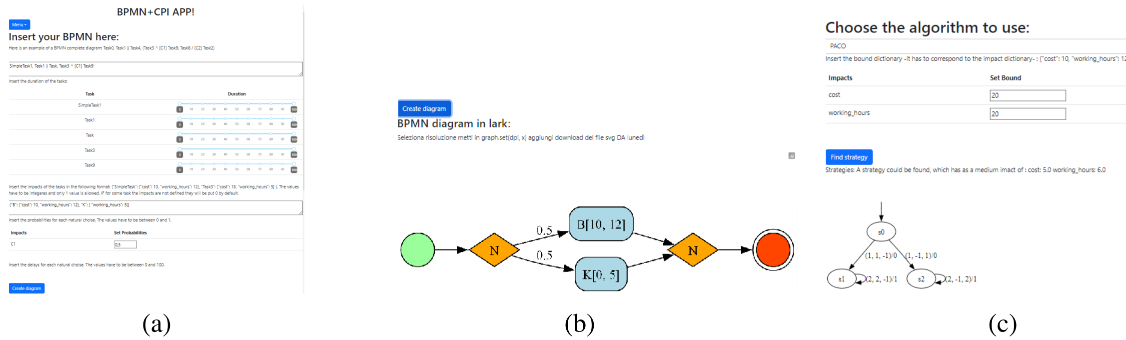
The formula described assigns to each final state $s \in F$ the contribution to the expected impact generated by paths terminating at s. Now, as a final measure, we resolve the k-cost game by selecting [1] a

[1]This is implemented by evaluating all possible subsets $F' \subseteq F$ such that $\sum_{s \in F'} C(p) \leq \mathbb{E}\mathbb{I}$ and for each $s' \in F \setminus F'$, $\sum_{s \in F' \cup \{s'\}} C(s) > \mathbb{E}\mathbb{I}$. We consider only the maximal admissible subsets of F, as they can "attract" the initial state if and only if at least one of their subsets does.

subset $F' \subseteq F$ such that the total expected impact satisfies: $\sum_{s \in F'} C(s) \leq \mathbb{E}\mathbb{I}$

We employ the standard attractor procedure as described in [27], initiating with $Attr^0 = F'$ in $\mathcal{B}$. A positive outcome, along with the strategy formulated by the attractor procedure, is confirmed if there exists $k \in \mathbb{N}$ such that $s_0 \in Attr^k$. While the attractor procedure itself runs in polynomial time, approximately $\mathcal{O}(nm)$ for a graph with n nodes and m edges, the non-deterministic selection of a candidate $Attr^0$ from the set of final states remains computationally intensive, since the number of final states may be exponential in the size of SPIN thus the above procedure for synthetizing a strategy operates in NEXPTIME.

Implementation The algorithm described in this section, known as *PACO*, has been developed and is accessible at `https://github.com/ansimonetti/PACO`. *PACO* is designed as a Dash App [20]. The process is written in Lark syntax [1], with all choices, probabilities, and impacts clearly defined, as visible in Figure 7a and printed using Graphviz [3] and PyDot [7], as shown in Figure 7b. A specific section is dedicated to defining the expected impacts vector. Subsequently, the AALpy automata [24] is employed to provide a strategy, as previously described, if one exists. If one is found, the algorithm returns it together with the associated impact factors. Moreover, it prints the tree associated with the strategy, indicating which tasks have to be done to complete the process within the bound vector as shown in Figure 7c.



(a) (b) (c)

Figure 7: Example of using our Dash App: defining the BPMN in our Dah App 7a, print the BPMN using Lark 7b and example of founded strategy using *PACO* 7c

6 Conclusion

In this study, we developed a BPMN extension, denoted as BPMN+CPI, designed to handle execution in the presence of impacts, probabilistic splits, and choices. The semantics for this extension were formulated using an enriched version of Petri Nets, namely, SPIN. The primary objective of this work was to create a system capable of informing users about the existence of a strategy for a given process and user-defined thresholds. This involves determining whether there is a controller capable of executing each step of the process while ensuring that the expected value of each resource across repeated process instances remains within the predefined thresholds.

First, we proved that the associated decision problem, i.e., determining if such a controller exists, belongs to the complexity class PSPACE. Then, we provided an effective method for building the controller by modifying classical reachability games over graphs. Based on these theoretical results, we implemented a tool capable of determining the existence of a strategy given a BPMN+CPI process and a given

threshold $\mathbb{E}\mathbb{I}$. This tool is currently under development, but a working prototype is available online for the benefit of the community.

For future work, we envision two promising extensions. The first, theoretical, aims to deal with loops in the workflow in a non-approximated fashion and to propose alternative algorithms for solving the problem, potentially closing the complexity gap, which currently stands between PSPACE and NP. The second, more practical extension, focuses on better representing the obtained strategy by integrating it into the choice gateway of the BPMN+CPI, for instance, representing decisions with a set of inequalities involving intervals of values for the impact components observed in specific choice nodes.

Acknowledgments This work has been carried out while Emanuele Chini was enrolled in the Italian National Doctorate on Artificial Intelligence run by Sapienza University of Rome in collaboration with the University of Verona.

References

[1] (2024): *Lark - Parsing Library & Toolkit*. Available at `https://github.com/lark-parser/lark`. Accessed: 2024-04-20.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Piotr Hofman, Richard Mayr, K. Narayan Kumar & Patrick Totzke (2014): *Infinite-state energy games*. In Thomas A. Henzinger & Dale Miller, editors: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, ACM, pp. 7:1–7:10, doi:10.1145/2603088.2603100.

[3] Sebastian Bank (2024): *Graphviz*. Available at `https://github.com/xflr6/graphviz`. Accessed: 2024-04-20.

[4] Gerd Behrmann, Agnes Cougnard, Alexandre David, Emmanuel Fleury, Kim G Larsen & Didier Lime (2007): *UPPAAL-Tiga: Time for Playing Games! (Tool Paper)*. In: *Computer Aided Verification: 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007. Proceedings 19*, Springer, pp. 121–125, doi:10.1007/978-3-540-73368-3_14.

[5] Cinzia Cappiello, Maria Grazia Fugini, GR Gangadharan, Alexandre Mello Ferreira, Barbara Pernici & Pierluigi Plebani (2010): *First-step toward energy-aware adaptive business processes*. In: *On the Move to Meaningful Internet Systems: OTM 2010 Workshops: Confederated International Workshops and Posters: International Workshops: AVYTAT, ADI, DATAVIEW, EI2N, ISDE, MONET, OnToContent, ORM, P2P-CDVE, SeDeS, SWWS and OTMA. Hersonissos, Crete, Greece, October 25-29, 2010. Proceedings*, Springer, pp. 6–7, doi:10.1007/978-3-642-16961-8_4.

[6] J. Carmona, J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno & A. Yakovlev (2008): *A Symbolic Algorithm for the Synthesis of Bounded Petri Nets*. In: *Proceedings of the 29th International Conference on Applications and Theory of Petri Nets*, PETRI NETS '08, Springer-Verlag, Berlin, Heidelberg, p. 92–111, doi:10.1007/978-3-540-68746-7_10.

[7] Ero Carrera (2024): *Pydot*. Available at `https://github.com/pydot/pydot`. Accessed: 2024-04-20.

[8] Krishnendu Chatterjee & Laurent Doyen (2011): *Energy and mean-payoff parity Markov decision processes*. In: *International Symposium on Mathematical Foundations of Computer Science*, Springer, pp. 206–218, doi:10.1007/978-3-642-22993-0_21.

[9] Carlo Combi, Barbara Oliboni & Francesca Zerbato (2019): *A modular approach to the specification and management of time duration constraints in BPMN*. *Information Systems* 84, pp. 111–144, doi:10.1016/j.is.2019.04.010.

[10] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis & Jakob Haahr Taankvist (2015): *Uppaal Stratego*. In Christel Baier & Cesare Tinelli, editors: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 206–211, doi:10.1007/978-3-662-46681-0_16.

[11] Massimiliano De Leoni, Wil MP Van Der Aalst & Boudewijn F Van Dongen (2012): *Data-and resource-aware conformance checking of business processes*. In: *Business Information Systems: 15th International Conference, BIS 2012, Vilnius, Lithuania, May 21-23, 2012. Proceedings 15*, Springer, pp. 48–59, doi:10.1007/978-3-642-30359-3_5.

[12] Remco M Dijkman, Marlon Dumas & Chun Ouyang (2008): *Semantics and analysis of business process models in BPMN*. *Information and Software technology* 50(12), pp. 1281–1294, doi:10.1016/j.infsof.2008.02.006.

[13] Marlon Dumas, Luciano García-Bañuelos & Artem Polyvyanyy (2010): *Unraveling Unstructured Process Models*. In Jan Mendling, Matthias Weidlich & Mathias Weske, editors: *Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings*, *Lecture Notes in Business Information Processing* 67, Springer, pp. 1–7, doi:10.1007/978-3-642-16298-5_1.

[14] Francisco Durán, Camilo Rocha & Gwen Salaün (2018): *Stochastic analysis of BPMN with time in rewriting logic*. *Science of Computer Programming* 168, pp. 1–17, doi:10.1016/j.scico.2018.08.007.

[15] Jorge Fernandes, João Reis, Nuno Melão, Leonor Teixeira & Marlene Amorim (2021): *The role of Industry 4.0 and BPMN in the arise of condition-based and predictive maintenance: A case study in the automotive industry*. *Applied Sciences* 11(8), p. 3438, doi:10.3390/app11083438.

[16] Nathanaël Fijalkow & Florian Horn (2010): *The surprizing complexity of generalized reachability games*. *arXiv preprint arXiv:1010.2420*, doi:10.48550/arXiv.1010.2420.

[17] Jerzy Filar & Koos Vrieze (2012): *Competitive Markov decision processes*. Springer Science & Business Media, doi:10.1007/978-1-4612-4054-9.

[18] Christoph Haase, Stephan Kreutzer, Joël Ouaknine & James Worrell (2009): *Reachability in Succinct and Parametric One-Counter Automata*. pp. 369–383, doi:10.1007/978-3-642-04081-8_25.

[19] Luke Herbert & Robin Sharp (2013): *Precise quantitative analysis of probabilistic business process model and notation workflows*. *Journal of Computing and Information Science in Engineering* 13(1), p. 011007, doi:10.1115/1.4023362.

[20] Plotly Technologies Inc. (2024): *Dash*. Available at `https://dash.plotly.com/`. Accessed: 2024-04-20.

[21] Richard E. Korf (1998): *A complete anytime algorithm for number partitioning*. *Artificial Intelligence* 106(2), pp. 181–203, doi:10.1016/S0004-3702(98)00086-1.

[22] M. Kwiatkowska, G. Norman & D. Parker (2011): *PRISM 4.0: Verification of Probabilistic Real-time Systems*. In G. Gopalakrishnan & S. Qadeer, editors: *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, *LNCS* 6806, Springer, pp. 585–591, doi:10.1007/978-3-642-22110-1_47.

[23] Matteo Magnani & Danilo Montesi (2007): *BPMN: How Much Does It Cost? An Incremental Approach*. In Gustavo Alonso, Peter Dadam & Michael Rosemann, editors: *Business Process Management*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 80–87, doi:10.1007/978-3-540-75183-0_6.

[24] Edi Muškardin, Bernhard Aichernig, Ingo Pill, Andrea Pferscher & Martin Tappler (2022): *AALpy: an active automata learning library*. *Innovations in Systems and Software Engineering* 18, pp. 1–10, doi:10.1007/s11334-022-00449-3.

[25] James L Peterson (1977): *Petri nets*. *ACM Computing Surveys (CSUR)* 9(3), pp. 223–252, doi:10.1145/356698.356702.

[26] Luise Pufahl, Francesca Zerbato, Barbara Weber & Ingo Weber (2022): *BPMN in healthcare: Challenges and best practices*. *Information Systems* 107, p. 102013, doi:10.1016/j.is.2022.102013.

[27] Wolfgang Thomas (1995): *On the synthesis of strategies in infinite games*. In: *Annual Symposium on Theoretical Aspects of Computer Science*, Springer, pp. 1–13, doi:10.1007/3-540-59042-0_57.

[28] Uri Zwick & Mike Paterson (1996): *The complexity of mean payoff games on graphs*. *Theoretical Computer Science* 158(1), pp. 343–359, doi:10.1016/0304-3975(95)00188-3.

Towards the Usage of Window Counting Constraints in the Synthesis of Reactive Systems to Reduce State Space Explosion

Linda Feeken
German Aerospace Center (DLR)
Oldenburg, Germany
linda.feeken@dlr.de

Martin Fränzle
Carl von Ossietzky Universität Oldenburg
Oldenburg, Germany
fraenzle@informatik.uni-oldenburg.de

The synthesis of reactive systems aims for the automated construction of strategies for systems that interact with their environment. Whereas the synthesis approach has the potential to change the development of reactive systems significantly due to the avoidance of manual implementation, it still suffers from a lack of efficient synthesis algorithms for many application scenarios. The translation of the system specification into an automaton that allows for strategy construction is nonelementary in the length of the specification in S1S and double exponential for LTL, raising the need of highly specialized algorithms. In this paper, we present an approach on how to reduce this state space explosion in the construction of this automaton by exploiting a monotony property of specifications. For this, we introduce window counting constraints that allow for step-wise refinement or abstraction of specifications. In an iterating synthesis procedure, those window counting constraints are used to construct automata representing over- or under-approximations (depending on the counting constraint) of constraint-compliant behavior. Analysis results on winning regions of previous iterations are used to reduce the size of the next automaton, leading to an overall reduction of the state space explosion extend. We present the implementation results of the iterated synthesis for a zero-sum game setting as proof of concept. Furthermore, we discuss the current limitations of the approach in a zero-sum setting and sketch future work in non-zero-sum settings.

1 Introduction

The automated translation of a system specification into its implementation is one of the most challenging problems in formal methods. Such a synthesis offers great potential in the development of new systems by significantly reducing the need for manual work in the engineering process. In this paper, we focus on synthesis for reactive systems, i.e. systems that are influenced by and interact with their environment. This interaction can be modeled as a game, in which the system tries to play according to its specification, whereas the moves of the environment can potentially impede the system from reaching its goal. Since the interaction between system and environment is typically of long-lasting nature without predefined end date, the game is infinite in the sense that a play of the game has infinite duration, while the arena, modeled as a graph, has finitely many states. The players play by moving a token from one state of the arena to the next. The player whose turn it is decides which of the outgoing transitions of the current state is chosen. A well-known type of game is the safety game: The system wins a play if it can avoid to reach predefined unsafe states. Otherwise, the environment wins. A player has a winning strategy, if it wins against all possible behavior of the other player. For two-player safety games on finite graphs, there always exists a winning strategy for one of the players and this winning strategy can be computed [1], [20]. However, the efficient computation of winning strategies (not only in the case of safety games) is still an open challenge in the synthesis of reactive systems. A common synthesis

A. Achilleos and A. Francalanza (Eds.): Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 53–69, doi:10.4204/EPTCS.409.8

© L. Feeken & M. Fränzle
This work is licensed under the Creative Commons Attribution License.

approach is to generate a deterministic word automaton as game graph from specifications written as Linear Temporal Logic (LTL) formulae. Finally, a strategy that is winning in the game is calculated. By construction, the strategy automatically satisfies the specification. Unfortunately, the construction of the deterministic word automaton leads to an automaton with a number of states that is double-exponential in the length of the specification [17], making the whole strategy synthesis unfeasible for many applications. For avoiding the most expensive part of the synthesis procedure, there exist synthesis algorithms that start with a subset of the specification language LTL, such that it is possible to construct the game graph in a more efficient way. One example for that is the usage of the LTL subclass Generalized Reactivity(1) (GR(1)), which allows to construct and solve the game in time $O(N^3)$ with N being the size of the state space [16]. While GR(1) is expressive enough for the specification of many systems [13], some specifications that do not fall into GR(1) remain unconsidered. For example, Maoz and Ringert mention the consideration of synthesis with counting patterns as future work in [13], but to the best of our knowledge, this is not yet done.

In this paper, we deal with the request for efficient synthesis for some types of counting patterns as part of the system specification and present the idea of iterated synthesis for such games. We call the considered counting patterns "window counting constraints". These are of the form

"The system plays action act at most k times out of l of its own moves."

with parameters $k,l \in \mathbb{N}$, $k \leq l$. The "at most" can also be replaced by "at least". Such constraints arise naturally when the desired behavior of systems includes reoccurring elements. For example, an automated guided vehicle on a factory floor might need to charge its battery in at least two out of ten moves to avoid to get empty batteries on an exit path. The term "window" in the constraint type name emphasizes the relation of those specifications to sliding windows in data stream monitoring [15]. For the sake of better readability, we also call them counting constraints in short.

We avoid the direct full translation of the specifications into a graph and instead focus on the following two observations: (1) It is possible to influence how hard it is to satisfy a counting constraint by varying the parameters k,l included in the counting constraints. More precisely, the (non-)existence of a strategy that fulfills the specification in a game with a set of counting constraints allows to make statements about the (non-)existence of such a strategy in a game with a set of counting constraints with varied parameters. (2) The values in the counting constraints influence the scale of the game graph that encodes all information given by the constraints. The greater k and l, the greater is the graph. Consequently, the values influence how much computational power and/or memory is needed in order to synthesize a winning strategy.

Combining these observations, the presented approach can be summarized as follows: Consider a two-player game graph and some specifications in the form of counting constraints. For solving the synthesis problem of finding a strategy for the system, such that the counting constraints are fulfilled, start with a subset of counting constraints that result in a small game graph or a trivially winnable game. Calculate winning strategies (if existent) and check what the (non-)existence of a winning strategy means for a game with refined/relaxed (depending on the constraints) constraints. This information shall give hints on which parts of the game with adapted values in the counting constraints are worth to investigate in the next iteration step and which parts of the game graph can then be neglected, leading to a reduction in the state space. In each iteration, the set of considered counting constraints converges more to the game of interest. Although the size of the game graphs may increase in each iteration, the gained state space reduction leads to a synthesis algorithm more efficient than when considering the game of interest as a whole from the beginning. The motivation for starting with a game graph accompanied with counting constraints instead of a pure set of specifications comes from the robotic domain. In many applications,

automated guided vehicles are moving in specified areas (like a factory floor). Modeling the setting as a game graph in which states encode the position of systems arises naturally. However, the initial game graph can also represent the winning region of a priorly solved safety game [14], [23], [10] that shall be accompanied with additional counting constraints. This way, it is possible to use the presented approach for safety games. Note that the safety game with neglected counting constraints is usually significantly smaller and hence easier to solve than the game with already included counting constraints.

This paper is structured as follows. In Section 2, related work in the field of synthesis for reactive systems is presented, focusing on the challenge of constructing efficient algorithms. After summarizing concepts and notations required to formulate the game, Section 3 provides the definition of a game with counting constraints. In Section 4, we present the idea of iterated synthesis with counting constraints, including the results of a non-optimized implementation as proof-of-concept. The presented algorithm delivers promising results, but suffers of limitations that are targeted by our current research work. We discuss planned directions of future work in Section 5. Section 6 concludes the paper.

2 Related Work

In 1957, Church formulated the Synthesis Problem as finding finite-memory procedures to transform an infinite sequence of input data into an infinite sequence of output data, such that the relation between input and output satisfies given specifications [3], [21]. Around a decade later, Büchi and Landweber showed the decidability of the problem [1]. However, the algorithmic complexity of synthesis algorithms remains a challenge. The translation of specifications from monadic second-order logic of one successor (S1S) into a Büchi automaton as part of the synthesis procedure is nonelementary in the length of specifications [19]. This indicates that it is not possible to construct a universally (or an in all cases) efficient synthesis algorithm that can handle complete S1S specifications. For specifications expressible in Linear Temporal Logic (LTL), the problem is 2EXPTIME-complete [17].
Acknowledging the absence of a generally low-complexity synthesis algorithm for arbitrary S1S/LTL specifications, the literature presents three primary approaches [8]. (1) The first approach restricts the scope of considered specifications for synthesis to less expressive logics. Here, the structure of the considered specifications is used to reduce the synthesis complexity. (2) The second one is tackling the internal representation of the problem. Solutions following this approach are often aiming for algorithms with in average good runtime. In this approach, it suffices if most systems can be synthesized with acceptable resources (memory, computational time), while the existence of corner cases with worst-case complexity is accepted. (3) The third approach focuses on the output of the problem, the implementation. The size of the implementation is restricted, such that only small implementations are accepted as solutions of the synthesis problem. The rationale behind this is that small and hence less complicated implementations often exist for applications. Such solutions are often easier (that is, with less computational time) identifiable than bigger (complicated) implementations, if it is possible to steer the algorithm towards small solutions. Synthesis algorithms can follow more than one of those approaches.

A well studied class of specifications for approach (1) is General Reactivity of Rank 1 (GR1), a fragment of LTL for which there are symbolic synthesis algorithms that are polynomial in the size of the state space of the design [16]. Examples for other specification classes for which efficient solutions of the synthesis problem are investigated are Safety LTL [22], Metric Temporal Logic with a Bounded Horizon [12] and Extended Bounded Response LTL [4].

Following approach (2), Kupferman and Vardi developed a synthesis method that does not require the costly determinization of non-deterministic Büchi automata representing the specification [11], which is

the most complex part in many synthesis algorithms. Other synthesis algorithms rely for instance on symbolic synthesis to represent sets of states of a game graph in a compact matter via antichains [6], [7], binary decision diagrams [5] and LTL fragements [4].

The work by Schewe and Finkbeiner presents a synthesis algorithm that employs bounded synthesis as approach (3). Their method uses translation of LTL specifications into sequences of safety tree automata, in order to constraint the size of the implementation [18]. "Lazy synthesis", in which an SMT solver is used to construct potential implementations for an incomplete constraint system, extends the system only if required [9].

The synthesis algorithm presented in this paper includes elements of approaches (2) and (3). We avoid the full construction of an automaton representing the specifications by starting with a small specification that is successively enlarged. In each step, the size of the resulting automaton is reduced (if possible). The procedure stops, if a winning strategy can already be found in some intermediate step, leading to small solutions. However, it is not possible to restrict the size of the implementation directly as commonly done in bounded synthesis.

The general idea is inspired by the work of Chen et al. on games with delay. In this work, one player only receives information on the moves of the environment with a delay of $k \in \mathbb{N}$ turns. For strategy synthesis, the delay is incrementally enlarged from zero to k with a graph reduction step after each iteration step [2].

3 Games with Counting Constraints

This section introduces games with counting constraints after repeating standard definitions for two-player games that are needed to formalize the presented game.

Definition 3.1 (Two-player game graph)**.** *A **two-player finite-state game graph** is of the form $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow)$ where S is a finite (non-empty) set of states, S_{EGO}, S_{ALTER} define a partition of S, $s_0 \in S_{EGO}$ is the initial state, Σ_{EGO} is a finite alphabet of actions for player EGO, Σ_{ALTER} is a finite alphabet of actions for player ALTER and $\rightarrow \subseteq S \times (\Sigma_{EGO} \cup \Sigma_{ALTER}) \times S$ is a set of labeled transitions satisfying the following four conditions:*

- *Bipartition: For each $(s, \sigma, s') \in \rightarrow$ holds either (1) $s \in S_{EGO}$ and $s' \in S_{ALTER}$ or (2) $s \in S_{ALTER}$ and $s' \in S_{EGO}$.*
- *Absence of deadlock: For each $s \in S$ there exists $\sigma \in \Sigma_{EGO} \cup \Sigma_{ALTER}$ and $s' \in S$, such that $(s, \sigma, s') \in \rightarrow$.*
- *Alphabet restriction on actions: For a player $p \in \{EGO, ALTER\}$ holds: If $(s, \sigma, s') \in \rightarrow$ with $s \in S_p$, then $\sigma \in \Sigma_p$.*
- *Determinacy of moves: For $p \in \{EGO, ALTER\}$ and $\sigma \in \Sigma_{EGO} \cup \Sigma_{ALTER}$ holds: if $s \in S_p$ and $(s, \sigma, s'), (s, \sigma, s'') \in \rightarrow$, then $s' = s''$.*

Such a game graph, also referred to as "arena", encodes a game between the two players EGO and $ALTER$. For $p \in \{EGO, ALTER\}$ the set of states S_p contains the states where it is the turn of player p to perform an action, also called "p controls s". Due to the bipartition and alphabetic restriction on actions, the game is "turn-based", i.e. the two players alternate between choosing one of the possible actions. Since the game graph does not contain deadlocks, it results in an infinite sequence of states and actions, called an infinite play.

Definition 3.2 (Infinite play)**.** *Let $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow)$ be a two-player game graph. An* ***infinite play*** *on G is an infinite sequence $\pi = (\pi_i\sigma_i)_{i\in\mathbb{N}_0} = \pi_0\sigma_0\pi_1\sigma_1\dots$ with $\pi_0 = s_0$ and $\pi_i\sigma_i\pi_{i+1} \in\rightarrow$ for all $i \in \mathbb{N}_0$. $\Pi(G)$ denotes the set of all infinite plays on G.*

In such an infinite play, the two players play against (or in case of collaborative games: with) each other. Players can have strategies that determine how they react in each step of the play.

Definition 3.3 (Strategy)**.** *Let $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow)$ be a two-player game graph.*

- *For a play $\pi = (\pi_i\sigma_i)_{i\in\mathbb{N}_0}$, a* ***prefix*** *of π up to position n is denoted by $\pi(n) = \pi_0\sigma_0\pi_1\dots\pi_{n-1}\sigma_{n-1}\pi_n$. The length of $\pi(n)$, denoted by $|\pi(n)|$, is $n+1$. The last state π_n of $\pi(n)$ is called the* ***tail*** *of the prefix $\pi(n)$, denoted by $Tail(\pi(n))$. The set of all prefixes of plays in the game graph G is $Pref(G)$.*
- *For a player $p \in \{EGO, ALTER\}$ and a game graph G, the set of all prefixes that end in a state controlled by p is $Pref_p(G) := \{\pi(n) \in Pref(G) \,|\, Tail(\pi(n)) \in S_p\}$.*
- *A* ***strategy*** *for a player $p \in \{EGO, ALTER\}$ in the game graph G is a mapping $\varphi : Pref_p(G) \longrightarrow 2^{\Sigma_p}$, such that for all prefixes $\pi(n) \in Pref_p(G)$ and all $\sigma \in \varphi(\pi(n))$ there exist a state $s \in S \setminus S_p$ and a transition $(Tail(\pi(n)), \sigma, s) \in\rightarrow$.*
- *The* ***outcome*** *$O(G, \varphi)$ of a strategy φ of $p \in \{EGO, ALTER\}$ in the game graph G is the set of all possible plays when player p follows the strategy φ and the other player plays arbitrary, i.e. $O(G, \varphi) := \{\pi = (\pi_i\sigma_i)_{i\in\mathbb{N}_0} \in \Pi(G) \,|\, \forall i \in \mathbb{N}_0 : \sigma_{2i} \in \varphi(\pi(2i))$ if $s_0 \in S_p$ and $\sigma_{2i+1} \in \varphi(\pi(2i+1))$ otherwise$\}$.*

In a safety game, the player *EGO* wins, if it has a strategy that guarantees to never visit predefined unsafe states. The environment, on the other hand, wins if an unsafe state is reached. Hence, each play of a two-player safety game always has exactly one winner and one loser. Games with this property are called zero sum games.

Definition 3.4 (Safety Game)**.** *A* ***safety game*** *$G = (G', \mathscr{U})$ consists of a two-player finite-state game graph $G' = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow)$ and a set $\mathscr{U} \subseteq S$ of unsafe states.*
Player EGO has a ***winning strategy*** *φ on G, if φ is a strategy on G', such that none of the plays in $O(G', \varphi)$ include a state $u \in \mathscr{U}$. The* ***winning region*** *of G is defined as the set of states $\tilde{S} \subseteq S$, where EGO can win from any state $s \in \tilde{S}$. This means EGO has a winning strategy in the game $\tilde{G}_s$ with $\tilde{G}_s = (S, s, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow, \mathscr{U})$.*

We are now introducing window counting constraints as a mean to encode reoccurring behavior of the player *EGO* with limits on which action can be selected how often in each snippet (or: window) of a play of a given length.

Definition 3.5 (Window Counting Constraints)**.** *Let G be a game graph with the two players EGO and ALTER, denote with a an action and $k, l \in \mathbb{N}$ with $k \leq l$. Let $\pi = (\pi_i\sigma_i)_{i\in\mathbb{N}_0}$ be a play on G.*

1. ***CC_max(EGO, a, k, l)*** *is defined as the abbreviation for "The player EGO plays action a at most k times out of l of its own turns."*
   $CC_{max}(EGO, a, k, l)$ is satisfied on π, if for all $i \in \mathbb{N}_0$ holds $|\{\sigma_{2m} \,|\, \sigma_{2m} = a,\ i \leq m \leq i+l\}| \leq k$.
2. ***CC_min(EGO, a, k, l)*** *is defined as the abbreviation for "The player EGO plays action a at least k times out of l of its own turns."*
   $CC_{min}(EGO, a, k, l)$ is satisfied on π, if for all $i \in \mathbb{N}_0$ holds $|\{\sigma_{2m} \,|\, \sigma_{2m} = a,\ i \leq m \leq i+l\}| \geq k$.

A prefix of a play on G satisfies a counting constraint, if it can be complemented to an infinite play that satisfies the counting constraint in any way (in particular, the extended prefix does not need to be a play on G). The parameter l is called the length of a counting constraint.

The above definition might raise the question why we do not consider similar counting constraints for the player $ALTER$, representing the environment. Such constraints impose a set of challenges, which we will discuss in Section 5 and plan to tackle as future work.

We extend the definition of satisfying a counting constraint for a play canonically to satisfying a set of counting constraints and counting constraints being satisfied on a strategy.
In a (zero-sum) game with counting constraints, the EGO player needs to satisfy all of its counting constraints in order to win the game.

Definition 3.6 (Games with Counting Constraints)**.** *A two-player* ***game with counting constraints*** *is defined as $G = (G', CC_{EGO})$, where*

- $G' = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow)$ *is a two-player finite-state game graph.*
- $CC_{EGO} \subset \{CC_m(EGO, a, k, l) \,|\, m \in \{min, max\}, k, l \in \mathbb{N}, k \leq l, a \in \Sigma_{EGO}\}$ *is a finite sets of counting constraints of EGO*

Player EGO wins a play on G, if the play satisfies all counting constraints CC_{EGO}. Otherwise, ALTER wins. A strategy φ of EGO is winning for EGO (or a ***winning strategy*** *of EGO), if EGO wins all plays in $O(G, \varphi)$.*

4 Iterated Synthesis with Counting Constraints

The key advantage of counting constraints for synthesis is their monotony property: If EGO has a strategy, such that EGO plays an action a at most k times in l turns (i.e. the strategy satisfies $CC_{max}(EGO, a, k, l)$), then EGO also plays a at most k times in $l-1$ turns (i.e. the strategy satisfies $CC_{max}(EGO, a, k, l-1)$). In other words: The existence of a winning strategy for a game with counting constraint $CC_{max}(EGO, a, k, l-1)$ is a necessary condition for the winning strategy for a game with $CC_{max}(EGO, a, k, l)$. Moreover, only a strategy that fulfills $CC_{max}(EGO, a, k, l-1)$ can also fulfill $CC_{max}(EGO, a, k, l)$. From an algorithmic perspective, it is more favorable to search for strategies that satisfy $CC_{max}(EGO, a, k, l-1)$ then for strategies that satisfy $CC_{max}(EGO, a, k, l)$, since the graph that encodes the first (shorter) constraint is smaller than the one that encodes the latter (longer) constraint. Intuitively, this is caused by more memory that is needed for remembering the last l own turns instead of only $l-1$ turns. The synthesis idea is related to the incremental approach used by synthesis with antichains [6].
For a counting constraint of the form $CC_{min}(EGO, a, k, l-1)$ ("EGO plays action a at least k times out of $l-1$ of its turns"), we can conduct that if a strategy fulfills the constraint, it automatically also fulfills the larger constraint $CC_{min}(EGO, a, k, l)$. Hence, if we already have a strategy that fulfills $CC_{min}(EGO, a, k, l-1)$, it is needless to do the more challenging search for a strategy that fulfills $CC_{min}(EGO, a, k, l)$.

Theorem 4.1. *Let $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow, CC_{EGO})$ be a two-player game with counting constraints.*

1. *For $CC_{max}(EGO, a, k, l) \in CC_{EGO}$ holds: If φ is a winning strategy for EGO on G, then it is also a winning strategy for EGO on G', where G' equals G except that $CC_{max}(EGO, a, k, l)$ is exchanged by $CC_{max}(EGO, a, k, l-1)$.*
2. *For $CC_{min}(EGO, a, k, l) \in CC_{EGO}$ holds: If φ is a winning strategy for EGO on G, then it is also a winning strategy for EGO on G', where G' equals G except that $CC_{max}(EGO, a, k, l)$ is exchanged by $CC_{max}(EGO, a, k, l+1)$.*

Since the proof is straightforward, we omit it here. It is also possible to vary the k parameter in the constraints instead of l with similar conclusions. With each of those iterations, the number of previously made turns that need to be memorized is increasing. We introduce situation graphs as a mean to encode the relevant history of a play into game graphs. In a nutshell, a situation is a state of the game graph G combined with the counting constraint-relevant part of the history on how the state was reached. It allows for categorizing states of the game into "part of the winning region" and "not winnable", which reduces a game with counting constraints to a classical safety game with states in which EGO violates its constraints as unsafe states.

Definition 4.1 (Situation Graph)**.** *Let* $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow, CC_{EGO})$ *be a game with counting constraints. Fix some order* $CC_{EGO} = \{C_{EGO,1}, \ldots, C_{EGO,q}\}$ *of the counting constraints. For each counting constraint* $C = CC_m(EGO, a, k, l) \in CC_{EGO}$*,* $m \in \{min, max\}$ *define a transition*

$$h_C\colon \{0,1,none\}^l \times \Sigma_{EGO} \rightarrow \{0,1,none\}^l, \quad ((v_1,\ldots,v_l), act) \mapsto \begin{cases} (1, v_1, \ldots, v_{l-1}), \textit{ if } act = a \\ (0, v_1, \ldots, v_{l-1}), \textit{ else.} \end{cases}$$

A situation is a tuple (s, H_{EGO}) *with* $s \in S$ *being a state in* G*,* $H_{EGO} \in \bigtimes_{i=1}^{q} codom(h_{C_{EGO,i}})$ *and* $codom(f) = Y$ *denoting the codomain of a function* $f\colon X \rightarrow Y$*. Denote the set of all situations by* $\tilde{S}$*. Define a transition*

$$\hookrightarrow'\colon \tilde{S} \times \Sigma_{EGO} \rightarrow \tilde{S}$$

$$((s,(v_1,\ldots,v_q)), act) \mapsto \begin{cases} (s', (h_{C_{EGO,1}}(v_1, act), \ldots, h_{C_{EGO,q}}(v_q, act))), \textit{ if } s \in S_{EGO} \\ (s', (v_1, \ldots, v_q)), \textit{ if } s \in S_{ALTER} \end{cases}$$

such that $(s, act, s') \in \rightarrow$*. The transition* $\hookrightarrow'$ *defines how to get from one situation to another when using the transition* $\rightarrow$ *in* G*.*
A situation (s, H_{EGO}) *is satisfying a counting constraint* $CC_{min}(EGO, a, k, l) \in CC_{EGO}$*, if for the corresponding part* $(v_1, \ldots, v_l)$ *in* H_{EGO} *holds* $|\{v_i \,|\, v_i = 1, i = 1, \ldots, l\}| \leq k$*. Similarly, the situation satisfies* $CC_{max}(EGO, a, k, l) \in CC_{EGO}$*, if* $|\{v_i \,|\, v_i = 1, i = 1, \ldots, l\}| \geq k$*.*
The situation graph of G *is the two-player finite game graph* $Sit = (S', s_{init}, S'_{EGO}, S'_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \hookrightarrow)$ *with*

- *initial state being the situation* $s_{init} = (s_0, H_{init,EGO})$ *with all entries in* $H_{init,EGO}$ *being none,*
- *transition relation* $\hookrightarrow \subseteq \tilde{S} \times \Sigma_{EGO} \times \tilde{S}$ *with* $(s, act, s') \in \hookrightarrow$
- *set of states* S' *being all situations that are reachable from* s_{init} *via* $\hookrightarrow$*,*
- $S'_p \subseteq S'$ *the states* (s, H_{EGO}) *that are controlled by player* $p \in \{EGO, ALTER\}$*, that is* $s \in S_p$*.*

The winning region of EGO in the situation graph is the set of states $\tilde{S} \subseteq S'$ *from which EGO has a winning strategy, that is, from which EGO can guarantee to only visit states that satisfy all counting constraints in* CC_{EGO}*.*

The situation graph of a game is a deterministic Büchi automaton that represents the full specification of EGO, if the complete set of counting constraints is considered. Counting constraints are expressible as (long) LTL-formulae, hence, using the full situation graph for synthesis is generally only doable in time double-exponential in the size of the LTL-specification [17]. The iterated synthesis approach avoids to construct the full situation graph. The general idea is to start with a rather small game by using counting constraints of small lengths and iterate over the length. In each iteration, a part of the corresponding

situation graph is constructed and analyzed and knowledge that can be reused in following iterations is identified. This knowledge is determining which parts of the situation graph for the next iteration needs to be constructed and which parts can be omitted, relying on Theorem 4.1.

For iteration over one counting constraint $CC_{min}(EGO,a,k,l)$, the synthesis procedure is sketched in Algorithm 1 and algorithms called therein. For better readability, the algorithms only handle one other counting constraint $CC_{max}(EGO,b,m,n)$ besides the one that is iterated over. However, since the other counting constraint remains fixed during the iteration approach, it is possible to add additional (fixed) counting constraints with only minor adaptions. Algorithm 1 basically alternates between calling two other algorithms: Starting with the smallest possible counting constraint $CC_{min}(EGO,a,k,k)$, the situation graph for the respective game is generated (Algorithm 2). After that, the resulting graph is analyzed in order to find the winning region for EGO (Algorithm 3). If the initial state of the situation graph belongs to the winning region, a set of winning strategies for EGO is found and the algorithm terminates. If the initial state is not winnable, the next iteration starts with the next longer counting constraint. If even the winning region of the situation graph for $CC_{min}(EGO,a,k,l)$ does not contain the initial state of the situation graph, no winning strategy for EGO exists.

Algorithm 2 generates (parts of) the situation graphs in each iteration. States of the situation graph are called "situations" in the algorithm in order to avoid confusion with the states of the underlying game graph. Note that the algorithm omits successors of states that violate counting constraints of EGO, since those states do not belong to the winning region (line 13). In the first iteration, there is no additional information on winnable states available, hence the full situation graph (minus successors of states violating constraints of EGO) needs to be constructed. Due to the small counting constraint length, this graph is significantly smaller than it would be for the full constraint length. As soon as Algorithm 3 identifies any winnable states, this knowledge can than be used in the construction of the situation graph in the next iteration: The construction begins with adding the initial state to an empty (directed) graph. Successors of already added states are added successively. For each added state, it is checked if there is a "related" state in the winning region of the previous iteration. If this is the case, the state can also be marked as being in the winning region and successors do not need to be considered. As a consequence, the situation graph is only partly constructed, saving computational time and memory. A state s of a situation graph in one iteration for a counting constraint with action a is related to a state s' of the situation graph of the previous iteration, if s can be transformed into s' by only deleting the last entry of the history of a. The identification of such states is the key factor for more efficient synthesis via the presented approach, since it allows to perform synthesis on incomplete graphs, allowing for a pruning step in each iteration.

Algorithm 3 calculates the winning region for a given (incomplete) situation graph. The reduction of the size of the graph by incomplete construction is again speeding up the algorithm. States of the situation graph without successor are considered first. Such states are either already identified as being winnable since they are related to winnable states of the previous iteration (line 1) or can be marked as non-winnable (aka $losing$, line 2), since the counting constraint of EGO is violated. The rest of the algorithm is rather generic and uses a version of fixed point computation for a finite-state two-player safety game with the already identified states in $losing$ as unsafe states.

In iterations over counting constraints of the form $CC_{max}(EGO,a,k,l)$, it is searched for states of the situation graph that are not in the winning region of EGO. Such states will also not be visited by winning strategies in the following iterations. Except for searching for non-winnable states instead of winnable states, the synthesis procedure is similar to the one for $CC_{min}(EGO,a,k,l)$ constraints. If the initial state of a situation graph in any iteration is marked as non-winnable, there exists no winning strategy for EGO.

Algorithm 1: Iterated Synthesis over one $CC_{min}(EGO,a,k,l)$ counting constraint

Input: $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow, \{CC_{min}(EGO,a,k,l), CC_{max}(EGO,b,m,n)\})$ - two-player game with counting constraints.

Output: If winning strategy for *EGO* exists:
situation_graph - part of the smallest situation graph in which a winning strategy exists
winning_situations - set of states of the situation graph, forming a subset of the winning region for the graph
Else: LOSING - no winning strategy for *EGO* exists

```
1  winning_region ⟵ empty directed graph
2  for c ∈ {k,...,l} do                // increase EGO counting constraint length from k to l
3      situation_graph, ⟵ generate_situation_graph(G,k,c,m,n,winning_region.states)
4      winning_region ⟵ find_winning_region(situation_graph, {CC_min(EGO,a,k,c), CC_max(EGO,b,m,n)}, S_EGO, S_ALTER)
5      if situation_graph.initial_situation ∈ winning_region.states then
6          return situation_graph, winning_region
7  return LOSING
```

Algorithm 2: generate_situation_graph: Construction of the situation graph without unfolding regions already won in previous iterations

Input: $G = (S, s_0, S_{EGO}, S_{ALTER}, \Sigma_{EGO}, \Sigma_{ALTER}, \rightarrow, \{CC_{min}(EGO,a,k,c), CC_{max}(EGO,b,m,n)\})$ two-player safety game with counting constraints; *previous_winning_situations* set of winnable states of the situation graph for G with $CC_{min}(EGO,a,k,c-1)$

Output: *situation_graph* - situation graph of G without unfolding areas that are already winnable in the previous iteration (considering $CC_{min}(EGO,a,k,c-1)$)

```
1  initial_situation ⟵ (s_0,[none for i in range(c)], [none for i in range(n)]); unfinished_situations ⟵ {initial_situation}
2  finished_situations ⟵ ∅; winning_situations ⟵ ∅
3  situation_graph ⟵ empty directed graph; situation_graph.situations ⟵ {initial_situation}
4  A ⟵ CC_min(EGO,a,k,c); B ⟵ CC_max(EGO,b,m,n)
5  while unfinished_situations do          // while not all successors of states in the graph are considered
       /* take a situation from unfinished_situations and add all needed successors to the graph     */
6      choose any current_situation ∈ unfinished_situations; unfinished_situations.remove(current_situation)
7      all_next_moves ⟵ {(current_situation.state, act, s') ∈
       (S_EGO ∪ S_ALTER) × (Σ_EGO ∪ Σ_ALTER) × (S_EGO ∪ S_ALTER) | (current_situation.state, act, s') ∈→}
8      for next_move ∈ all_next_moves do
9          if current_situation.state ∈ S_EGO then           // case: EGO controls the current situation
               /* construct one successor of current_situation in the situation graph                */
10             next_situation ⟵ (next_move.tail, [next_move.action == a, current_situation.history_EGO.A[:
                 −1]], [next_move.action == a, current_situation.history_EGO.B[: −1]])
11             if next_situation ∉ situations then        // case: situation not yet in the situation graph
12                 situation_graph.situations.add(next_situation)
13                 if next_situation does not satisfy A or B then finished_situations.add(next_situation)
14                 else  // check if next_situation is related to a winnable situation of prev. iteration
15                     related_next_situation ⟵ (next_situation.state, next_situation.history_EGO.A[:
                         −1], next_situation.history_EGO.B)
16                     if related_next_situation ∈ previous_safe_situations then
17                         winning_situations.add(next_situation); finished_situations.add(next_situation)
18                     else
19                         if next_situation ∉ finished_situations then unfinished_situations.add(next_situation)
20             situation_graph.transitions.add((current_situation, next_move.action, next_situation))
21         else                                             // case: ALTER controls the current situation
22             next_situation ⟵ (next_move.tail, current_situation.history_EGO)
23             if next_situation ∉ situations then
24                 situation_graph.situations.add(next_situation)
25                 related_next_situation ⟵ (next_situation.state, next_situation.history_EGO.A[:
                     −1], next_situation.history_EGO.B)
26                 if related_next_situation ∈ previous_safe_situations then
27                     winning_situations.add(next_situation); finished_situations.add(next_situation)
28             situation_graph.transitions.add((current_situation, next_move.action, next_situation))
29             if next_situation ∉ finished_situations then unfinsihed_situations.add(next_situation)
30     finished_situations.add(current_situation)
31 return situation_graph
```

Algorithm 3: find_winning_region

Input: *situation_graph* as constructed in Algorithm 2; $CC_{min}(EGO,a,k,c)$, $CC_{max}(EGO,b,m,n)$ counting constraints belonging to *situation_graph*; S_{EGO}, S_{ALTER} states of the underlaying game

Output: part of the winning region for EGO in *situation_graph*

`/* Divide states of` *situation_graph* `without successor into winnable and losing states */`

1 *winning* $\longleftarrow \{sit \mid$ state *sit* has no successor and satisfies $CC_{min}(EGO,a,k,c)$ and $CC_{max}(EGO,b,m,n)\}$

2 *losing* $\longleftarrow \{sit \mid$ state *sit* has no successor and does not satisfy $CC_{min}(EGO,a,k,c)$ or $CC_{max}(EGO,b,m,n)\}$

`/* mark predecessors of winning` $ALTER$`-situations as winning */`

3 *winning*.add($\{pred \mid pred$ is predecessor of some $sit \in winning$ with $sit.state \in S_{EGO}\}$)

`/* mark` $ALTER$`-situations as winning, if all successors are winning */`

4 *winning*.add($\{sit \mid sit.state \in S_{ALTER},$ for all successors *suc* of *sit* holds: $suc \in winning\}$)

`/* identify losing states */`

5 *progress* $\leftarrow$ TRUE

6 **while** *progress* **do**

7   *progress* $\leftarrow$ FALSE

    `/* handle all situations controlled by` EGO `and marked as losing */`

8     $losing_EGO_sit = \{situation \mid situation.state \in S_{EGO}\} \cap losing$

9   **if** *losing_EGO_sit* **then**

10   *losing*.add($\{predecessor \mid \exists sit \in losing_EGO_sit : sit$ is a successor of $predecessor\}$)

        `/* delete all ingoing and outgoing transitions from states in` *losing_EGO_sit* `and those states itself from` *situation_graph* `*/`

11   *situation_graph*.remove_nodes_from(*losing_EGO_sit*); *progress* $\longleftarrow$ TRUE

    `/* handle situations controlled by` $ALTER$ `& already marked as losing */`

12     $losing_ALTER_sit \leftarrow \{situation \mid situation.state \in S_{ALTER}\} \cap losing$

13   **if** *losing_ALTER_sit* **then**

        `/* delete all ingoing and outgoing transitions from states in` *losing_ALTER_sit* `and those states itself from` *situation_graph* `*/`

14   *situation_graph*.remove_nodes_from(*losing_ALTER_sit*); *progress* $\longleftarrow$ TRUE

    `/* handle situations not marked as winning and without successor */`

15     $no_win \leftarrow \{situation \mid situation \notin winning, situation$ has no successor in *situation_graph*$\}$

16   **if** *no_win* **then** *losing*.add(*no_win*); *progress* $\longleftarrow$ TRUE

17 **return** *situation_graph*

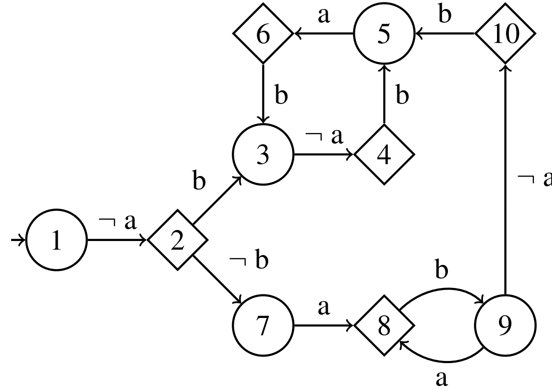
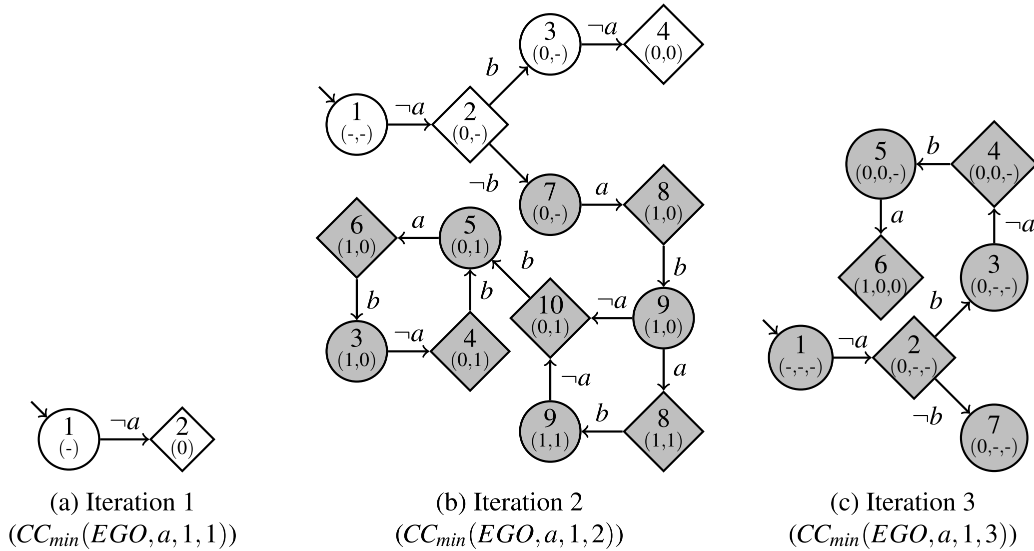


Figure 1: Two-player game graph. States represented as circles are controlled by EGO, diamond-shaped states are controlled by $ALTER$. EGO shall fulfill the counting constraint $CC_{min}(EGO,a,1,7)$ (EGO plays a at least one time in 7 turns).

For illustrating the synthesis algorithm, we consider the game in Figure 1 as small example. Circles represent locations controlled by EGO. Diamond-shaped locations are controlled by $ALTER$. Let $CC_{min}(EGO,a,1,7)$ be a counting constraint that EGO needs to satisfy. For the sake of keeping the example small, we pass on more counting constraints and only distinguish between the actions "a" and "$\neg a$" of EGO. The constructed parts of the situation graphs for three iterations are shown in Figure 2. Each state of the situation graph is marked with the respective state number of the game graph and with the history of last counting constraint-relevant turns of EGO. The history length depends on the size of the counting constraint in the considered iteration. For example, the state marked with state 9 and history $(1,0)$ in Figure 2b encodes that EGO played a in its last turn and played something else ($\neg a$) in its second to last turn. States highlighted with gray background are identified as being winnable. The first iteration reduces the counting constraint to $CC_{min}(EGO,a,1,1)$ ("EGO plays a at least in one of 1 turns"), fully specifying how EGO is allowed to behave. The corresponding situation graph is shown in Figure 2a. State $2,(0)$ has no successor, since the counting constraint is already violated in this state. None of the states of the situation graph are in the winning region of the game. In the second iteration, the counting constraint for EGO is more relaxed, consequently the situation graph (Figure 2b) has more states. 10 of the states belong to the winning region of EGO, since EGO can guarantee to avoid states with counting constraint violations (state $4,(0,0)$) from those states. Since the initial state is not marked as winnable, there exists no winning strategy for EGO and the third iteration is entered. In the situation graph for the third iteration (Figure 2c) the benefit of the iterated approach becomes visible: State $7,(0,-,-)$ is related to state $7,(0,-)$ of the previous iteration and since the latter one is already marked as winnable, so can state $7,(0,-,-)$. Hence, successors of $7,(0,-,-)$ do not need to be further considered. The same holds for state $6,(1,0,0)$, which is related to the winnable state $6,(1,0)$ of the second iteration. As a consequence, the situation graph of the third iteration is even smaller than the one of the second iteration. The initial state $1,(-,-,-)$ can now be marked as winnable, hence there already exists a winning strategy for EGO in the third iteration and no further iteration is required. Please note that the focus on the example is to show how the situation graph evolves over multiple iterations, illustrating the benefit of the iterated approach. However, the example is too small to actually be significantly more efficient than synthesizing a winning strategy without iterations.

A non-optimized explicit state implementation of Algorithm 1 in Python was used to give an idea for



(a) Iteration 1 ($CC_{min}(EGO, a, 1, 1)$)

(b) Iteration 2 ($CC_{min}(EGO, a, 1, 2)$)

(c) Iteration 3 ($CC_{min}(EGO, a, 1, 3)$)

Figure 2: Situation graphs for the game in Figure 1 with iteration over $CC_{min}(EGO, a, 1, 7)$. More than three iterations are not necessary, since there already is a winning strategy for EGO in the third iteration.

the performance of iterated synthesis with counting constraints in larger examples. We will now sketch the insights retrieved from an exemplary game, solved with this implementation. The game graph had around 1.8mio states, 2.7mio transitions and a counting constraint for EGO of length 10. The algorithm took around 28 minutes to find a winning strategy for EGO in the 8th iteration. Hence, iteration 9 and 10 were neither needed nor performed. The situation graph in the last required iteration had around 2.8mio states. For comparing those numbers with a non-iterated synthesis approach, Algorithm 2 and Algorithm 3 were used to directly calculate a winning strategy for constraint length 8 for the same game. The calculation required around 4 times longer and used around 2.5 times more states. Note that limiting the constraint length directly to 8 was only possible because of the retrieved knowledge on strategy existence for this constraint length of the iterated synthesis calculations before. The comparison would even be more in favor of the iteration approach if the minimal counting constraint for which a winning strategy exists were not given for the non-iterative computation. We did not let the non-iterative algorithm run for the full counting constraint length of 10, since the expected amount of required states would have reached hardware limitations. The realized comparison shows the great potential of successively enlarging counting constraints, allowing for incomplete graph constructions due to retrieved information on already winnable states of prior graphs instead of encoding the full constraints directly in a graph for strategy synthesis.

5 Discussion and Future Work

The exploitation of the monotony property inherent in counting constraints for iterative synthesis has demonstrated promising outcomes, indicating the potential for time- and memory-efficient computation of controllers for reactive systems. The current investigation aimed to explore the broader applicability of iterated synthesis utilizing counting constraints, an objective that has been achieved. However, certain challenges and considerations in the chosen game setting should be discussed in the following, paving the way for future research directions.

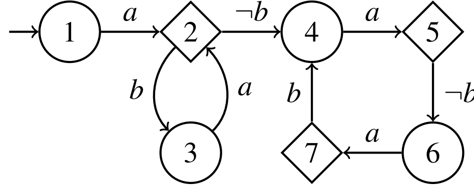


Figure 3: $ALTER$ can always fulfill the counting constraint $CC_{min}(ALTER, b, 1, 3)$, but can run into a violation for $CC_{min}(ALTER, b, 1, 2)$.

Towards cooperative games: As already mentioned above, the idea of adding window counting constraints like "The player $ALTER$ plays a at least (or: at most) k times out of l of its own turns." for the other player $ALTER$ seems obvious. In the current setting, we apply the synthesis algorithm on the winning region of the underlying safety game. If $ALTER$-constraints are added, the previous winning region (without counting constraints) would only be an under-approximation of the winning region for the safety game with counting constraints. Hence, the synthesis algorithm may fail to find an existing winning strategy for EGO. The problem can be solved by omitting the calculation of the winning region beforehand and integrate the safety condition in the iterated synthesis approach. This can be done by handling unsafe states the same way as states in which EGO violates its constraints. If we want to stay in a zero-sum game setting, we could restrict the games of interest to those in which $ALTER$ can actually fulfill its constraints. The following property could be added to the definition of a game with counting constraints (Definition 3.6). $ALTER$ cannot be forced into constraint violations: For each prefix $\pi(n) = \pi_0\sigma_0\pi_1 \dots \pi n - 1\sigma_{n-1}\pi_n$, $n \in 2\mathbb{N}+1$, of a play on G that satisfies all counting constraints of $ALTER$, there exists $(\pi_n, a, \pi_{n+1}) \in \rightarrow$, such that $\pi_0\sigma_0\pi_1 \dots \pi_{n-1}\sigma_{n-1}\pi_n a \pi_{n+1}$ is also a prefix of a play on G that satisfies the counting constraints of $ALTER$. This property simplifies the formulation of winning conditions for EGO, circumventing complex scenarios arising from ambiguous outcomes wherein one player forces the other into constraint violations at the expense of own future constraint violations. However, this restriction is limiting the possibility to iterate over counting constraints to constraints of EGO. In general, a game with counting constraints may satisfy the requirement of $ALTER$ always being able to adhere to its counting constraints, only to find the requirement violated for the game with a modified counting constraint as used in the iterations. An illustrative example is provided in Figure 3. $ALTER$ has the counting constraint $CC_{min}(ALTER, b, 1, 3)$, i.e. $ALTER$ plays b at least once in three of its turns. Recall that "$ALTER$ cannot be forced into constraint violations" is defined in Definition 3.6 as $ALTER$ is able to enlarge each prefix that satisfies the constraint such that the resulting prefix is also satisfying the constraint. This property is fulfilled when considering the game graph and the constraints $CC_{min}(ALTER, b, 1, 1)$ or $CC_{min}(ALTER, b, 1, 3)$. However, it is violated for $CC_{min}(ALTER, b, 2, 3)$, since the prefix $(1, a, 2, \neg b, 4, a, 5)$ satisfies the constraint[1], but there is no possibility for $ALTER$ to still satisfy the constraint with the next turn. Since the definition of a winning strategy relies on the game property of $ALTER$ not be forceable into counting constraint violations, Theorem 4.1 cannot be extended to iterations over $ALTER$-constraints. However, such an extension would offer additional potential for more efficient synthesis algorithms.

We plan to approach this problem by leaving the zero-sum setting. The environment wins if it has a strategy that guarantees to satisfy all of its counting constraints. In particular, it is possible that the environment violates a constraint and loses. The envisioned game setting shall avoid the well-known

[1] $ALTER$ could play b forever to complete the prefix to an infinite play that satisfies the constraint. This play is not in G, but nonetheless is sufficient according to the definition of a prefix satisfying a constraint in Definition 3.5.

problem of EGO winning only by falsifying the assumptions in form of counting constraints on $ALTER$. Instead, EGO shall support $ALTER$ in satisfying all constraints as long as this does not compromise the adherence of own constraints. This leads us in the direction of searching for strategy profiles with certain properties as synthesis results instead of winning strategies only for EGO with the exact profile properties yet to be determined. It can be foreseen that this setting requires more synchronization between the players than that presented by a zero-sum setting, in which $ALTER$ did not even need knowledge on counting constraints of EGO.

Extension of counting constraint types: It is worth to consider additional specification patterns with similar monotony properties as the presented counting constraints. For instance, a pattern like "if x is played, EGO plays y after at most k turns" is frequently used as specification. Satisfying such a specification becomes easier for larger k. In terms of an iterative algorithm: states of the situation graph for some iteration are winnable, if the related state is winnable in an earlier iteration. The identification of additional counting constraints and the adaption of the iterative strategy synthesis algorithm to such constraints increases the applicability of the approach to more systems.

Combination of various counting constraints: In the presented synthesis algorithm, iteration is only done over one counting constraint. All other constraints remain fixed. We anticipate greater savings in memory and computational time than already provided by the presented algorithm by iterating over several constraints (successively or alternating). Such an extension is expected to require only manageable modifications of the existing algorithm for sets of counting constraints that use the same type of information from one iteration to the other (e.g. exclusively on winnable states of the various situation graphs). The iteration over a set of constraints that use different types of information during iteration (e.g. on winnable states for some of the constraints and on non-winnable states for other constraints) is expected to require a more thorough adaption of the algorithm.

Symbolic representation: The presented synthesis approach uses an explicit representation of states in the situation graph as arena. However, symbolic synthesis showed to be significantly more efficient than explicit synthesis algorithms for many (but not all) applications [8]. Since the presented approach already has similarities to antichains and the states of the arena have a special structure (representing a snippet of the history of a play), we expect that the approach can be transformed in a symbolic algorithm. We plan to investigate a symbolic version of the algorithm and to compare its performance with its explicit version.

6 Conclusion

Synthesis algorithms for reactive systems are promising tools for various engineering tasks, most prominently for the creation of correct-by-construction controllers and for checking the feasibility of specifications. The efficiency of such algorithms is a challenge for getting synthesis into application, since the translation of the system specification into an automaton that is suitable for synthesis is costly in terms of memory and computational time. The exploitation of specific properties in the specification can help to overcome this challenge. In this paper, we have shown the potential of iterative synthesis algorithms for specifications with monotony properties as for the presented counting constraints. With each iteration, the automaton encoding the specification is becoming larger. The key idea is to gather information in each iteration that can be used in the next iteration to reduce the size of the automaton. The precise nature of this information depends on the considered specification. We have shown an iterated algorithm for a counting constraint of the form "the system does a specific move m at least in k turns out of l", in which information on winnable states of the automaton in one iteration can be used to deter-

mine which parts of the automaton for the next iteration do not need to be constructed. In the presented example, the iterative approach requires significantly less memory and computational time than direct synthesis with full specification translation into one automaton. As future work, we plan to extend the iterative synthesis in four dimensions: (1) Consideration of more cooperative behavior between system and its environment instead of a purely adversarial setting, (2) identification of new specification types with monotony properties that can be exploited via iterated synthesis, (3) development of algorithms that use advantages of different specification types simultaneously and (4) transformation of the synthesis approach to a symbolic synthesis algorithm.

References

[1] J. Richard Buchi & Lawrence H. Landweber (1969): *Solving sequential conditions by finite-state strategies*. In Ernst W. Mayr & Claude Puech, editors: *Transactions of the American Mathematical Society*, *Transactions of the American Mathematical Society* 138, American Mathematical Society, pp. 295–311, doi:10.1090/S0002-9947-1969-0280205-0. Available at `https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1087&context=cstech`.

[2] Mingshuai Chen, Martin Fränzle, Yangjia Li, Peter Nazier Mosaad & Naijun Zhan (2018): *What's to Come is Still Unsure - Synthesizing Controllers Resilient to Delayed Interaction*. In Shuvendu K. Lahiri & Chao Wang, editors: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, *Lecture Notes in Computer Science* 11138, Springer, pp. 56–74, doi:10.1007/978-3-030-01090-4_4. Available at `https://moves.rwth-aachen.de/wp-content/uploads/ATVA2018_FULL.pdf`.

[3] Alonso Church (1957): *Applications of recursive arithmetic to the problem of circuit synthesis*. In: *Summaries of the Summer Institute of Symbolic Logic*, Cornell Univ., Ithaca, NY, pp. 3–50, doi:10.2307/2271310.

[4] Alessandro Cimatti, Luca Geatti, Nicola Gigante, Angelo Montanari & Stefano Tonetta (2020): *Reactive Synthesis from Extended Bounded Response LTL Specifications*. In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, IEEE, pp. 83–92, doi:10.34727/2020/ISBN.978-3-85448-042-6_15.

[5] Rüdiger Ehlers (2010): *Symbolic Bounded Synthesis*. In Tayssir Touili, Byron Cook & Paul B. Jackson, editors: *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, *Lecture Notes in Computer Science* 6174, Springer, pp. 365–379, doi:10.1007/978-3-642-14295-6_33.

[6] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2009): *An Antichain Algorithm for LTL Realizability*. In Ahmed Bouajjani & Oded Maler, editors: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, *Lecture Notes in Computer Science* 5643, Springer, pp. 263–277, doi:10.1007/978-3-642-02658-4_22.

[7] Emmanuel Filiot, Naiyong Jin & Jean-François Raskin (2011): *Antichains and compositional algorithms for LTL synthesis*. *Formal Methods Syst. Des.* 39(3), pp. 261–296, doi:10.1007/S10703-011-0115-3.

[8] Bernd Finkbeiner (2016): *Synthesis of Reactive Systems*. In Javier Esparza, Orna Grumberg & Salomon Sickert, editors: *Dependable Software Systems Engineering*, *NATO Science for Peace and Security Series - D: Information and Communication Security* 45, IOS Press, pp. 72–98, doi:10.3233/978-1-61499-627-9-72. Available at `https://finkbeiner.groups.cispa.de/publications/F16.pdf`.

[9] Bernd Finkbeiner & Swen Jacobs (2012): *Lazy Synthesis*. In Viktor Kuncak & Andrey Rybalchenko, editors: *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, *Lecture Notes in Computer Science* 7148, Springer, pp. 219–234, doi:10.1007/978-3-642-27940-9_15. Available at `https://finkbeiner.groups.cispa.de/publications/lazySynthesis.pdf`.

[10] Marcin Jurdzinski (2000): *Small Progress Measures for Solving Parity Games*. In Horst Reichel & Sophie Tison, editors: *STACS 2000, 17th Annual Symposium on Theoretical Aspects of Computer Science, Lille, France, February 2000, Proceedings*, *Lecture Notes in Computer Science* 1770, Springer, pp. 290–301, doi:10.1007/3-540-46541-3_24. Available at `https://www.dcs.warwick.ac.uk/~mju/Papers/Jur00-STACS.pdf`.

[11] Orna Kupferman & Moshe Y. Vardi (2005): *Safraless Decision Procedures*. In: *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, IEEE Computer Society, pp. 531–542, doi:10.1109/SFCS.2005.66.

[12] Oded Maler, Dejan Nickovic & Amir Pnueli (2007): *On Synthesizing Controllers from Bounded-Response Properties*. In Werner Damm & Holger Hermanns, editors: *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, *Lecture Notes in Computer Science* 4590, Springer, pp. 95–107, doi:10.1007/978-3-540-73368-3_12.

[13] Shahar Maoz & Jan Oliver Ringert (2015): *GR(1) synthesis for LTL specification patterns*. In Elisabetta Di Nitto, Mark Harman & Patrick Heymans, editors: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, ACM, pp. 96–106, doi:10.1145/2786805.2786824. Available at `https://www.researchgate.net/publication/299909728_GR1_synthesis_for_LTL_specification_patterns`.

[14] Robert McNaughton (1993): *Infinite Games Played on Finite Graphs*. *Ann. Pure Appl. Logic* 65(2), pp. 149–184, doi:10.1016/0168-0072(93)90036-D.

[15] Kostas Patroumpas & Timos K. Sellis (2006): *Window Specification over Data Streams*. In Torsten Grust, Hagen Höpfner, Arantza Illarramendi, Stefan Jablonski, Marco Mesiti, Sascha Müller, Paula-Lavinia Patranjan, Kai-Uwe Sattler, Myra Spiliopoulou & Jef Wijsen, editors: *Current Trends in Database Technology - EDBT 2006, EDBT 2006 Workshops PhD, DataX, IIDB, IIHA, ICSNW, QLQP, PIM, PaRMA, and Reactivity on the Web, Munich, Germany, March 26-31, 2006, Revised Selected Papers*, *Lecture Notes in Computer Science* 4254, Springer, pp. 445–464, doi:10.1007/11896548_35. Available at `https://dl.ifip.org/db/conf/edbtw/edbtw2006/PatroumpasS06.pdf`.

[16] Nir Piterman, Amir Pnueli & Yaniv Sa'ar (2006): *Synthesis of Reactive(1) Designs*. In E. Allen Emerson & Kedar S. Namjoshi, editors: *Verification, Model Checking, and Abstract Interpretation, 7th International Conference, VMCAI 2006, Charleston, SC, USA, January 8-10, 2006, Proceedings*, *Lecture Notes in Computer Science* 3855, Springer, pp. 364–380, doi:10.1007/11609773_24. Available at `https://www.wisdom.weizmann.ac.il/~saar/data/synth.pdf`.

[17] Amir Pnueli & Roni Rosner (1989): *On the Synthesis of an Asynchronous Reactive Module*. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini & Simona Ronchi Della Rocca, editors: *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, *Lecture Notes in Computer Science* 372, Springer, pp. 652–671, doi:10.1007/BFB0035790.

[18] Sven Schewe & Bernd Finkbeiner (2007): *Bounded Synthesis*. In Kedar S. Namjoshi, Tomohiro Yoneda, Teruo Higashino & Yoshio Okamura, editors: *Automated Technology for Verification and Analysis, 5th International Symposium, ATVA 2007, Tokyo, Japan, October 22-25, 2007, Proceedings*, *Lecture Notes in Computer Science* 4762, Springer, pp. 474–488, doi:10.1007/978-3-540-75596-8_33. Available at `https://link.springer.com/content/pdf/10.1007/s10009-012-0228-z.pdf`.

[19] Larry J. Stockmeyer (1974): *The complexity of decision problems in automata theory and logic*. Ph.D. thesis, Massachusetts Institute of Technology, USA. Available at `http://hdl.handle.net/1721.1/15540`.

[20] Wolfgang Thomas (1995): *On the Synthesis of Strategies in Infinite Games*. In Ernst W. Mayr & Claude Puech, editors: *STACS 95, 12th Annual Symposium on Theoretical Aspects of Computer Science, Munich, Germany, March 2-4, 1995, Proceedings*, *Lecture Notes in Computer Science* 900, Springer, pp. 1–13, doi:10.1007/3-540-59042-0_57.

[21] Wolfgang Thomas (2009): *Facets of Synthesis: Revisiting Church's Problem*. In Luca de Alfaro, editor: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*

York, UK, March 22-29, 2009. Proceedings, *Lecture Notes in Computer Science* 5504, Springer, pp. 1–14, doi:10.1007/978-3-642-00596-1_1.

[22] Shufang Zhu, Lucas M. Tabajara, Jianwen Li, Geguang Pu & Moshe Y. Vardi (2017): *A Symbolic Approach to Safety LTL Synthesis*. In Ofer Strichman & Rachel Tzoref-Brill, editors: *Hardware and Software: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, *Lecture Notes in Computer Science* 10629, Springer, pp. 147–162, doi:10.1007/978-3-319-70389-3_10. Available at `https://arxiv.org/abs/1709.07495`.

[23] Wieslaw Zielonka (1998): *Infinite Games on Finitely Coloured Graphs with Applications to Automata on Infinite Trees*. *Theor. Comput. Sci.* 200(1-2), pp. 135–183, doi:10.1016/S0304-3975(98)00009-7.

Deterministic Suffix-reading Automata

R Keerthan
Tata Consultancy Services Innovation Labs
Pune, India
Chennai Mathematical Institute, India
keerthan.r@tcs.com

B Srivathsan
Chennai Mathematical Institute, India
CNRS IRL 2000, ReLaX, Chennai, India
sri@cmi.ac.in

R Venkatesh   Sagar Verma
Tata Consultancy Services Innovation Labs
Pune, India
r.venky@tcs.com   verma.sagar2@tcs.com *

We introduce deterministic suffix-reading automata (DSA), a new automaton model over finite words. Transitions in a DSA are labeled with words. From a state, a DSA triggers an outgoing transition on seeing a word *ending* with the transition's label. Therefore, rather than moving along an input word letter by letter, a DSA can jump along blocks of letters, with each block ending in a suitable suffix. This feature allows DSAs to recognize regular languages more concisely, compared to DFAs. In this work, we focus on questions around finding a "minimal" DSA for a regular language. The number of states is not a faithful measure of the size of a DSA, since the transition-labels contain strings of arbitrary length. Hence, we consider total-size (number of states + number of edges + total length of transition-labels) as the size measure of DSAs.

We start by formally defining the model and providing a DSA-to-DFA conversion that allows to compare the expressiveness and succinctness of DSA with related automata models. Our main technical contribution is a method to *derive* DSAs from a given DFA: a DFA-to-DSA conversion. We make a surprising observation that the smallest DSA derived from the canonical DFA of a regular language L need not be a minimal DSA for L. This observation leads to a fundamental bottleneck in deriving a minimal DSA for a regular language. In fact, we prove that given a DFA and a number $k \geq 0$, the problem of deciding if there exists an equivalent DSA of total-size $\leq k$ is NP-complete.

1 Introduction

Deterministic Finite Automata (DFA) are fundamental to many areas in Computer Science. Apart from being the cornerstone in the study of regular languages, automata have been applied in several contexts: such as text processing [17], model-checking [4], software verification [2, 1, 9], and formal specification languages [12]. A central challenge in the application of automata is the size of the automaton involved. Non-determinism gives exponential succinctness, however, a deterministic model is useful in formal specifications and automata implementations. The literature offers different ways to get succinct representations of DFAs. We recall a few of them below and propose a new solution to this problem.

One of the reasons for large DFAs is the size of the alphabet, for instance, consider the alphabet of all ASCII characters. Having a transition for each letter from each state blows up the size of the automata. Symbolic automata [18, 6] have been proposed to handle large alphabets. Letters on the edges are replaced by formulas, which club together several transitions between a pair of states into one symbolic transition. Symbolic automata have been implemented in many tools and have been widely applied (see [5] for a list of tools and applications).

*All authors have contributed equally and are listed in the alphabetical order of last names.

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 70–87, doi:10.4204/EPTCS.409.9

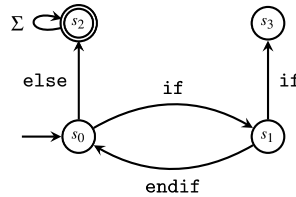


Figure 1: DSA for out-of-context `else`

Another dimension in reducing the DFA representation is to consider transitions on a block of letters. Generalized automata (GA) are extensions of non-deterministic finite automata (NFAs) that can contain strings instead of letters on transitions. A word w is accepted if it can be broken down as $w_1 w_2 \dots w_k$ such that each segment is read by a transition. This model was defined by Eilenberg [7], and later Hashiguchi [13] proved that for every regular language L there is a minimal GA in which the edge labels are at most a polynomial function in m, where m is size of the syntactic monoid of L. Giammarresi *et al.* [8] considers deterministic generalized automata (DGA) and proposes an algorithm to generate a minimal DGA (in terms of the number of states) in which the edges have length at most the size of the minimal DFA. The algorithm uses a method to suppress states and create longer labels. The key observation is that minimal DGAs can be derived from the canonical DFA by suppressing states.

Our model. In this work, we introduce *Deterministic Suffix-reading Automata (DSAs)*. We continue to work with strings on transitions, as in DGA. However, the meaning of transitions is different. A transition $q \xrightarrow{abba} q'$ is enabled if at q, a word w *ending* with *abba* is seen, and moreover no other transition out of q is enabled at a prefix of w. Intuitively, the automaton tracks a finite set of pattern strings at each state. It stays in a state until one of them appears as the *suffix* of the word read so far, and then makes the appropriate transition. We start with a motivating example. Consider a model for out-of-context `else` statements, in relation to `if` and `endif` statements in a programming language. Assume a suitable alphabet Σ of characters. Let $L_{\texttt{else}}$ be the set of all strings over the alphabet where (1) there are no nested `if` statements, and (2) there is an `else` which is not between an `if` and an `endif`. A DFA for this language performs string matching to detect the `if`, `else` and `endif`. The DSA is shown in Figure 1: at s_0, it passively reads letters until it first sees an `if` or an `else`. If it is an `if`, the automaton transitions to s_1. For instance, on a word `abf4fgif` the automaton goes to s_1, since it ends with `if` and there is no `else` seen so far. Similarly, at s_1 it waits for one of the patterns `if` or an `endif`. If it is the former, it goes to s_3 and rejects, otherwise it moves to s_0, and so on.

Suffix-reading automata have the ability to wait at a state, reading long words until a matching pattern is seen. This results in an arguably more readable specification for languages which are "pattern-intensive". This representation is orthogonal to the approaches considered so far. Symbolic automata club together transitions between a pair of states, whereas DSA can do this clubbing across several states and transitions. DGA have this facility of clubbing across states, but they cannot ignore intermediate letters, which results in extra states and transitions.

Overview of results. We formally present deterministic suffix-reading automata and its semantics, quantify its size in comparison to an equivalent DFA, and study an algorithm to construct DSAs starting from a DFA. This is in the same spirit as in DGAs, where smaller DGAs are obtained by suppressing states. For automata models with strings on transitions, number of states is not a faithful measure of the size of a DSA. As described in [8], we consider the total size of a DSA which includes the number of states, edges, and the sum of label lengths. The key contributions of this paper are:

1. Presentation of a definition of a new kind of automaton - DSA (Section 3).

2. Proof that DSAs accept regular languages, and nothing more. Every complete DFA can be seen as a DSA. For the converse, we prove that for every DSA of size k, there is a DFA with size at most $2k \cdot (1+2|\Sigma|)$, where Σ is the alphabet (Lemma 1, Theorem 1). This answers the question of how small DSAs can be in comparison to DFAs for a certain language : if n is the size of the minimal DFA for a language L, minimal DSAs for L cannot be smaller than $\frac{n}{2\cdot(1+2|\Sigma|)}$. When the alphabet is large, one could expect smaller sized DSAs. We describe a family of languages L_n, with alphabet size n, for which the minimal DFA has size quadratic in n, whereas size of DSAs is a linear function of n (Lemma 2).
3. We present a method to derive DSAs out of DFAs, a DFA-to-DSA conversion (Section 5). In a nutshell, the derivation procedure selects subsets of DFA-states, and adds transitions labeled with (some of) the acyclic paths between them. Our main technical contribution lies in identifying sufficient conditions on the selected subset of states, so that the derivation procedure preserves the language (Theorem 9).
4. We remark that minimal DSAs need not be unique, and make a surprising observation: the smallest DSA that we derive from the canonical DFA of L need not be a minimal DSA. We find this surprising because (1) firstly, our derivation procedure is surjective: every DSA (satisfying some natural assumptions) can be derived from some corresponding DFA, and in particular, a minimal DSA can be derived from some DFA; (2) the observation suggests that one may need to start with a bigger DFA in order to derive a minimal DSA – so, starting with a bigger DFA may result in a smaller DSA (Section 6).
5. Finally, we show that given a DFA and a number k, deciding if there exists a DSA of size $\leq k$ is NP-complete (Section 7).

Related work. The closest to our work is [8] which introduces DGAs, and gives a procedure to derive DGAs from DFAs. The focus however is on getting DGAs with as few states as possible. The ideas presented in Section 6 of our work, also apply for state-minimality: the same example shows that in order to get fewer states, one may have to start with a bigger DFA. This is in sharp contrast to the DGA setting, where the derivation procedure of [8] yields a minimal DGA (in the number of states) when applied on the canonical DFA. The problem of deriving DGAs with minimal total-size was left open in [8], and continues to remain so, to the best of our knowledge. Expression automata [11] allow regular expressions as transition labels. This model was already considered in [3] to convert automata to regular expressions. Every DFA can be converted to a two state expression automaton with a regular expression connecting them. A model of deterministic Expression automata (DEA) was proposed in [11] with restrictions that limit the expressive power. An algorithm to convert a DFA to a DEA, by repeated state elimination, is proposed in [11]. The resulting DEA is minimal in the number of states. The issue with Expression automata is the high expressivity of the transition condition, that makes states almost irrelevant. On the other hand, DEA have restrictions that make the model less expressive than DFAs. Minimization of NFAs was studied in [15] and shown to be hard.Succinctness of models with different features, like alternation, two-wayness, pebbles, and a notion of concurrency, has been studied in [10].

2 Preliminaries

We fix a finite alphabet Σ. Following standard convention, we write Σ^* for the set of all words (including ε) over Σ, and $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$. For $w \in \Sigma^*$, we write $|w|$ for the length of w, with $|\varepsilon|$ considered to be 0. A word u is a *prefix* of word w if $w = uv$ for some $v \in \Sigma^*$; it is a *proper-prefix* if $v \in \Sigma^+$. Observe that ε is a

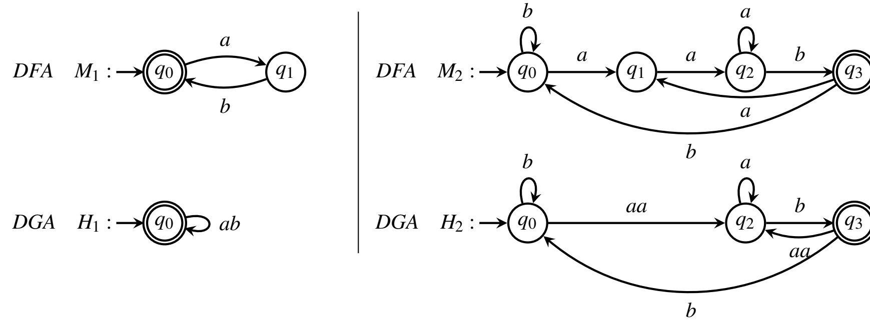


Figure 2: Examples of DFAs and corresponding DGAs, over alphabet $\{a,b\}$.

prefix of every word. A set of words W is said to be a *prefix-free set* if no word in W is a prefix of another word in W. A word u is a *suffix* (resp. *proper-suffix*) of w if $w = vu$ for some $v \in \Sigma^*$ (resp. $v \in \Sigma^+$).

A *Deterministic Finite Automaton (DFA)* M is a tuple $(Q, \Sigma, q^{init}, \delta, F)$ where Q is a finite set of states, $q^{init} \in Q$ is the initial state, $F \subseteq Q$ is a set of accepting states, and $\delta : Q \times \Sigma \to Q$ is a partial function describing the transitions. If δ is complete, the automaton is said to be a complete DFA. Else, it is called a trim DFA. The run of DFA M on a word $w = a_1 a_2 \dots a_n$ (where $a_i \in \Sigma$) is a sequence of transitions $(q_0, a_1, q_1)(q_1, a_2, q_2) \dots (q_{n-1}, a_n, q_n)$ where each $(q_i, a_{i+1}, q_{i+1}) \in \delta$ for $0 \leq i < n$, and $q_0 = q^{init}$, the initial state of M. The run is accepting if $q_n \in F$. If the DFA is complete, every word has a unique run. On a trim DFA, each word either a has unique run, or it has no run. The language $\mathscr{L}(M)$ of DFA M, is the set of words for which M has an accepting run.

We will now recall some useful facts about minimality of DFAs. Here, by minimality, we mean DFAs with the least number of states. Every complete DFA M induces an equivalence $\sim_M$ over words: $u \sim_M v$ if M reaches the same state on reading both u and v from the initial state. In the case of trim DFAs, this equivalence can be restricted to set of prefixes of words in $\mathscr{L}(M)$. For a regular language L, we have the Nerode equivalence: $u \approx_L v$ if for all $w \in \Sigma^*$, we have $uw \in L$ iff $vw \in L$. By the well-known Myhill-Nerode theorem (see [14] for more details), there is a canonical DFA M_L with the least number of states for L, and $\sim_{M_L}$ equals the Nerode equivalence $\approx_L$. Furthermore, every DFA M for L is a *refinement* of M_L: $u \sim_M v$ implies $u \sim_{M_L} v$. If two words reach the same state in M, they reach the same state in M_L.

A *Deterministic Generalized Automaton (DGA)* [8] H is given by $(Q, \Sigma, q^{init}, E, F)$ where Q, q^{init}, F mean the same as in DFA, and $E \subseteq Q \times \Sigma^+ \times Q$ is a finite set of edges labeled with words from Σ^+. For every state q, the set $\{\alpha \mid (q, \alpha, q') \in E\}$ is a prefix-free set. A run of DGA H on a word w is a sequence of edges $(q_0, \alpha_1, q_1)(q_1, \alpha_2, q_2) \dots (q_{n-1}, \alpha_n, q_n)$ such that $w = \alpha_1 \alpha_2 \dots \alpha_n$, with q_0 being the initial state. As usual, the run is accepting if $q_n \in F$. Due to the property of the set of outgoing labels being a prefix-free set, there is a atmost one run on every word. The language $\mathscr{L}(H)$ is the set of words with an accepting run. Figure 2 gives examples of DFAs and corresponding DGAs.

It was shown in [8] that there is no unique smallest DGA. The paper defines an operation to suppress states and create longer labels. A state of a DGA is called *superflous* if it is neither the initial nor final state, and it has no self-loop. For example, in Figure 2, in M_1 and M_2, state q_1 is superfluous. Such states can be removed, and every pair $p \xrightarrow{\alpha} q$ and $q \xrightarrow{\beta} r$ can be replaced with $p \xrightarrow{\alpha\beta} r$. This operation is extended to a set of states: given a DGA H, a set of states S, a DGA $\mathscr{S}(H, S)$ is obtained by suppressing states of S, one after the other, in any arbitrary order. For correctness, there should be no cycle in the induced subgraph of H restricted to S. The paper proves that minimal DGAs (in number of states) can be derived by suppressing states, starting from the canonical DFA.

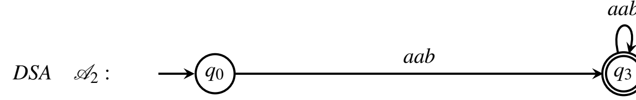


Figure 3: DSA $\mathscr{A}_2$ accepts $L_2 = \Sigma^* aab$, with $\Sigma = \{a,b\}$.

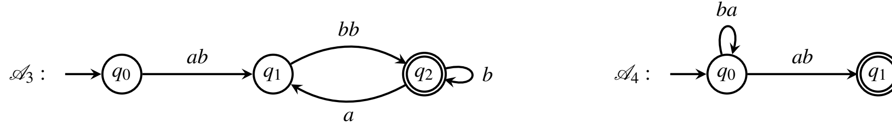


Figure 4: $\mathscr{A}_3$ accepts $L_3 = \Sigma^* ab \Sigma^* bb$ and $\mathscr{A}_4$ accepts $L_4 = (b^* ba)^* a^* ab$.

3 A new automaton model – DSA

We have seen an example of a deterministic suffix automaton in Figure 1. A DSA consists of a set of states, and a finite set of outgoing labels at each state. On an input word w, the DSA finds the earliest prefix which ends with an outgoing label of the initial state, erases this prefix and goes to the target state of the transition with the matching label. Now, the DSA processes the rest of the word from this new state in the same manner. In this section, we will formally describe the syntax and semantics of DSA.

We start with some more examples. Figure 3 shows a DSA for $L_2 = \Sigma^* aab$, the same language as the automata M_2 and H_2 of Figure 2. At q_0, DSA $\mathscr{A}_2$ waits for the first occurrence of *aab* and as soon as it sees one, it transitions to q_3. Here, it waits for further occurrences of *aab*. For instance, on the word *abbaabbbaab*, it starts from q_0 and reads until *abbaab* to move to q_3. Then, it reads the remaining *bbaab* to loop back to q_3 and accepts. On a word *baabaa*, the automaton moves to q_3 on *baab*, and continues reading *aa*, but having nowhere to move, it makes no transition and rejects the word. Consider another language $L_3 = \Sigma^* ab \Sigma^* bb$ on the same alphabet Σ. A similar machine (as $\mathscr{A}_2$) to accept L_3 would look like $\mathscr{A}_3$ depicted in Fig. 4. For example, on the word *abbbb*, it would read until *ab* and move from q_0 to q_1, read further until *bb* and move to q_2, then read b and move back to q_2 to accept. We can formally define such machines as automata that transition on suffixes, or suffix-reading automata.

Definition 1 (DSA)**.** *A deterministic suffix-reading automaton (DSA) $\mathscr{A}$ is a tuple $(Q, \Sigma, q^{init}, \Delta, F)$ where Q is a finite set of states, Σ is a finite alphabet, $q^{init} \in Q$ is the initial state, $\Delta \subseteq Q \times \Sigma^+ \times Q$ is a finite set of transitions, $F \subseteq Q$ is a set of accepting states. For a state $q \in Q$, we define* $\mathrm{Out}(q) := \{\alpha \mid (q, \alpha, q') \in \Delta \text{ for some } q' \in Q\}$ *for the set of labels present in transitions out of q. No state has two outgoing transitions with the same label: if $(q, \alpha, q') \in \Delta$ and $(q, \alpha, q'') \in \Delta$, then $q' = q''$.*

The (total) size $|\mathscr{A}|$ of DSA $\mathscr{A}$ is defined as the sum of the number of states, the number of transitions, and the size $|\mathrm{Out}(q)|$ for each $q \in Q$, where $|\mathrm{Out}(q)| := \sum_{\alpha \in \mathrm{Out}(q)} |\alpha|$.

As mentioned earlier, at a state q the automaton waits for a word that ends with one of its outgoing labels. If more than one label matches,‘ then the transition with the longest label is taken. For example, consider the DSA in Figure 1. At state s_1 on reading $fghendif$, both the `if` and `endif` transitions match. The longest match is `endif` and therefore the DSA moves to s_0. This gives a deterministic behaviour to the DSA. More precisely: at a state q, it reads w to fire (q, α, q') if α is the longest word in $\mathrm{Out}(q)$ which is a suffix of w, and no proper prefix of w has any label in $\mathrm{Out}(q)$ as suffix. We call this a ‘move’ of the DSA. For example, consider $\mathscr{A}_4$ of Figure 4 as a DSA. Let us denote $t := (q_0, ab, q_1)$ and $t' := (q_0, ba, q_1)$. We have moves (t, ab), (t, aab), $(t, aaab)$, and (t', ba), (t', bba), etc. In order to make a move on t, the word should end with *ab* and should have neither *ab* nor *ba* in any of its proper prefixes.

Definition 2. *A* move *of DSA $\mathscr{A}$ is a pair (t,w) where $t = (q,\alpha,q') \in \Delta$ is a transition of $\mathscr{A}$ and $w \in \Sigma^+$ such that*

- *α is the longest word in $\mathrm{Out}(q)$ which is a suffix of w, and*
- *no proper prefix of w contains a label in $\mathrm{Out}(q)$ as suffix.*

A move (t,w) denotes that at state q, transition t gets triggered on reading word w. We will also write $q \xrightarrow[\alpha]{w} q'$ for the move (t,w).

Whether a word is accepted or rejected is determined by a 'run' of the DSA on it. Naturally the set of words with accepting runs gives the language of the DSA. Moreover, due to our "move" semantics, there is a unique run for every word.

Definition 3. *A run of $\mathscr{A}$ on word w, starting from a state q, is a sequence of moves that consume the word w, until a (possibly empty) suffix of w remains for which there is no move possible: formally, a run is a sequence $q = q_0 \xrightarrow[\alpha_0]{w_0} q_1 \xrightarrow[\alpha_1]{w_1} \cdots \xrightarrow[\alpha_{m-1}]{w_{m-1}} q_m \xrightarrow{w_m}$ such that $w = w_0 w_1 \dots w_{m-1} w_m$, and $q_m \xrightarrow{w_m}$ denotes that there is no move using any outgoing transition from q_m on w_m or any of its prefixes. The run is accepting if $q_m \in F$ and $w_m = \varepsilon$ (no dangling letters in the end). The language $\mathscr{L}(\mathscr{A})$ of $\mathscr{A}$ is the set of all words that have an accepting run starting from the initial state q^{init}.*

4 Comparison with DFA and DGA

Every complete DFA can be seen as an equivalent DSA — since $\mathrm{Out}(q) = \Sigma$ for every state, the equivalent DSA is forced to move on each letter, behaving like the DFA that we started off with. For the DSA-to-DFA direction, we associate a specific DFA to every DSA, as follows. The idea is to replace transitions of a DSA with a string matching DFA for $\mathrm{Out}(q)$ at each state. Figure 5 gives an example. The intermediate states correspond to proper prefixes of words in $\mathrm{Out}(q)$.

Definition 4 (Tracking DFA for a DSA.)**.** *For a DSA $\mathscr{A} = (Q^{\mathscr{A}}, \Sigma, q_{in}^{\mathscr{A}}, \Delta^{\mathscr{A}}, F^{\mathscr{A}})$, we give a DFA $M_{\mathscr{A}}$, called its* tracking DFA. *For $q \in Q^{\mathscr{A}}$, let $\overline{\mathrm{Out}}(q)$ be the set of all prefixes of words in $\mathrm{Out}(q)$. States of $M_{\mathscr{A}}$ are given by: $Q^M = \bigcup_{q \in Q^{\mathscr{A}}} \{(q,\beta) \mid \beta \in \overline{\mathrm{Out}}(q)\} \cup q_{copy}$.*

The initial state is $(q_{in}^{\mathscr{A}}, \varepsilon)$ and final states are $\{(q,\varepsilon) \mid q \in F^{\mathscr{A}}\}$. Transitions are as below: For every $q \in Q^{\mathscr{A}}, \beta \in \overline{\mathrm{Out}}(q), a \in \Sigma$, let β' be the longest word in $\overline{\mathrm{Out}}(q)$ s.t β' is a suffix of βa.

- *$(q,\beta) \xrightarrow{a} (q',\varepsilon)$ if $(q,\beta',q') \in \Delta^{\mathscr{A}}$, ($q'$ may equal q also)*
- *$(q,\beta) \xrightarrow{a} (q,\beta')$ if $\beta' \notin \mathrm{Out}(q)$ and $\beta' \neq \varepsilon$,*
- *$(q,\beta) \xrightarrow{a} q_{copy}$ if $\beta' = \varepsilon$,*
- *$q_{copy} \xrightarrow{a} s$, if $(q,\varepsilon) \xrightarrow{a} s$ according to the above (same outgoing transitions).*

Intuitively, the tracking DFA implements the transition semantics of DSAs. Starting at (q,ε), the tracking DFA moves along states marked with q as long as no label of $\mathrm{Out}(q)$ is seen as a suffix. For all such words, the tracking DFA maintains the longest word among $\overline{\mathrm{Out}}(q)$ seen as a suffix so far. For instance, in Figure 5, at q on reading word aab, the DFA on the right is in state ab (which is the equivalent of (q,ab) in the tracking DFA definition).

Lemma 1. *For every DSA $\mathscr{A}$, the language $\mathscr{L}(\mathscr{A})$ equals the language $\mathscr{L}(M_{\mathscr{A}})$ of its tracking DFA.*

Lemma 1 and the fact that every complete DFA is also a DSA, prove that DSAs recognize regular languages. We will now compare succinctness of DSA wrt DFA and DGA. We start with a family of languages for which DSAs are concise.

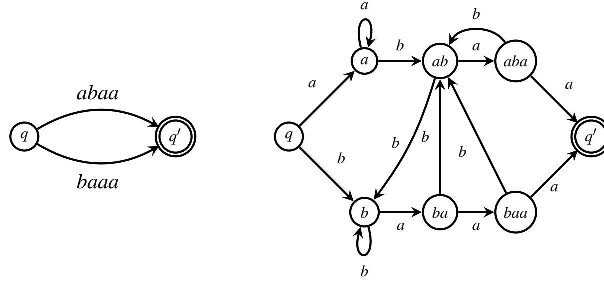


Figure 5: A DSA on the left, and the corresponding DFA for matching the strings *abaa* and *baaa*.

Lemma 2. *Let $\Sigma = \{a_1, a_2, \ldots, a_n\}$ for some $n \geq 1$. Consider the language $L_n = \Sigma^* a_1 a_2 \ldots a_n$. There is a DSA for this language with size $4+2n$. Any DFA for L_n has size at least n^2.*

We now state the final result of this section, which summarizes the size comparison between DSAs, DFAs, DGAs. For the comparison to DFAs, we use the fact that every DSA of size k can be converted to its tracking DFA, which has atmost $2k$ states. Therefore, size of the tracking DFA is bounded by $2k$ (states) $+2k \cdot |\Sigma|$ (edges) $+2k \cdot |\Sigma|$ (label length), which comes to $2k(1+2|\Sigma|)$.

Theorem 1. *For a regular language L, let $n_F^{cmp}, n_F^{trim}, n_G^{trim}, n_S$ denote the size of the minimal complete DFA, minimal trim DFA, minimal trim DGA and minimal DSA respectively, where size is counted as the sum of the number of states, edges and length of edge labels, in all the automata. We have:*

1. $\dfrac{n_F^{cmp}}{2(1+2|\Sigma|)} \leq n_S \leq n_F^{cmp}$
2. *no relation between n_S and n_F^{trim}, n_G^{trim}: there is a language for which n_S is the smallest, and another language for which n_S is the largest of the three.*

5 Suffix-tracking sets – obtaining DSA from DFA

For DGAs, a method to derive smaller DGAs by suppressing states was recalled in Section 2. Our goal is to investigate a similar procedure for DSAs. The DSA model creates new challenges. Suppressing states may not always lead to smaller automata (in total size). Figure 6 illustrates an example where suppressing states leads to an exponentially larger automaton, due to the exponentially many paths created. But, suppressing states may sometimes indeed be useful: in Figure 7, the DFA on the left is performing a string matching to deduce the pattern *ab*. On seeing *ab*, it accepts. Any extension is rejected. This is succinctly captured by the DSA on the right. Notice that the DSA is obtained by suppressing states q_1 and q_3. So, suppressing states may sometimes be useful and sometimes not. In [8], the focus was on getting a DGA with minimal number of states, and hence suppressing states was always useful.

More importantly, when can we suppress states? DGAs cannot "ignore" parts of the word. This in particular leads to the requirement that a state with a self-loop cannot be suppressed. DSAs have a more sophisticated transition semantics. Therefore, the procedure to suppress states is not as simple. This is the subject of this section. We deviate from the DGA setting in two ways: we will select a subset of good states from which we can construct a DSA (essentially, this means the rest of the states are suppressed); secondly, our starting point will be complete DFA, on which we make the choice of states (in DGAs, one could start with any DGA and suppress states). Our procedure can be broken down into two steps: (1) Start from a complete DFA, select a subset of states and build an induced DSA by connecting states using acyclic paths between them; (2) Remove some useless transitions.

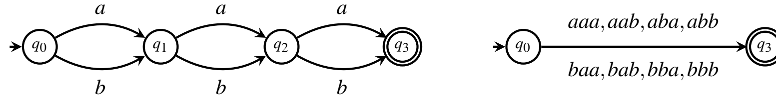


Figure 6: Suppressing states can add exponentially many labels and increase total size.

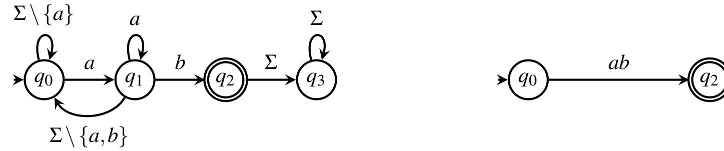


Figure 7: Suppressing states can sometimes reduce total size

Building an induced DSA. We start with an illustrative example. Consider DFA M in Figure 8. The DSA on the right of the figure shows such an induced DSA obtained by marking states $\{q_0, q_2\}$ and connecting them using simple paths. Notice that the language of the induced DSA and the original DFA are same in this case. Intuitively, all words that end with an a land in q_1. Hence, q_1 can be seen to "track" the suffix a. Now, consider Figure 9. We do the same trick, by marking states $\{q_0, q_2\}$ and inducing a DSA. Observe that the DSA does not accept aba, and hence is not language equivalent. When does a subset of states induce a language equivalent DSA? Roughly, this is true when the states that are suppressed track "suitable suffixes" (a reverse engineering of the tracking DFA construction of Definition 4). As we will see, the suitable suffixes will be the simple paths from the selected states to the suppressed states. We begin by formalizing these ideas and then present sufficient conditions that ensure language equivalence of the resulting DSA.

Definition 5 (Simple words)**.** *Consider a complete DFA $M = (Q, \Sigma, q^{init}, \Delta, F)$. Let $S \subseteq Q$ be a subset of states, and $p, q \in Q$. We define $\mathsf{SP}(p \rightsquigarrow q, S)$, the* simple words from p to q modulo S, *as the set of all words $a_1 a_2 \dots a_n \in \Sigma^+$ such that there is a path: $p = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots p_{n-1} \xrightarrow{a_n} p_n = q$ in M where*

- *no intermediate state belongs to S: $\{p_1, \dots, p_{n-1}\} \subseteq Q \setminus S$, and*
- *there is no intermediate cycle: if $p_i = p_j$ for some $0 \le i < j \le n$, then $p_i = p_0$ and $p_j = p_n$.*

We write $\mathsf{SP}(p, S)$ for $\bigcup_{q \in Q} \mathsf{SP}(p \rightsquigarrow q, S)$, the set of all simple words modulo S, emanating from p.

For example, in Figure 8, with $S = \{q_0, q_2\}$, we have $\mathsf{SP}(q_0 \rightsquigarrow q_1, S) = \{a\}$, $\mathsf{SP}(q_0 \rightsquigarrow q_0, S) = \{b\}$ and $\mathsf{SP}(q_0 \rightsquigarrow q_2, S) = ab$. These are the same in Figure 9, except $\mathsf{SP}(q_0 \rightsquigarrow q_2, S) = aa$.

Fix a complete DFA M for this section. A DSA can be 'induced' from M using S, by fixing states to be S (initial and final states retained) and transitions to be the simple words modulo S connecting them i.e. $p \xrightarrow{\sigma} q$ if $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$ (Figure 8).

Definition 6 (Induced DSA)**.** *Given a DFA M and a set S of states in M that contains the initial and final states, we define the induced DSA of M (using S). The states of the induced DSA are given by S. The initial and final states are the same as in M. The transitions are given by the simple words modulo S i.e. $p \xrightarrow{\sigma} q$ if $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$, for every pair of states $p, q \in S$.*

The induced DSA may not be language-equivalent (Figure 9); to ensure that, we need to check some conditions. Here is a central definition.

Definition 7 (Suffix-compatible transitions)**.** *Fix a subset $S \subseteq Q$. A transition $q \xrightarrow{a} u$ is suffix-compatible w.r.t. S if either of $q, u \in S$ **OR** $\forall p \in S$, and for every $\sigma \in \mathsf{SP}(p \rightsquigarrow q, S)$, there is an $\alpha \in \mathsf{SP}(p \rightsquigarrow u, S)$ s.t.:*

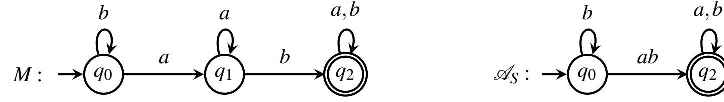


Figure 8: DFA M and an equivalent DSA $\mathscr{A}_S$ 'induced' with $S = \{q_0, q_2\}$.

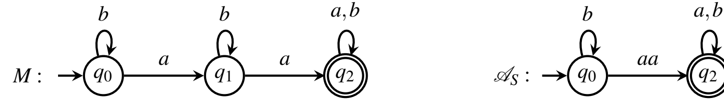


Figure 9: DFA M and DSA $\mathscr{A}_S$ 'induced' with $S = \{q_0, q_2\}$. Not equivalent.

- *α is a suffix of σa, and*
- *moreover, α is the longest suffix of σa among words in $\mathsf{SP}(p,S)$.*

Note that a transition $q \xrightarrow{a} u$ is trivially suffix-compatible if $q \in S$ or $u \in S$. The rest of the condition only needs to be checked when both of $q,u \notin S$. In Figure 9, we find the self-loop at q_1 to not be suffix-compatible: we have $S = \{q_0,q_2\}$, and $\mathsf{SP}(q_0 \rightsquigarrow q_1,S) = \{a\}$, $\mathsf{SP}(q_0,S) = \{b,a,ab\}$; the transition $q_1 \xrightarrow{b} q_1$ is not suffix-compatible since there is no suffix of ab in $\mathsf{SP}(q_0 \rightsquigarrow q_1,S)$. Whereas in Figure 8, the loop is labeled a instead of b. The transition $q_1 \xrightarrow{a} q_1$ is suffix-compatible, since the longest suffix of aa among $\mathsf{SP}(q_0,S)$ is a and it is present in $\mathsf{SP}(q_0 \rightsquigarrow q_1,S)$. Let us take the DFA in the right of Figure 5, and let $S = \{q,q'\}$. Here are some of the simple path sets: $\mathsf{SP}(q \rightsquigarrow ab,S) = \{ab,bab,baab\}$, $\mathsf{SP}(q \rightsquigarrow aba,S) = \{aba,baba,baaba\}$. Consider the transition $aba \xrightarrow{b} ab$. It can be verified that for every $\sigma \in \mathsf{SP}(q \rightsquigarrow aba,S)$, the longest suffix of the extension σa, among simple paths out of q, indeed lies in the state ab. In fact, all transitions satisfy suffix-compatibility w.r.t. the chosen set S.

The suffix-compatibility condition is described using simple paths to states. It requires that every transition take each simple word reaching its source to the state tracking the longest suffix of its one-letter extension. This condition on simple paths, transfers to all words, that circle around the suppressed states. In Figure 5, this property can be verified by considering the word $bbabab$ and its run: $q \xrightarrow{b} b \xrightarrow{b} b \xrightarrow{a} ba \xrightarrow{b} ab \xrightarrow{a} aba \xrightarrow{b} ab$. At each step, the state reached corresponds to the longest suffix among the simple words out of q. In the next two lemmas, we prove this claim.

We will use a special notation: for a state $p \in S$, we write $\mathrm{Out}(p,S)$ for $\bigcup_{r \in S} \mathsf{SP}(p \rightsquigarrow r,S)$; these are the simple words that start at p and end in some state r of S. Notice that these are the words that appear as transitions in the induced DSA. In particular, $\mathrm{Out}(p)$ in the induced DSA equals $\mathrm{Out}(p,S)$.

Lemma 3. *Let S be a set of states such that every transition of M is suffix-compatible w.r.t. S. Pick $p \in S$, and let $w \in \Sigma^+$ be a word with a run $p = p_0 \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \dots p_{n-1} \xrightarrow{w_n} p_n$ such that the intermediate states $p_1,\dots,p_{n-1}$ belong to $Q \setminus S$. The state p_n may or may not be in S. Then:*

- *no proper prefix of w contains any word from $\mathrm{Out}(p,S)$ as suffix, and*
- *there is $\alpha \in \mathsf{SP}(p \rightsquigarrow p_n,S)$ such that α is the longest suffix of w among words in $\mathsf{SP}(p,S)$.*

Lemma 4. *Let S be a set of states such that every transition of M is suffix-compatible w.r.t. S. Let $p \in S$, and $w \in \Sigma^+$ be a word such that no proper prefix of w contains a word in $\mathrm{Out}(p,S)$ as suffix. Then:*

- *The run of M starting from p, is of the form $p \xrightarrow{w_1} p_1 \xrightarrow{w_2} p_2 \dots p_{n-1} \xrightarrow{w_n} p_n$ where $\{p_1,\dots,p_{n-1}\} \subseteq Q \setminus S$ (notice that we have not included p_n, which may or may not be in S).*

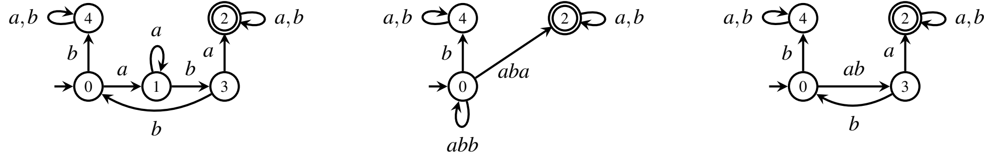


Figure 10: A DFA, a non-equivalent DSA and an equivalent induced DSA.

- *the longest suffix of w, among* $\mathsf{SP}(p,S)$ *lies in* $\mathsf{SP}(p \rightsquigarrow p_n,S)$.

Suffix-compatibility alone does not suffice to preserve the language. In Figure 10, consider $S = \{0,2,4\}$. Every transition is suffix-compatible w.r.t. S. The DSA induced using S is shown in the middle. Notice that it is not language equivalent, due to the word *aba* for instance. The run of *aba* looks as follows: $0 \xrightarrow{ab} 4 \xrightarrow{b} 4$. The expected run was $0 \xrightarrow{aba} 2$, but that does not happen since there is a shorter prefix with a matching transition. Even though, we have suffix-compatibility, we need to ensure that there are no "conflicts" between outgoing patterns. This leads to the next definition.

Definition 8 (Well-formed set)**.** *A set of states* $S \subseteq Q$ *is well-formed if there is no* $p \in S, q \in S$ *and* $q' \notin S$*, with a pair of words* $\alpha \in \mathsf{SP}(p \rightsquigarrow q,S)$ *(simple word to a state in S) and* $\beta \in \mathsf{SP}(p \rightsquigarrow q',S)$ *(simple word to a state not in S) such that* α *is a suffix of* β*.*

We observe that the set $S = \{0,2,4\}$ is not well-formed since $b \in \mathsf{SP}(0 \rightsquigarrow 4,S), ab \in \mathsf{SP}(0 \rightsquigarrow 3,S)$ and b is a suffix of ab. Whereas $S' = \{0,2,3,4\}$ is both suffix-tracking, and well-formed, and induces an equivalent DSA. On the word *aba*, the run on the DSA would be $0 \xrightarrow{ab} 3 \xrightarrow{a} 2$. The first move $0 \xrightarrow{ab} 3$ applies the longest match criterion, and the transition since ab is a longer suffix than b. This was not possible before since $3 \notin S$. It turns out that the two conditions — suffix-compatibility and well-formedness — are sufficient to induce a language equivalent DSA.

Definition 9 (Suffix-tracking sets)**.** *A set of states* $S \subseteq Q$ *is suffix-tracking if it contains the initial and accepting states, and*

1. *every transition of M is suffix-compatible w.r.t. S,*
2. *and S is well-formed.*

All these notions lead to the main theorem of this section.

Theorem 2. *Let S be a suffix-tracking set of complete DFA M, and let* $\mathscr{A}_S$ *be the DSA induced using S. Then:* $\mathscr{L}(\mathscr{A}_S) = \mathscr{L}(M)$

Proof. Pick $w \in \mathscr{L}(M)$. There is an accepting run $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_n} q_n$ of M on w. By Definition 9, we have $q_0, q_n \in S$. Let $1 \le i \le n$ be the smallest index greater than 0, such that $q_i \in S$. Consider the run segment $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots \xrightarrow{w_i} q_i$. By Lemma 3, and by the definition of induced DSA 6, no transition of $\mathscr{A}_S$ out of q_0 is triggered until $w_1 \dots w_{i-1}$, and then on reading w_i, the transition $q_0 \xrightarrow{\alpha} q_i$ is triggered, where $\alpha \in \mathsf{SP}(p \rightsquigarrow q,S)$, and α is also the longest suffix of $w_1 \dots w_i$ among $\mathsf{SP}(p,S)$. In particular, it is the longest suffix among outgoing labels from q_0 in $\mathscr{A}_S$. This shows there is a move $q_0 \xrightarrow[\alpha]{w_1 \dots w_i} q_i$ in $\mathscr{A}_S$. Repeat this argument on rest of the run $q_i \xrightarrow{w_{i+1}} q_{i+1} \xrightarrow{w_{i+1}} \dots \xrightarrow{w_n} q_n$ to extend the run of $\mathscr{A}_S$ on the rest of the word. This shows $w \in \mathscr{L}(\mathscr{A}_S)$.

Pick $w \in \mathscr{L}(\mathscr{A}_S)$. There is an accepting run ρ of $\mathscr{A}_S$ starting at the initial state q_0. Consider the first move $q_0 \xrightarrow[\alpha]{w_1 \dots w_i} q_i$ of $\mathscr{A}_S$ on the word. By the semantics of a move (Definition 2) and Lemma 4, we obtain a run $q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \dots q_{i-1} \xrightarrow{w_i} q_i$ of M where the intermediate states $q_1, \dots, q_{i-1}$ lie in $Q \setminus S$. We apply this argument for each move ρ in the accepting run of $\mathscr{A}_S$ to get an accepting run of M. ☐

Figure 11: Illustrating bigger-suffix transitions and when they are useless

Removing some useless transitions. Let us now get back to Figure 5 to see if we can derive the DSA on the left from the DFA on the right (assuming q is the initial state). As seen earlier, the set $S = \{q, q'\}$ is suffix tracking. It is also well formed since *baaa* is not a suffix of any prefix of *abaa* and vice-versa. The DSA $\mathscr{A}_S$ induced using q and q' will have the set of words in $\mathsf{SP}(q \rightsquigarrow q', S)$ as transitions between q and q'. Both *abaa* and *baaa* belong to $\mathsf{SP}(q \rightsquigarrow q', S)$. However, there are some additional simple words: for instance, *abbaaa*. Notice that *baaa* is a suffix of *abbaaa*, and therefore even if we remove the transition on *abbaaa*, there will be a move to q' via $q \xrightarrow{baaa} q'$. This tempts us to use only the suffix-minimal words in the transitions of the induced DSA. This is not always safe, as we explain below. We show how to carefully remove "bigger-suffix-transitions".

Consider the DSA on the left in Figure 11. If *caba* is removed, the moves which were using *caba* can now be replaced by *ba* and we still have the same pair of source and target states. Consider the picture on the right of the same figure. There is an outgoing edge to a different state on *aba*. Suppose we remove *caba*. The word *caba* would then be matched by the longer suffix *aba* and move to a different state. Another kind of useless transitions are some of the self-loops on DSAs. In Figure 8, the self-loop on b at q_0 can be removed, without changing the language. This can be generalized to loops over longer words, under some conditions.

Definition 10. *Let $\mathscr{A}$ be a DSA, q, q' be states of $\mathscr{A}$ and $t := q \xrightarrow{\alpha} q'$ be a transition.*

We call t a bigger-suffix-transition *if there exists another transition (q, β, q') with β a suffix of α.*

If there is a transition $t' := q \xrightarrow{\gamma} q''$ ($q'' \neq q'$), such that β is a suffix of γ, and γ is a suffix of α, we call t useful. *A bigger-suffix-transition is called* useless *if it is not useful.*

We will say that t is a useless self-loop *if $q = q'$, q is not an accepting state, and no suffix of α is a prefix of some outgoing label in* $\mathrm{Out}(q)$.

In Figure 11, for the automaton on the left, the transition on *caba* is useless. Whereas for the DSA on the right, *caba* is a bigger-suffix-transition, but it is useful. The self-loop on q_0 in Figure 8 is useless, but the loop on q_0 in Figure 4 is useful. Lemmas 5 and 6 prove correctness of removing useless transitions.

Lemma 5. *Let $\mathscr{A}$ be a DSA, and let $t := q \xrightarrow{\alpha} q'$ be a useless bigger-suffix-transition. Let $\mathscr{A}'$ be the DSA obtained by removing t from $\mathscr{A}$. Then, $L(\mathscr{A}) = L(\mathscr{A}')$.*

Proof. To show $L(\mathscr{A}) \subseteq L(\mathscr{A}')$. Let $w \in L(\mathscr{A})$ and let $q_0 \xrightarrow[\alpha_0]{w_0} q_1 \xrightarrow[\alpha_1]{w_1} \cdots \xrightarrow[\alpha_{m-1}]{w_{m-1}} q_m$ be an accepting run. If no (q_i, α_i, q_{i+1}) equals (q, α, q'), then the same run is present in S', and hence $w \in L(S')$. Suppose $(q_j, \alpha_j, q_{j+1}) = (q, \alpha, q')$ for some j. So, the word w_j ends with α. As (q, α, q') is a bigger-suffix-transition, there is another (q, β, q') such that $\beta \sqsubseteq_{\mathsf{sf}} \alpha$. Therefore, the word w_j also ends with β. Since there was no transition matching a proper prefix of w_j, the same will be true at $\mathscr{A}'$ as well, since it has fewer transitions. It remains to show that $q_j \xrightarrow[\beta]{w_j} q_{j+1}$ is a move. The only way this cannot happen is if there is a $q \xrightarrow{\gamma} q''$ with $\beta \sqsubseteq_{\mathsf{sf}} \gamma \sqsubseteq_{\mathsf{sf}} \alpha$. But this is not possible since $q \xrightarrow{\alpha} q'$ is a useless bigger-suffix transition. Therefore, every move using (q, α, q') in $\mathscr{A}$ will now be replaced by (q, β, q') in $\mathscr{A}'$. Hence we get an accepting run in $\mathscr{A}'$, implying $w \in L(\mathscr{A}')$.

To show $L(\mathscr{A}') \subseteq L(\mathscr{A})$. Consider $w \in L(\mathscr{A}')$ and an accepting run $q_0 \xrightarrow[\alpha_0]{w_0} q_1 \xrightarrow[\alpha_1]{w_1} \cdots \xrightarrow[\alpha_{m-1}]{w_{m-1}} w_m$ in $\mathscr{A}'$. Notice that if $q \xrightarrow[\beta]{w_j} q'$ is a move in $\mathscr{A}'$, the same is a move in $\mathscr{A}$ when $\alpha \not\sqsubseteq_{\mathsf{sf}} w_j$. When $\alpha \sqsubseteq_{\mathsf{sf}} w_j$, then the bigger-suffix-transition $q \xrightarrow{\alpha} q'$ will match and the move $q \xrightarrow[\beta]{w_j} q'$ gets replaced by $q \xrightarrow[\alpha]{w_j} q'$. Hence we will get the same run, except that some of the moves using $q \xrightarrow{\beta} q'$ may get replaced with $q \xrightarrow{\alpha} q'$. ☐

For the correctness of removing useless self-loops, we assume that the DFA that we obtain is well-formed (Definition 12) and has no useless bigger-suffix-transitions. The induced DSA that we obtain from suffix-tracking sets is indeed well-formed. Starting from this induced DSA, we can first remove all useless bigger-suffix-transitions, and then remove the useless self-loops.

Lemma 6. *Let $\mathscr{A}$ be a well-formed DSA that has no removable bigger-suffix-transitions. Let $t := (q,\alpha,q)$ be a removable self-loop. Then the DSA $\mathscr{A}'$ obtained by removing t from $\mathscr{A}$ satisfies $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{A}')$.*

Proof. To show $\mathscr{L}(\mathscr{A}) \subseteq \mathscr{L}(\mathscr{A}')$. Let $w \in \mathscr{L}(\mathscr{A})$ and let $\rho := q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{m-1}} q_m$ be an accepting run. Suppose t matches the segment $q_j \xrightarrow{w_j} q_{j+1}$. Hence $q_j = q_{j+1} = q$. Observe that as q is not accepting, we have $j+1 \neq m$. Therefore there is a segment $q_{j+1} \xrightarrow{w_{j+1}} q_{j+2}$ in the run. We claim that if t is removed, then no transition out of q can match any prefix of $w_j w_{j+1}$.

First we see that no prefix of w_j can be matched, including w_j itself: if at all there is a match, it should be at w_j, and a β that is smaller than α. By assumption, α is not a removable bigger-suffix-transition. Therefore, there is a transition $q \xrightarrow{\gamma} q'$, with $\beta \sqsubseteq_{\mathsf{sf}} \gamma \sqsubseteq_{\mathsf{sf}} \alpha$. This contradicts the assumption that α is a removable self-loop. Therefore there is no match upto w_j.

Suppose some (q,β,q') matches a prefix $w_j u$ such that $\beta = vu$, that is, β overlaps both w_j and w_{j+1}. If $\alpha \sqsubseteq_{\mathsf{sf}} v$, then it violates well-formedness of S since it would be a suffix of a proper prefix (v) of β. This shows $v \sqsubseteq_{\mathsf{sf}} \alpha$ (since both are suffixes of w_j) and $v \sqsubseteq_{\mathsf{pr}} \beta$, contradicting the assumption that t is removable. Therefore, β does not overlap w_j. But then, if β is a suffix of a proper prefix of w_{j+1}, we would not have the segment $q_{j+1} \xrightarrow{w_{j+1}} q_{j+2}$ in the run ρ. Therefore, the only possibility is that we have a segment $q_j \xrightarrow{w_j w_{j+1}} q_{j+2}$. We have fewer occurrences of the removable loop (q,α,q) in the modified run. Repeating this argument for every match of (q,α,q) gives an accepting run of $\mathscr{A}'$. Hence $w \in L(\mathscr{A}')$.

To show $L(\mathscr{A}') \subseteq L(\mathscr{A})$. Let $w \in L(\mathscr{A}')$ and $\rho' := q_0 \xrightarrow{w_0} q_1 \xrightarrow{w_1} \cdots \xrightarrow{w_{m-1}} q_m$ be an accepting run in $\mathscr{A}'$. Suppose $q_j \xrightarrow{w_j} q_{j+1}$ is matched by (q,β,q'). Let $w_j = vu$ with $\alpha \sqsubseteq_{\mathsf{sf}} v$. Then the removable-self-loop (q,α,q) will match the prefix v. Suppose β overlaps with both v and u, that is $\beta = \beta' u$. We cannot have $\alpha \sqsubseteq_{\mathsf{sf}} \beta'$ due to well-formedness of $\mathscr{A}$. We cannot have $\beta' \sqsubseteq_{\mathsf{sf}} \alpha$ since this would mean there is a suffix of α which is a prefix of β, violating the removable-self-loop condition. Therefore, β is entirely inside u, that is, $\beta \sqsubseteq_{\mathsf{sf}} u$. Hence in $\mathscr{A}$ the run will first start with $q \xrightarrow{v} q$. Applying the same argument, prefixes of the remaining word where t matches will be matched until there is a part of the word where (q,β,q') matches. This applies to every segment, thereby giving us a run in $\mathscr{A}$. ☐

We now get to the core definition of this section, which tells how to derive a DSA from a DFA, using the methods developed so far.

Definition 11 (DFA-to-DSA derivation)**.** *A DSA is said to be* derived from *DFA M using $S \subseteq Q$, if it is identical to an induced DSA of M (using S) with all useless transitions removed.*

By Theorem 2 and Lemma 5, we get the following result.

Theorem 3. *Every DSA that is derived from a complete DFA is language equivalent to it.*

Figure 12: Minimal DSA is not unique

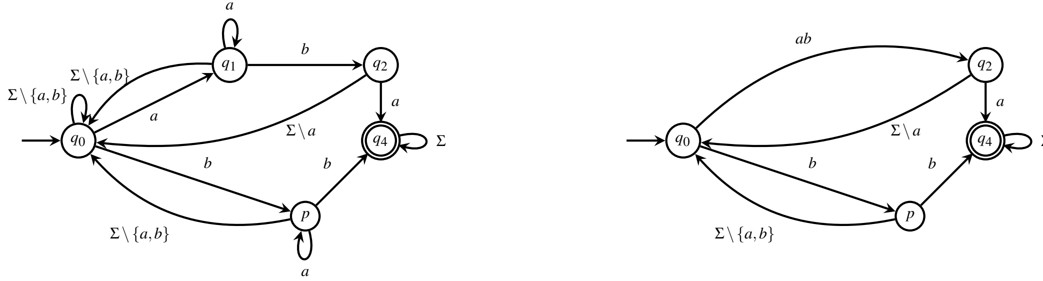


Figure 13: DFA M^* on the left and a derived DSA $\mathscr{A}_S^*$ with $S = \{q_0, q_2, q_4, p\}$ on the right.

6 Minimality, some observations and some challenges

Theorem 1 shows that we can not expect DSAs to be smaller than (trim) DFAs or DGAs in general. However, Lemma 2 and Figure 1 show that there are cases where DSAs are smaller and more readable. This motivates us to ask the question of how we can find a minimal DSA, that is, a DSA of the smallest (total) size. The first observation is that minimal DSAs need not be unique — see Figure 12. The next simple observation is that a minimal DSA will not have useless transitions since removing them gives an equivalent DSA with strictly smaller size. In fact, we can assume a certain well-formedness condition on the minimal DSAs, in the same spirit as the definition of well-formed sets in our derivation procedure: if there are two transitions $q \xrightarrow{\alpha} q_1$ and $q \xrightarrow{\beta_1 \alpha \beta_2} q_2$, then we can remove the second transition since it will never get fired.

Definition 12 (Well-formed DSA)**.** *A DSA $\mathscr{A}$ is* well-formed *if for every state q, no outgoing label $\alpha \in \mathrm{Out}(q)$ is a suffix of some proper prefix β' of another outgoing label $\beta \in \mathrm{Out}(q)$.*

Any transition violating well-formedness can be removed, without changing the language. Therefore, we can safely assume that minimal DSAs are well-formed. Due to the "well-formedness" property in suffix-tracking sets, the DSAs induced by suffix-tracking sets are naturally well-formed. Since removing useless transitions preserves this property, the DSAs that are derived using our DFA-to-DSA procedure (Definition 11) are well-formed. The next proposition shows that every DSA that is well-formed and has no useless transitions (and in particular, the minimal DSAs) can be derived from the corresponding tracking DFAs.

Proposition 1. *Every well-formed DSA with no useless transitions can be derived from its tracking DFA.*

Proposition 1 says that if we somehow had access to the tracking DFA of a minimal DSA, we will be able to derive it using our procedure. The challenge however is that this tracking DFA may not necessarily be the canonical DFA for the language. In fact, we now show that a smallest DSA that can be derived from the canonical DFA need not be a minimal DSA.

Figure 13 shows a DFA M^*. Observe that M^* is minimal: every pair of states has a distinguishing suffix. Let us now look at DSAs that can be derived from M^*. Firstly, any suffix-tracking set on M^* would contain q_0, q_4 (since they are initial and accepting states). If p is not picked, the transition $p \xrightarrow{a} p$ is not suffix-compatible. Therefore, p should belong to the selected set. If p is picked, and q_2 not picked,

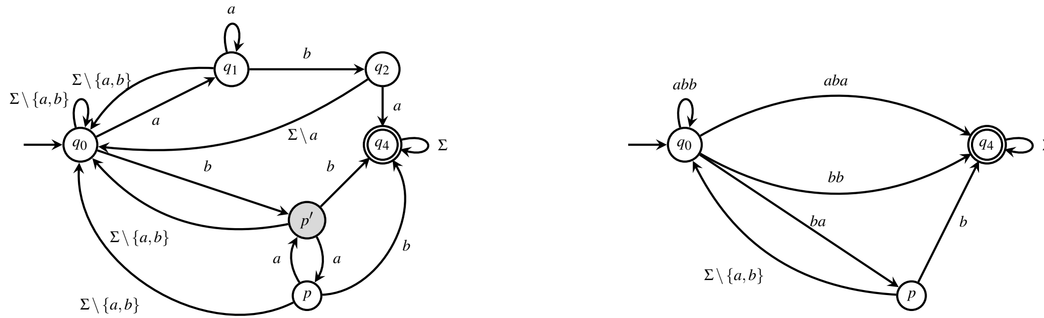


Figure 14: DFA M^{**} on the left and a derived DSA $\mathscr{A}_S^{**}$ with $S = \{q_0, q_2, q_4, p\}$ on the right.

then the set is not well-formed (see Definition 8): the simple word b from q_0 to p is a suffix of the simple word ab to q_2. Therefore, any suffix-tracking set should contain the 4 states q_0, p, q_2, q_4. This set $S = \{q_0, p, q_2, q_4\}$ is indeed suffix-tracking, and the DSA derived using S is shown in the right of Figure 13. The only other suffix-tracking set is the set S' of all states. The DSA derived using S' will have state q_1 in addition, and the transitions $\Sigma \setminus \{a,b\}$. If Σ is sufficiently large, this DSA would have total size bigger than $\mathscr{A}_S^*$. We deduce $\mathscr{A}_S^*$ to be the smallest DSA that can be derived from M^*.

Figure 14 shows DFA M^{**} which is obtained from M^* by duplicating state p to create a new state p', which is equivalent to p. So M^{**} is language equivalent to M^*, but it is not minimal. Here, if we choose p in a suffix-tracking set, the simple word to p is ba, which is not a suffix of ab (the simple word to q_2). Hence, we are not required to add q_2 into the set. Notice that $S = \{q_0, p, q_4\}$ is indeed a suffix-tracking set in M^{**}. The derived DSA $\mathscr{A}_S^{**}$ is shown in the right of the figure. The "heavy" transition on $\Sigma \setminus a$ disappears. There are some extra transition, like $q_0 \xrightarrow{bb} q_4$, but if Σ is large enough, the size of $\mathscr{A}_S^{**}$ will be smaller than $\mathscr{A}_S^*$. This shows that starting from a big DFA helps deriving a smaller DSA, and in particular, the canonical DFA of a regular language may not derive a minimal DSA for the language.

7 Complexity of minimization

The goal of this section is to prove the following theorem.

Theorem 4. *Given a DFA M and positive integer k, deciding whether there exists an equivalent DSA of total size $\leq k$ equivalent to M is NP-complete.*

If k is bigger than the size of the DFA M, then the answer is trivial. Therefore, let us assume that k is smaller than the DFA size. For the NP upper bound, we guess a DSA of total size k, compute its tracking DFA in time $\mathscr{O}(k \cdot |\Sigma|)$ and check for its language equivalence with the given DFA M. This can be done in polynomial-time by minimizing both the DFA and checking for isomorphism.

The rest of the section is devoted to proving the lower bound. We provide a reduction from the minimum vertex cover problem which is a well-known NP-complete problem [16]. A vertex cover of an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ of vertices, such that for every edge $e \in E$, at least one of its end points is in S. The decision problem takes a graph G and a number $k' \geq 1$ as input and asks whether there is a vertex cover of G with size at most k'. Using the graph G, we will construct a DFA M_G over an alphabet Σ_G. We then show that G has a vertex cover of size $\leq k'$ iff M_G has an equivalent DSA with total size $\leq k$ where $k = (k'+2) \times 2\Delta + (2\Delta - 1)$. Here, Δ is a sufficiently large polynomial in $|V|, |E|$ which we will explain later.

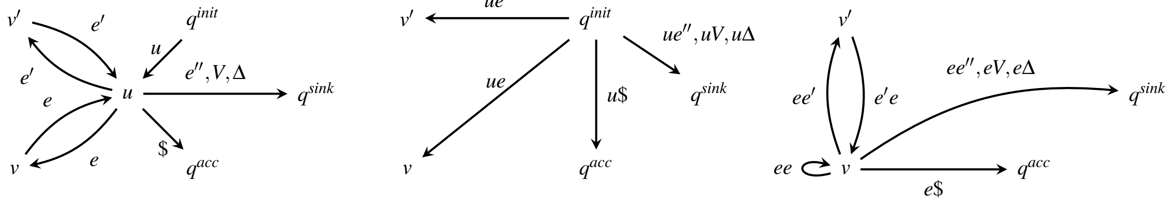


Figure 15: Left: Illustration of the neighbourhood of state u in the DFA M_G. Middle, Right: Transitions induced from q^{init} and v, on removing u.

The alphabet Σ_G is given by $V \cup E \cup \{\$\} \cup D$ where $D = \{1,2,\ldots,\Delta\}$. States of M_G are $V \cup \{q^{init}, q^{sink}, q^{acc}\}$. For simplicity, we use the same notation for v as a vertex in G, v as a letter in Σ_G and v as a state of M_G. The actual role of v will be clear from the context. For every edge $e = (u,v)$, there are two transitions in the automaton: $u \xrightarrow{e} v$ and $v \xrightarrow{e} u$. For every $v \in V$, there are transitions $q^{init} \xrightarrow{v} v$ and $v \xrightarrow{\$} q^{acc}$. This automaton can be completed by adding all missing transitions to the sink state q^{sink}. Figure 15 (left) illustrates the neighbourhood of a state u. The notation e'' stands for any edge that is not incident on u; there is one transition for every such e''. Initial and accepting states are respectively q^{init} and q^{acc}. Let $L_G(u)$ be the set of words that have an accepting run in M_G starting from u as the initial state. If $u \neq v$, $L_G(u) = L_G(v)$ implies (u,v) is an edge and there are no other edges outgoing either from u or v. To avoid this corner case, we restrict the vertex cover problem to connected graphs of 3 or more vertices. Then we have M_G to be a minimal DFA, with no two states equivalent. Here are two main ideas.

Suppressing a state. Suppose state u of M_G is suppressed (i.e. u is not in a suffix-tracking set). In Figure 15, we show the induced transitions from q^{init} and a vertex v. However, some of them will be useless transitions: most importantly, the set of transitions $q^{init} \xrightarrow{u1,u2,\ldots,u\Delta} q^{sink}$ will be useless bigger-suffix-transitions due to $q^{init} \xrightarrow{1,2,\ldots,\Delta} q^{sink}$. Similarly, $v \xrightarrow{e1,e2,\ldots,e\Delta} q^{sink}$ will be removed. There are some more useless bigger-suffix-transitions, like $v \xrightarrow{ee''} q^{sink}$ for some e'' that is not incident on v and u. So from each v, at most $2|E|$ transitions are added. But crucially, after removing useless transitions, the Δ transitions from u no longer appear. If we choose Δ large enough to compensate for the other transitions, we get an overall reduction in size by suppressing states.

Two states connected by an edge cannot both be suppressed. Suppose $e = (u,v)$ is an edge. If S is a set where $u, v \notin S$, then the transition $v \xrightarrow{e} u$ is not suffix-compatible: the simple word ue from q^{init} to v, when extended with e gives the word uee; no suffix of uee is a simple word from q^{init} to u. We deduce that suffix-tracking sets in M_G correspond to a vertex cover in G, and vice-versa.

These two observations lead to a translation from minimum vertex cover to suffix-tracking sets with least number of states. Due to our choice of Δ, DSAs with smallest (total) size are indeed obtained from suffix-tracking sets with the least number of states. Let $k = (k'+2) \times 2\Delta + (2\Delta - 1)$.

Vertex cover $\leq k'$ implies DSA $\leq k$. Assume there is a vertex cover $\{v_1, \ldots, v_p\}$ in G with $p \leq k'$. Let S be the set of states in M_G corresponding to $\{v_1, \ldots, v_p\}$. Observe that $S \cup \{q^{init}, q^{sink}, q^{acc}\}$ is a suffix-tracking set; every transition is trivially suffix-compatible ($\forall q \xrightarrow{a} u, q \in S$ or $u \in S$). Well-formedness holds because $\forall p, q \in S, \alpha \in \mathsf{SP}(p \rightsquigarrow q, S)$ we have $|\alpha| \leq 2$; this means $\forall q' \notin S, \beta \in \mathsf{SP}(p \rightsquigarrow q', S)$, we have $\alpha \not\sqsubseteq_{\mathsf{sf}} \beta$ (since $|\beta| = 1$). Hence the derived DSA will be equivalent to M.

The derived DSA has $p+3$ states, and transitions $q \xrightarrow{1,2,\ldots,\Delta} q^{sink}$ from each except for the q^{sink} state. The transitions on q^{sink} are removable, and hence will be absent. All of this adds $(p+2) \times 2\Delta$ to the total

size (edges + label lengths). Apart from these, there are transitions with labels of length at most 2, over the alphabet $V \cup E \cup \$$. From each vertex, v, there are $|V|$ transitions to q^{sink}, one transition to q^{acc} and at most $2|E|$ transitions to other states or q^{sink}. We can choose a large enough Δ (say $(|V|+|E|)^4$), so that the size of these extra transitions is at most $2\Delta - 1$. Hence, total size is $\leq (p+2) \times 2\Delta + (2\Delta - 1)$.

By assumption, we have $p \leq k'$. Therefore, the size of the DSA is $\leq (k'+2) \times 2\Delta + (2\Delta - 1) = k$.

DSA $\leq k$ implies vertex cover $\leq k'$. Let $\mathscr{A}$ be a DSA with size $\leq k$. It may not be derived from M_G. However, by Proposition 1 we know $\mathscr{A}$ is derived from a DFA M, the tracking DFA for $\mathscr{A}$. Moreover since M_G is the minimal DFA, we know that M will be a *refinement* of M_G (see Section 2 for definition).

Let us consider a pair of states u and v from M_G, such that the vertices $u, v \in G$ have an edge between them labeled e. The DFA M will have two sets of states $u_1, u_2, \dots, u_i$ and $v_1, v_2, \dots, v_j$ that are language-equivalent to u and v respectively. Its initial state must have a transition on v to one of $v_1, v_2, \dots, v_j$. Without loss of generality, let it be to v_1. Each of $v_1, v_2, \dots, v_j$ must have a transition on e to one of $u_1, u_2, \dots, u_i$ (for equivalence with M_G) and vice-versa. Consider the run from the initial state on ve^{i+j+1}. At least one of the states among $u_1, u_2, \dots, u_i, v_1, v_2, \dots, v_j$ must be visited twice; consider the first such instance. The transition on e that re-visits a state cannot be suffix-compatible w.r.t a set S, if none of these states are in S. For it to be suffix-compatible, the string $ve^k.e$ (from initial state to the first repeated state) must have its longest simple-word suffix go the same state. Since $ve^k.e$ is not simple by itself, its longest suffix must consist entirely of e's. But on any string of e's, the initial state moves only to the sink state(s) and not to any of $u_1, u_2, \dots, u_i, v_1, v_2, \dots, v_j$. Hence any suffix-tracking set must contain at least one of these states, which maps to at least one of u or v in G. Every suffix-tracking set of M therefore maps to a vertex cover $\{v_1, v_2, \dots, v_p\}$.

Now we show that the size of this vertex cover is $\leq k'$. Each of the states picked in the suffix-tracking set will contribute to atleast 2Δ in the total size, due to the Δ transitions. We will also have these Δ transitions from the initial and accepting states. Therefore, the total size is $(p+2) \times 2\Delta + y$ for some $y > 0$. Hence $(p+2) \times 2\Delta \leq k$. This implies $p \leq k'$: otherwise we will have $p \geq k'+1$, and hence $(p+2) \times 2\Delta \geq (k'+1+2) \times 2\Delta = (k'+2) \times 2\Delta + 2\Delta > k$, a contradiction.

8 Conclusion

We have introduced the model of deterministic suffix-reading automata, compared its size with DFAs and DGAs, proposed a method to derive DSAs from DFAs, and presented the complexity of minimization. The work on DGAs [8] inspired us to look for methods to derive DSAs from DFAs, and investigate whether they lead to minimal DSAs for a language. This led to our technique of suffix-tracking sets, which derives DSAs from DFAs. The technique imposes some natural conditions on subsets of states, for them to be tracking patterns at each state. However, surprisingly, the smallest DSA that we can derive from the canonical DFA need not correspond to the minimal DSA of a language. This leads to several questions about the DSA model, and our derivation methodology.

When does the smallest DSA derived from the canonical DFA correspond to a minimal DSA? Can we use our techniques to study minimality in terms of number of states? Closure properties of DSAs - do we perform the union, intersection and complementation operations on DSAs without computing the entire equivalent DFAs? What about practical studies of using DSAs? To sum up, we believe the DSA model offers advantages in the specification of systems and in also studying regular languages from a different angle. The results that we have presented throw light on some of the different aspects in this model, and lead to many questions both from theoretical and practical perspectives.

References

[1] Ahmed Bouajjani, Peter Habermehl & Tomás Vojnar (2004): *Abstract Regular Model Checking*. In: *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, pp. 372–386, doi:10.1007/978-3-540-27813-9_29.

[2] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson & Tayssir Touili (2000): *Regular Model Checking*. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pp. 403–418, doi:10.1007/10722167_31.

[3] Janusz A. Brzozowski & Edward J. McCluskey (1963): *Signal Flow Graph Techniques for Sequential Circuit State Diagrams*. *IEEE Trans. Electron. Comput.* 12(2), pp. 67–76, doi:10.1109/PGEC.1963.263416.

[4] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled & Helmut Veith (2018): *Model checking, 2nd Edition*. MIT Press. Available at https://mitpress.mit.edu/books/model-checking-second-edition.

[5] Loris D'Antoni: *Symbolic automata*. https://pages.cs.wisc.edu/~loris/symbolicautomata.html.

[6] Loris D'Antoni & Margus Veanes (2017): *The Power of Symbolic Automata and Transducers*. In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, pp. 47–67, doi:10.1007/978-3-319-63387-9_3.

[7] Samuel Eilenberg (1974): *Automata, languages, and machines. A*. Pure and applied mathematics, Academic Press. Available at https://www.worldcat.org/oclc/310535248.

[8] Dora Giammarresi & Rosa Montalbano (1999): *Deterministic generalized automata*. *Theoretical Computer Science* 215(1-2), pp. 191–208, doi:10.1016/S0304-3975(97)00166-7.

[9] D. Giannakopoulou & K. Havelund (2001): *Automata-based verification of temporal properties on running programs*. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pp. 412–416, doi:10.1109/ASE.2001.989841.

[10] Noa Globerman & David Harel (1996): *Complexity Results for Two-Way and Multi-Pebble Automata and their Logics*. *Theor. Comput. Sci.* 169(2), pp. 161–184, doi:10.1016/S0304-3975(96)00119-3.

[11] Yo-Sub Han & Derick Wood (2004): *The Generalization of Generalized Automata: Expression Automata*. In: *Implementation and Application of Automata, 9th International Conference, CIAA 2004, Kingston, Canada, July 22-24, 2004, Revised Selected Papers*, pp. 156–166, doi:10.1007/978-3-540-30500-2_15.

[12] David Harel (1987): *Statecharts: A Visual Formalism for Complex Systems*. *Sci. Comput. Program.* 8(3), pp. 231–274, doi:10.1016/0167-6423(87)90035-9.

[13] Kosaburo Hashiguchi (1991): *Algorithms for Determining the Smallest Number of Nonterminals (States) Sufficient for Generating (Accepting) a Regular Language*. In: *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, pp. 641–648, doi:10.1007/3-540-54233-7_170.

[14] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2007): *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition, Addison-Wesley.

[15] Tao Jiang & Bala Ravikumar (1993): *Minimal NFA Problems are Hard*. *SIAM J. Comput.* 22(6), pp. 1117–1141, doi:10.1137/0222067.

[16] Richard M. Karp (1972): *Reducibility Among Combinatorial Problems*. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, pp. 85–103, doi:10.1007/978-1-4684-2001-2_9.

[17] Mehryar Mohri, Pedro J. Moreno & Eugene Weinstein (2009): *General suffix automaton construction algorithm and space bounds*. *Theor. Comput. Sci.* 410(37), pp. 3553–3562, doi:10.1016/j.tcs.2009.03.034.

[18] Margus Veanes, Pieter Hooimeijer, Benjamin Livshits, David Molnar & Nikolaj S. Bjørner (2012): *Symbolic finite state transducers: algorithms and applications*. In: *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pp. 137–150, doi:10.1145/2103656.2103674.

Adding Reconfiguration to Zielonka's Asynchronous Automata *

Mathieu Lehaut
University of Gothenburg, Gothenburg, Sweden
lehaut@chalmers.se

Nir Piterman
University of Gothenburg, Gothenburg, Sweden
piterman@chalmers.se

We study an extension of Zielonka's (fixed) asynchronous automata called reconfigurable asynchronous automata where processes can dynamically change who they communicate with. We show that reconfigurable asynchronous automata are not more expressive than fixed asynchronous automata by giving translations from one to the other. However, going from reconfigurable to fixed comes at the cost of disseminating communication (and knowledge) to all processes in the system. We then show that this is unavoidable by describing a language accepted by a reconfigurable automaton such that in every equivalent fixed automaton, every process must either be aware of all communication or be irrelevant.

1 Introduction

In recent years, computation devices have become so widely available that they are now everywhere. They are lighter, cheaper, prevalent, and, ultimately, mobile. Sensor networks, multi-agent systems, and robot teams use mobile and ad-hoc networks. In such networks, participants/agents/processes come and go and change the communication configuration based on need, location, and various restrictions. These systems force us to consider how communication changes when participants are numerous, mobile, and required to collaborate.

We consider a canonical formalism in language theory for distributed systems with a fixed communication structure – Zielonka's *asynchronous automata*. These are a well known model that supports distribution of language recognition under a fixed communication topology. In this model, a number of processes are each connected to some fixed set of channels. They can then communicate with others processes by synchronizing on a channel. During such a communication, all processes involved share their local states with each other, and then progress to a new state. Another feature of this model is that a communication can only happen if all processes involved are ready for it; if even a single process does not accept then the communication is blocked. The model is especially interesting due to Zielonka's seminal result on the ability to distribute languages in this model [9]. Zielonka's result starts from a given regular language and a target (fixed) distribution of the alphabet. He then shows that if the deterministic automaton for the language satisfies a simple condition about independence of communications then the language can be distributed and accepted by a distributed team of processes. Zielonka's quite involved construction has been revisited and optimized several times, let us cite e.g. [8, 5, 4] for the general construction and [6] for an example of a simpler construction in a restricted case. This result lead to several applications notably in synthesis [7], further establishing the usefulness of this model.

The aim of this paper is to extend the power of asynchronous automata by giving them reconfigurability. To this end, processes comprising a system are extended with the ability to dynamically connect and

*Supported by the ERC Consolidator grant D-SynMA (No. 772459).

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 88–102, doi:10.4204/EPTCS.409.10

© M. Lehaut and N. Piterman
This work is licensed under the
Creative Commons Attribution License.

disconnect from channels after a communication. As before, a communication can only occur on a channel if all the processes that are connected to the channel agree on it, and otherwise the communication is blocked. This is the exact notion of communication of asynchronous automata, except that processes can now connect and disconnect to channels dynamically during an execution. In order to allow more than mere synchronization on the channel, communications are extended by a data value, which correspond to the state sharing of asynchronous automata. We call this variant *reconfigurable asynchronous automata*. They are inspired by attribute-based communication calculus [2, 1] and channeled transition systems [3], though they are much simpler than those and adapted to the context of asynchronous automata. To prevent confusions, we sometimes refer to the base variant as *fixed* (as opposed to reconfigurable) asynchronous automata.

With the definition of this new extension, the first natural question is whether reconfigurable asynchronous automata are more expressive than the fixed variant. To this we answer negatively by showing how to translate from one model to the other. Going from fixed to reconfigurable is easy. We also show that if the fixed asynchronous automaton has local transitions, i.e. the next state of a process only depends on its own current state and not on the state of others, then it corresponds to a reconfigurable asynchronous automaton that does not use data values during communications. The other direction is also relatively easy, however with an important caveat: every process in the fixed automaton participates in every communication, autonomously deciding which communications to ignore and which to act upon.

With the two models being equi-expressive, the second natural question is what does adding reconfigurability actually bring? It is well known that non-deterministic finite automata are as expressive as deterministic ones, but can be exponentially smaller in size. In the case of fixed versus reconfigurable asynchronous automata, the gain is not in size, but in the communication structure. As explained just before, our translation from reconfigurable to fixed asynchronous automata results in essentially sharing all information to all parts of the system, and letting each process decide whether that information is actually needed. This is undesirable for many reasons. First, in real systems, each communication takes time and costs energy to process, so one should not waste resources sending information that would be useless to some process. Second, for privacy reasons, it is obviously not desirable that every process in the system has access to every communication; we would rather that a process only receives information based on its "need-to-know". Thirdly, it implies that every process is always connected to every other process in the system, which is not good in systems with a high number of participants that only require a small number of connections at each time.

We then show that this sharing is, in general, unavoidable. We suggest a language that can be recognized by reconfigurable asynchronous automata but for which any equivalent fixed asynchronous automaton has the pitfall described above. In this language, using reconfigurable communication, processes actively connect and disconnect from channels and keep themselves informed only about crucial information. Throughout, processes are connected to a very small number of channels that is independent of system size. However, some (changing) channels are used for coordination of how to connect and disconnect from the other channels. We show that for asynchronous automata to recognize the same language, some processes must be connected to the full set of channels and be informed of everything. What's more, every process that is not connected to the full set of channels can be made trivial by accepting unconditionally all possible communication on channels that they are connected to. We also show that the system contains a subsystem performing the same computation in which the processes that are not fully connected are completely trivial: the system does not need them at all to perform exactly the same computation.

The rest of the paper is organized as follows. In Section 2 we recall the definition of Zielonka's

asynchronous automata and give the definition of reconfigurable asynchronous automata. In Section 3 we give the translations between the models and show that the data of reconfigurable asynchronous automata correspond to the global transitions of fixed asynchronous automata. We then show in Section 4 that in every translation that removes the reconfigurability, all processes either know everything or are trivial. Finally, we conclude and discuss our results in Section 5.

2 Definitions

2.1 Fixed Communication Structure

2.1.1 Distributed Alphabets

We fix a finite set $\mathbb{P}$ of processes. Let Σ be a finite alphabet, and $dom : \Sigma \to 2^{\mathbb{P}}$ a domain function associating each letter with the subset of processes listening to that letter. The pair (Σ, dom) is called a distributed alphabet. We let $dom^{-1}(p) = \{a \in \Sigma \mid p \in dom(a)\}$. It induces an independence binary relation I in the following way: $(a,b) \in I \Leftrightarrow dom(a) \cap dom(b) = \emptyset$. Two words $u = u_1 \dots u_n$ and $v = v_1 \dots v_n$ are said to be equivalent, denoted by $u \sim v$, if one can start from u, repeatedly switch two consecutive independent letters, and end up with v. Let us denote by $[u]$ the equivalence class of a word u. Let $\mathscr{A} = (Q, \Sigma, q_0, \Delta, F)$ be a deterministic automaton over Σ. We say that $\mathscr{A}$ is I-diamond if for all pairs of independent letters $(a,b) \in I$ and all states $q \in Q$, we have $\Delta(q,ab) = \Delta(q,ba)$. If $\mathscr{A}$ has this property, then a word u is accepted by $\mathscr{A}$ if and only if all words in $[u]$ are accepted. Zielonka's result states that an I-diamond automaton can be distributed to processes who are connected to channels according to dom [9].

2.1.2 Asynchronous Automata

An *asynchronous automaton* (in short: AA) [9] over distributed alphabet (Σ, dom) and processes $\mathbb{P}$ is a tuple

$$\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma}, \mathrm{Acc})$$

such that:

- S_p is the finite set of states for process p, and $s_p^0 \in S_p$ is its initial state,
- $\delta_a : \prod_{p\in dom(a)} S_p \to \prod_{p\in dom(a)} S_p$ is a partial transition function for letter a that only depends on the states of processes in $dom(a)$ and leaves those outside unchanged,
- $\mathrm{Acc} \subseteq \prod_{p\in\mathbb{P}} S_p$ is a set of accepting states.

A global state of the automaton is of the form $\mathbf{s} = (s_p)_{p\in\mathbb{P}}$, giving the state of each process. For every such global state and every subset $P \subseteq \mathbb{P}$, we denote by $\mathbf{s}\downarrow_P = (s_p)_{p\in P}$ the subset of $\mathbf{s}$ of states from processes in P. Then a run of $\mathscr{B}$ is a sequence $\mathbf{s}_0 a_1 \mathbf{s}_1 a_2 \dots \mathbf{s}_n$ where for all $0 < i \leq n$, $\mathbf{s}_i \in \prod_{p\in\mathbb{P}} S_p$, $a_i \in \Sigma$, satisfying $\mathbf{s}_0 = (s_p^0)_{p\in\mathbb{P}}$ and the following relation:

$$\mathbf{s}_i \downarrow_{dom(a_i)} = \delta_{a_i}(\mathbf{s}_{i-1} \downarrow_{dom(a_i)}) \text{ and } \mathbf{s}_i \downarrow_{\mathbb{P}\setminus dom(a_i)} = \mathbf{s}_{i-1} \downarrow_{\mathbb{P}\setminus dom(a_i)}$$

A run is accepting if $\mathbf{s}_n$ belongs to Acc. The word $a_1 a_2 \dots$ is accepted by $\mathscr{B}$ if such an accepting run exists (note that automata are deterministic but runs on certain words may not exist). The language of $\mathscr{B}$, denoted by $\mathscr{L}(\mathscr{B})$, is the set of words accepted by $\mathscr{B}$. For the rest of this paper, we will drop the Acc component as we focus more on the runs themselves over whether they can reach a certain target. That is, we assume that $\mathrm{Acc} = \prod_{p\in\mathbb{P}} S_p$. This restricts the languages that can be recognized by asynchronous automata but still allows us to prove all our results.

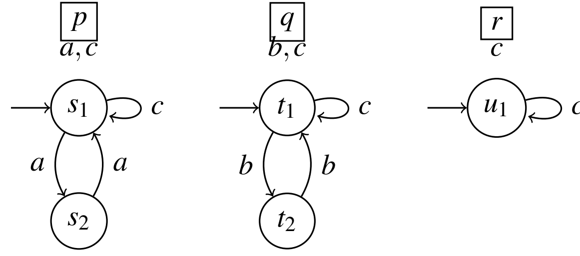


Figure 1: An asynchronous automaton $\mathscr{B}$ over three processes.

Example 1. We give an example of an asynchronous automaton $\mathscr{B}$ in Figure 1. There are three letters $\Sigma = \{a,b,c\}$ distributed over three processes $\mathbb{P} = \{p,q,r\}$ with the domain: $dom(a) = \{p\}, dom(b) = \{q\}, dom(c) = \mathbb{P}$. An example of a run is the following sequence:

$$(s_1,t_1,u_1)\ a\ (s_2,t_1,u_1)\ b\ (s_2,t_2,u_1)\ b\ (s_2,t_1,u_1)\ a\ (s_1,t_1,u_1)\ c\ (s_1,t_1,u_1)$$

which gives *abbac* as a word in $\mathscr{L}(\mathscr{B})$. More generally, $\mathscr{B}$ accepts all words where all occurrences of c are such that there are an even number of a's and an even number of b's in the prefix before the c occurrence. That is,

$$\mathscr{L}(\mathscr{B}) = \left\{ v_0 \dots v_n \in \{a,b,c\}^* \;\middle|\; \begin{array}{r} \forall i \,.\, v_i = c \text{ implies } a_\sharp(v_0 \dots v_i) =_{mod2} 0 \\ \text{and } b_\sharp(v_0 \dots v_i) =_{mod2} 0 \end{array} \right\},$$

where $\sigma_\sharp(w)$ is the number of occurrences of letter σ in word w.

2.1.3 Local Asynchronous Automata

We also define a weaker version of asynchronous automata, called *local* asynchronous automata (short: LAA or local AA), in which the transition function is local to each process, and therefore independent with respect to the states of all other processes. To avoid confusion, we sometimes refer to normal asynchronous automata as defined earlier as *global* asynchronous automata (or global AA), though by default AA refers to global AA.

A *local asynchronous automaton* over (Σ, dom) and $\mathbb{P}$ is a tuple

$$\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_p)_{p\in\mathbb{P}}),$$

where S_p and s_p^0 are defined as before, and $\delta_p : S_p \times dom^{-1}(p) \to S_p$ is the transition function of process p. A run of $\mathscr{B}$ is a sequence $\mathbf{s}_0 a_1 \mathbf{s}_1 a_2 \dots \mathbf{s}_n$ where $\mathbf{s}_0 = (s_p^0)_{p\in\mathbb{P}}$ and for all $0 < i \leq n$, $\mathbf{s}_i = (s_i^p)_{p\in\mathbb{P}} \in \prod_{p\in\mathbb{P}} S_p$, $a_i \in \Sigma$, satisfying the following relation:

$$s_i^p = \begin{cases} \delta_p(s_{i-1}^p, a_i) & \text{if } p \in dom(a_i), \\ s_{i-1}^p & \text{otherwise.} \end{cases}$$

Observe that a local AA is a syntactic restriction of global AA. There are languages recognizable by global AA that cannot be recognized by local AA, because intuitively it would be impossible to make a process react differently to the same communication based on differences observed by another process. For example, take $\Sigma = \{a, \bar{a}, b, c, \bar{c}\}$ and two processes p,q such that p listens to $a, \bar{a}, b$ and q listens to

$c, \bar{c}, b$. Then take language $L = \{abc, \bar{a}b\bar{c}\}$. One can easily see that L can be recognized by a global AA but by no local AA.

In particular, Zielonka's distribution result [9] no longer holds for local AA. Note that the automaton given in Figure 1 is local.

2.2 Reconfigurable Communication

Let us consider here a model where the communication structure is not fixed, and can be modified dynamically during a run. As before, let us fix a finite set $\mathbb{P}$ of processes. Let us as well fix a finite set C of channels, with a role similar to the alphabet Σ of the previous section. Here, the function *dom* is replaced by a state-dependent listening function through which processes reconfigure their communication interfaces depending on their current state. Finally, let T be a finite set of message contents. The intuition behind T is to abstract the state-sharing part of a communication to allow us to define each process' transition function independently of other processes. We emphasize that this has nothing to do with reconfigurability and is just a way to have nicer definitions.

2.2.1 Reconfigurable Asynchronous Automata

A *reconfigurable asynchronous automaton* (in short: RAA) over C is a tuple $\mathscr{A} = (S, s^0, \Delta, L)$ where:

- S is a set of states, $s^0 \in S$ being the initial state,
- $\Delta : S \times (T \times C) \to S$ is the partial transition function, where $\Delta(s,(t,c)) = s'$ means going from state s to s' after having a message on channel c with content t. We write $(s,(t,c),s') \in \Delta$ for $\Delta(s,(t,c)) = s'$,
- $L : S \to 2^C$ is a listening function such that $c \in L(s)$ if there is a transition of the form $(s,(t,c),s') \in \Delta$, i.e. state s must be listening to channel c if there is some transition from s involving a message on c.

A run of $\mathscr{A}$ is a sequence $s_0 m_1 s_1 m_2 \dots s_n$ starting from the initial state $s_0 = s^0$ and where for all $0 < i \leq n, m_i \in T \times C$ and $\Delta(s_{i-1}, m_i) = s_i$. The language of $\mathscr{A}$, denoted by $\mathscr{L}(\mathscr{A})$, is the set of words over C of the form $c_0 c_1 \dots$ such that there exists a run of the form $s_0(t_0,c_0)s_1(t_1,c_1)\dots$, i.e. we focus only on the sequence of channels where messages are sent, and drop the states and message contents.

Intuitively this definition represents the behavior of a single process, communicating with the outside on channels from C. In order to be able to reconstruct the whole system, we now define the parallel composition of RAA.

Given a sequence of RAA $(\mathscr{A}_p)_{p \in \mathbb{P}}$ with $\mathscr{A}_p = (S_p, s^0_p, \Delta_p, L_p)$, one can define their parallel composition $\mathscr{A}_{\|\mathbb{P}} = (S, s^0, \Delta, L)$:

- $S = \prod_{p \in \mathbb{P}} S_p$ and $s_0 = (s^0_p)_{p \in \mathbb{P}}$,
- $L((s_p)_{p \in \mathbb{P}}) = \bigcup_{p \in \mathbb{P}} L_p(s_p)$,
- $\Delta((s_p)_{p \in \mathbb{P}}, (t,c)) = (s'_p)_{p \in \mathbb{P}}$ if the following conditions are met:
    1. $\exists p$ s.t. $c \in L_p(s_p)$,
    2. $\forall p$ s.t. $c \in L_p(s_p), (s_p,(t,c),s'_p) \in \Delta_p$, and
    3. $\forall p$ s.t. $c \notin L_p(s_p), s'_p = s_p$.

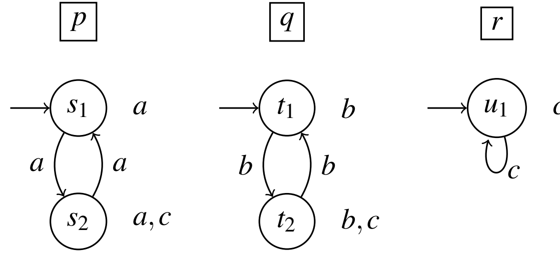


Figure 2: An RAA $\mathscr{A}$ over three processes. The listening function is given to the right of each state.

In plain words, there is a transition if all processes listening to the corresponding channel have a transition with the *same* message content, with at least one process listening to the channel, whereas those that do not listen are left unchanged. Note that if some process listens to that channel but does not implement the transition, then that transition is blocked.

By convention, an RAA over C and $\mathbb{P}$ refers to an RAA of the form $\mathscr{A}_{\|\mathbb{P}}$ as described above.

Example 2. Figure 2 shows an example of RAA over channels $C = \{a,b,c\}$ and three processes $\mathbb{P} = \{p,q,r\}$. Here we take $T = \{t\}$ as the set of message contents, so for readability purposes it is omitted from the transitions. Note that when process p is in state s_2, it is listening to channel c but no c-transition is implemented, therefore a communication on c is impossible (similarly for q and t_2). So the only way a communication happens on c is when p and q are in s_1 and t_1 respectively, which means only process r listens to c. It is then easy to see that this RAA accepts the same language as the AA given in Figure 1. Note that it does so without p or q ever taking part in a communication on c, contrary to the previous example.

3 From Fixed to Reconfigurable and Back

We now focus on comparing the expressive power of these two formalisms. For the rest of this section, we fix a finite set $\mathbb{P}$ of processes.

3.1 Fixed AA to Reconfigurable AA

Let (Σ, dom) be a distributed alphabet, and let $\mathscr{B}$ be an AA over it. One can construct an RAA $\mathscr{A}_{\|\mathbb{P}}$ with Σ as set of channels that recognizes the same language as $\mathscr{B}$.

The intuition is as follows. The listening function of each process is the same for all states: each process always listens to the channels that have this process in their domain. The only part that is not straightforward to emulate is that a transition of an AA depends on the states of all processes in the domain of the corresponding letter. Therefore each process in the RAA needs to share their states via message contents to all others when emulating a transition.

Theorem 1. *Every language recognized by an AA over (Σ, dom) and $\mathbb{P}$ can be recognized by an RAA with set of channels Σ and processes $\mathbb{P}$.*

Proof. Let $\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$ be an AA as described earlier. For the set of messages, we take $T = \bigcup_{a\in\Sigma}(\prod_{p\in dom(a)} S_p)$.

Then let $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$ be a RAA for process p where:

- $L_p(s) = \{a \in \Sigma \mid p \in dom(a)\}$ for all $s \in S_p$,
- $\Delta_p(s_p,(t,a)) = (\delta_a(t))\downarrow_{\{p\}}$ if $s_p = t\downarrow_{\{p\}}$

i.e. an a-transition is possible if and only if the message t is the tuple comprising the current states of all processes in $dom(a)$, and all processes then update their state according to δ_a.

By construction, one can show inductively that for each run of $\mathscr{B}$, there is a corresponding run of $\mathscr{A}_{\|\mathbb{P}}$ where at each point, the state of each process p is the same in both runs. It follows that $\mathscr{L}(\mathscr{B}) \subseteq \mathscr{L}(\mathscr{A}_{\|\mathbb{P}})$ and conversely $\mathscr{A}_{\|\mathbb{P}}$ can only emulate runs of $\mathscr{B}$, showing the reverse inclusion. □

Note that the size of the constructed RAA lies almost entirely in the size of T, the message contents set, which is $\prod_{p\in\mathbb{P}} S_p$.

For local AA the translation is even more straightforward, as no message content is required (i.e. T can be reduced to a singleton).

Corollary 2. *Every language recognized by an LAA over (Σ, dom) can be recognized by an RAA with set of channels Σ and where $|T| = 1$.*

Proof. In the case of LAA, the transition δ_p does not depend on the states of other processes. Let $T = \{t\}$. We replace the transition Δ_p in the proof of Theorem 1 by $\Delta_p(s_p,(t,a)) = \delta_p(s_p,a)$. □

3.2 Reconfigurable to Fixed

Let us now focus on the reverse direction. Let $(\mathscr{A}_p)_{p\in\mathbb{P}}$ be a sequence of RAA over $\mathbb{P}$ with set of channels C, and let $\mathscr{A}$ be their parallel composition. Our goal is to create an AA with alphabet C that recognizes the same language. The question that arises is: what should dom be defined as for the distributed alphabet (C, dom)?

The solution is to define it as the complete domain function $Fdom$: $Fdom(a) = \mathbb{P}$ for all channels. In that case, it is simple to build an AA over $(C, Fdom)$ that emulates $\mathscr{A}$, as each process can simply stutter when they are not supposed to listen to a channel.

Theorem 3. *Every language recognized by an RAA over set of channels C and processes $\mathbb{P}$ can be recognized by an AA over $(C, Fdom)$ and the same set of processes.*

Proof. Consider $(\mathscr{A}_p)_{p\in\mathbb{P}}$, where $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$, with $\mathscr{A} = (S, s^0, \Delta, L)$ being their parallel composition over message contents T. We build $\mathscr{B} = ((Q_p)_{p\in\mathbb{P}}, (q_p^0)_{p\in\mathbb{P}}, (\delta_c)_{c\in C})$ as follows:

- for all $p \in \mathbb{P}$, $Q_p = S_p$, and $q_p^0 = s_p^0$
- For channel $c \in C$ we have δ_c defined as follows.

$$\delta_c = \left\{ ((q_p)_{p\in\mathbb{P}}, (q'_p)_{p\in\mathbb{P}}) \;\middle|\; \begin{array}{l} \exists p \in \mathbb{P}. c \in L_p(q_p) \text{ and} \\ \exists t \in T. \forall p \in \mathbb{P}. \\ \quad \text{if } c \in L_p(q_p), (q_{p'}, (t,c), q'_{p'}) \in \Delta_{p'} \text{ and} \\ \quad \text{if } c \notin L_p(q_p), q_p = q'_p \end{array} \right\}$$

Similarly to Theorem 1, the construction makes it so that any run from $\mathscr{A}$ has a corresponding run in $\mathscr{B}$ where the state are identical for each process, and the same in the other direction. □

Note that having global transitions is necessary to ensure all processes share the same message content t. However if we assume that T is a singleton, then local transitions suffice. Additionally, notice that the construction would still work with a set T of infinite size, so we could consider RAA where processes synchronize by agreeing on, say, an integer.

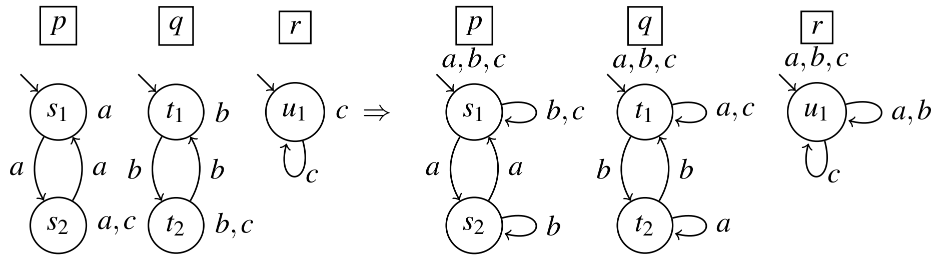


Figure 3: On the left, the RAA from Example 2. On the right, its translation to an AA.

Corollary 4. *Every language recognized by an RAA over C and $\mathbb{P}$, where $|T| = 1$, can be recognized by an LAA over $(C, Fdom)$ and $\mathbb{P}$.*

We illustrate this construction in Figure 3. Note that for this particular example the general construction is not optimal. For example, process p is made to listen to b but can never block a communication on this channel with all states having a self-loop on reading b. Thus, one could safely remove the letter b from the alphabet of p. By doing similarly on other processes, one can get back the AA from Example 1.

There is an alternative construction that does not require *all* processes to listen to *all* channels. If one process does while also storing the state information of every other processes, then it can simulate the original automaton by itself; meanwhile every other process can listen to an arbitrary set of channels as long as they accept every communication. In other words, one process serves as a centralized executor of the simulation, while others simply need to be non-blocking. With a centralized executor there is no point in having computation done anywhere but in the centralized executor. By abuse of definition we still refer to such a domain function as a complete domain.

In the next section we show that there is no hope of finding a transformation that does not require a complete domain.

4 Trivializable, Fully Listening, and Trivial

The method described above is a general method to transform a reconfigurable asynchronous automaton into an equivalent fixed asynchronous automaton, with the cost of needing a complete domain function. It is of course possible that for some particular examples such a heavy construction is not needed, and a translation with a much smaller domain could be possible. However, we show that there is no better (in terms of channel domain) *general* translation by giving an example of an RAA such that every equivalent AA relies on a complete domain.

The idea is to allow every possible subset of channels to be either fully independent, that is every one of those channels can be used in parallel, or make them sequentially dependent, that is they can only be used in a certain order. This status can be switched by a communication on a separate channel (that all processes listen to), called the switching channel. Moreover, after enough switches, a different channel will serve as the switching channel. That way, all channels have the opportunity to serve as the switching channel, given enough switches. Our construction does not use data values during communications. Thus, already the weakest form of RAA is enough for this example.

4.1 Description of the switching RAA

Let $\mathbb{P} = \{p_1, \dots, p_n\}$. We fix $C = \{c_1, \dots, c_n, c_{n+1}\}$, that is, we have one channel per process and one additional channel to be used as switching channel (dynamically).

For all $\mathrm{sc} \in C$ (sc stands for *switching channel*), fix $<_{\mathrm{sc}}$ an arbitrary total order over $2^{C\setminus\{\mathrm{sc}\}}$, with the only requirement that $\emptyset$ be the minimal element. Intuitively, a set in $2^{C\setminus\{\mathrm{sc}\}}$ will represent the set of dependent channels, and a switch will go to the next one with respect to $<_{\mathrm{sc}}$. Let us denote by $inc_{<_{\mathrm{sc}}} : 2^{C\setminus\{\mathrm{sc}\}} \to 2^{C\setminus\{\mathrm{sc}\}} \cup \{\bot\}$ the function that returns the next set according to $<_{\mathrm{sc}}$ or $\bot$ for the maximal element.

Additionally, for every subset $D \subseteq C$, we fix $\overset{D}{+}1 : C \to C$ a function that cycles through all elements of D and is the identity on $C \setminus D$. For convenience we write $d \overset{D}{+} 1$ for $\overset{D}{+}1(d)$. We also define $\overset{D}{-}1 : D \to D$ the inverse function and use the same notation. Namely, for every $d \in D$ we have $(d \overset{D}{-} 1) \overset{D}{+} 1 = d$ and $(d \overset{D}{+} 1) \overset{D}{-} 1 = d$. We denote by $c_D \in D$ an arbitrary element of D.

Finally, we set $T = \{t\}$, and omit the message content component in transitions.

We build $\mathscr{A}_p = (S_p, s_p^0, \Delta_p, L_p)$ for $p = p_k$ as follows:

- $S_p = \{(c, \mathrm{sc}, D, d) \mid c, \mathrm{sc} \in C, D \subseteq C \setminus \{\mathrm{sc}\}, d \in D \cup \{c\}\}$, and $s_p^0 = (c_k, c_{n+1}, \emptyset, c_k)$.

  The first component is the channel assigned to this process, initially c_k for process p_k, but may change if c_k becomes the switching channel. The second component is the current switching channel, initialized to c_{n+1} for all processes. Component D represents the set of channels that are currently dependent, and d is the next channel that $\mathscr{A}_k$ is listening to on which it is expecting communication.

- All processes listen to the switching channel and their assigned channel, plus the previous one if D contains the assigned channel:

$$L_p(c, \mathrm{sc}, D, d) = \begin{cases} \{\mathrm{sc}, c, c \overset{D}{-} 1\} & \text{if } c \in D \\ \{\mathrm{sc}, c\} & \text{if } c \notin D \end{cases}$$

- The transition Δ_p is the union of the following sets .

$$\{((c, \mathrm{sc}, D, c), c, (c, \mathrm{sc}, D, c \overset{D}{-} 1))\} \tag{1}$$

$$\{((c, \mathrm{sc}, D, c \overset{D}{-} 1), c \overset{D}{-} 1, (c, \mathrm{sc}, D, c)\} \tag{2}$$

  The first two kinds of transitions handle the independence of all channels in $C \setminus D$ and the cycling through the channels of D. If $c \notin D$ then $c = c \overset{D}{-} 1$. In this case, the first two sets simply say that a transition on c is always possible. If $c \in D$, then the process awaits until it gets a message on $c \overset{D}{-} 1$ and then is ready to interact on c. After interaction on c it awaits another interaction on $c \overset{D}{-} 1$. It follows that all the processes owning the channels in D enforce together the cyclic order on the messages in D. This part is further illustrated in Figure 4.

  Remaining transitions describe what happens when a switch occurs.

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}, D', c)) \mid D' = inc_{<_{\mathrm{sc}}}(D) \neq \bot \text{ and } c = c_{D'}\} \tag{3}$$

$$\{((c, \mathrm{sc}, D, d), \mathrm{sc}, (c, \mathrm{sc}, D', c \overset{D'}{-} 1)) \mid D' = inc_{<_{\mathrm{sc}}}(D) \neq \bot \text{ and } c \neq c_{D'}\} \tag{4}$$

  Sets three and four describe what happens when the next set according to $<_{\mathrm{sc}}$ is defined. In this case, the next set becomes the new set of dependent channels D. Set three handles the case of the process that is in charge of the channel becoming the first channel to communicate on the new set

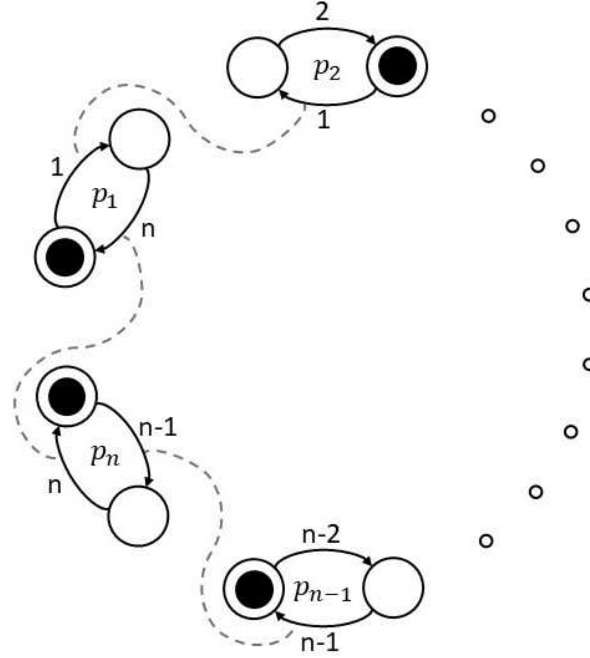


Figure 4: Illustration of how the order on the channels in D is maintained. We consider the case where $D = \{1,\dots,n\}$ and p_i is in charge of channel i. The order between the channels is the natural order on $\{1,\dots,n\}$. The black token indicates the current state for each process. Transitions that are on the same channel are connected with a dashed line. The system is set up for next communication on channel 1 and all other channels are blocked. Indeed, both processess listening to channel 1 are ready to interact on 1 (p_1 in state $(1,n+1,D,1)$ and p_2 in state $(2,n+1,D,1)$) and for every other channel $i > 2$ process i is awaiting communication on $i-1$ (p_i in state $(i,n+1,D,i-1)$) so channel i is not enabled.

$inc_{<_{\mathrm{sc}}}(D)$. This process is ready for communication on this first channel. The fourth set handles the case of all other processes. All other processes are either in charge of channels in D', in which case they set themselves to await a communication on the previous in D' or they are in charge of channels not in D' in which case, c and $c -^{D'} 1 = c$, and the process is ready to communicate on c.

$$\{((c,\mathrm{sc},D,d),\mathrm{sc},(c,\mathrm{sc}+^{C}1,\emptyset,c)) \mid inc_{<_{\mathrm{sc}}}(D) = \perp \text{ and } c \neq \mathrm{sc}+^{C}1\} \tag{5}$$

$$\{((c,\mathrm{sc},D,d),\mathrm{sc},(\mathrm{sc},\mathrm{sc}+^{C}1,\emptyset,\mathrm{sc})) \mid inc_{<_{\mathrm{sc}}}(D) = \perp \text{ and } c = \mathrm{sc}+^{C}1\} \tag{6}$$

Finally, sets five and six describe what happens when the next set according to $<_{\mathrm{sc}}$ is undefined. In this case, the next dependent set becomes $\emptyset$. Most processes just set the dependent set to $\emptyset$ and allow communication on "their" channel (set 5). The process that was in charge of the new switching channel $\mathrm{sc}+^{C}1$ takes over the old switching channel sc and is ready to communicate on it (set 6). Notice that communications on the switching channel affect all processes. The change in D and the change of the switching channel is further illustrated in Figure 5.

An illustration of the whole construction for $n = 2$ (i.e. 2 processes and 3 channels) is given in Figure 6. There we have processes $\mathbb{P} = \{p,q\}$ and channels $C = \{1,2,3\}$. Initially p is assigned channel 1 and q channel 2, while channel 3 is the switching channel. We chose as order for the dependent sets the following order: $\emptyset <_3 \{1\} <_3 \{2\} <_3 \{1,2\}$. The blue states illustrate the moment when the dependent

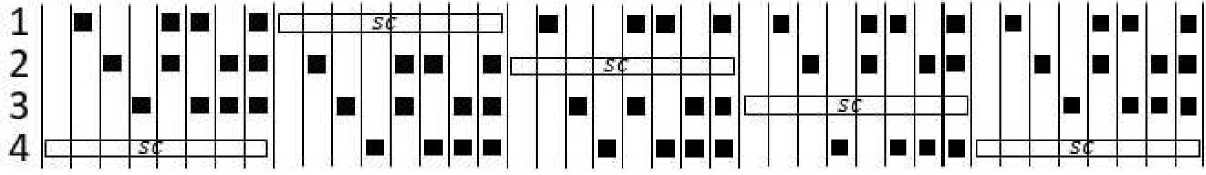


Figure 5: Illustration of how the set D and the switching channel sc change whenever there is a communication on sc. We consider the case where there are three processes and four channels. Each column corresponds to the status after one more communication on sc. Each channel is in turn the switching channel starting with 4. The channels in D at a certain time/column are marked with a black box. We cycle through the states in $\{1,\dots,4\} \setminus \{\text{sc}\}$ according to set size first and then lexigographically on the sorted set.

set has two channels that must be used in the correct order ($1 \to 2 \to 1 \to \dots$). The red transitions lead to a change in the switching channel. At that point, 1 becomes the new switching channel. Thus p gets a new assigned channel (3, i.e. the previous switching channel), while q keeps its old assigned channel. After enough changes of the switching channel, the state cycles back to the initial state for both.

Let $\mathscr{A}$ be the parallel composition of $(\mathscr{A}_p)$. A state for one process keeps track of 3 channels and one set of channels, so its size is in $O(n^3.2^n)$. Therefore, the size of the state space of $\mathscr{A}$ is in $O((n^3.2^n)^n)$.

4.2 Asynchronous Automata Construction

We show that an AA that recognizes the same language as $\mathscr{A}$ has the following property: for each process p, either p listens to every channel ($dom^{-1}(p) = C$), or from every reachable state there is a path to a bottom strongly connected component that is complete w.r.t. $dom^{-1}(p)$. That is, for every state s in this bottom SCC and for every channel in $dom^{-1}(p)$ the transition $\delta(s,c)$ is defined. In the former case, we call p *fully-listening*. In the latter case, we say that p is *trivializable*, as once it is in this bottom SCC it always includes transitions for all the channels it listens to. Thus, p becomes irrelevant to the rest of the computation.

Theorem 5. *Let $\mathscr{B}$ be an AA such that $\mathscr{L}(\mathscr{B}) = \mathscr{L}(\mathscr{A})$. Each process in $\mathscr{B}$ is either fully-listening or trivializable.*

Proof. Let $\mathscr{B} = ((S_p)_{p\in\mathbb{P}}, (s_p^0)_{p\in\mathbb{P}}, (\delta_a)_{a\in\Sigma})$, and let $p \in \mathbb{P}$. Assume that p is not fully-listening, so let $C_p = dom^{-1}(p) \subsetneq C$. In particular, let $c \in C \setminus C_p$ be a channel that p does not listen to.

Let s_p be a reachable state for p in $\mathscr{B}$. Then there is w a computation in $\mathscr{L}(\mathscr{B})$ such that p reaches state s_p after w. Consider the same computation in $\mathscr{A}$, and let sc be the current switching channel in $\mathscr{A}$ at the end of w. Let $\text{sc} \cdot c_1 \cdot \dots \cdot c_{n-1} \cdot c$ be the sequence of channels from sc to c according to the order from +C1. Then there is a continuation w' of w of the form $\text{sc}^{k_0} \cdot c_1^{k_1} \cdot \dots \cdot c_{n-1}^{k_{n-1}}$ with $k_0, \dots, k_{n-1} \in \mathbb{N}$ such that:

- $ww' \in \mathscr{L}(\mathscr{A})$,
- after w', c is the current switching channel and the dependent set D is $\emptyset$.

From this, every continuation $w'' \in (C \setminus \{c\})^*$ is still in $\mathscr{L}(\mathscr{A})$ and does not change the switching channel or the dependent set. In particular, every $w'' \in (dom^{-1}(p))^*$ maintains that $w \cdot w' \cdot w''$ is also in $\mathscr{L}(\mathscr{A}) = \mathscr{L}(\mathscr{B})$. Therefore, from the state reached in $\mathscr{B}$ after ww', there is a path to a strongly connected component that will implement all transitions in $dom^{-1}(p)$, i.e. a complete one. $\square$

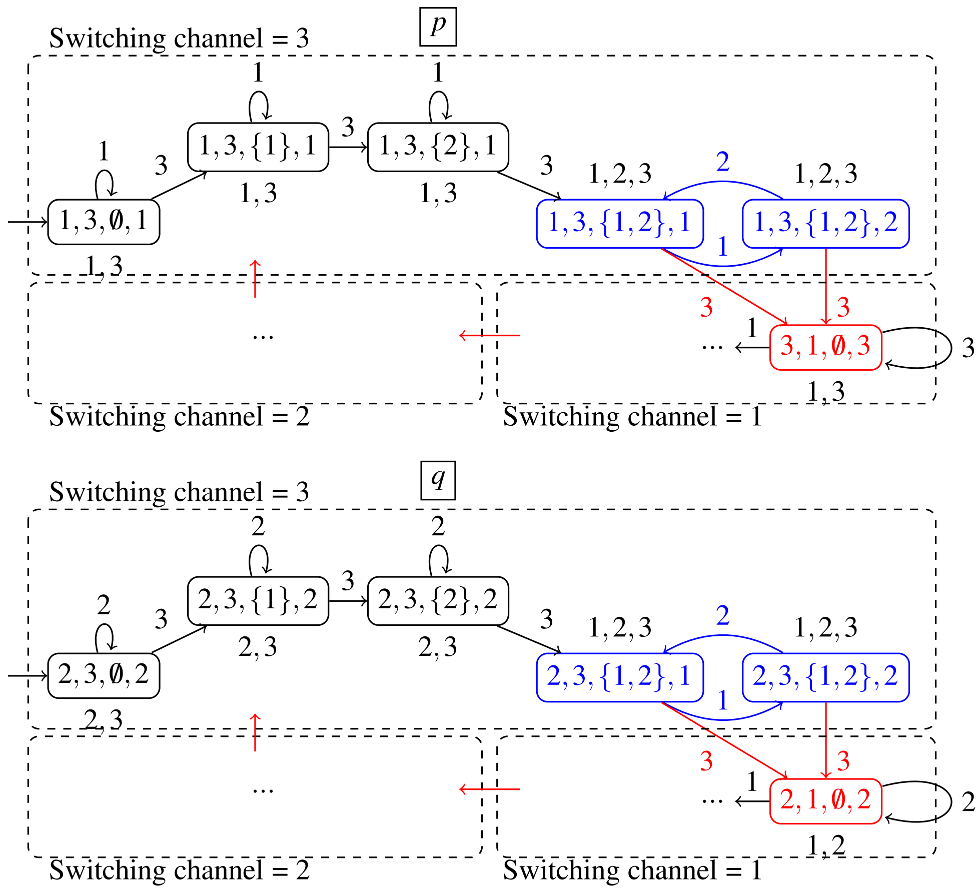


Figure 6: Illustration of the switching RAA for $n = 2$.

A process that is trivializable may become irrelevant. This means that there are pathological runs where only fully listening processes are active in the computation while others passively accept everything. However, trivializable processes *may* still initially participate in the computation. Nevertheless, for the languages given in this section, we show that there exists an alternative initial configuration of the system where trivializable processes actually start trivialized. This means that, in essence, all the machinery required for doing the entire computation is present within the remaining fully listening processes only.

Given a language $\mathscr{L}$ and a word w let $w \backslash \mathscr{L} = \{w' \mid ww' \in \mathscr{L}\}$ and let $\mathit{pref}(\mathscr{L}) = \{w \mid \exists w' \,.\, ww' \in \mathscr{L}\}$. A language $\mathscr{L}$ is *repetitive* if for every word $w \in \mathit{pref}(\mathscr{L})$ there exists a word w' such that $ww' \backslash \mathscr{L} = \mathscr{L}$.

Lemma 6. *The language $\mathscr{L}(\mathscr{A})$ is repetitive.*

Proof. Consider a word w and the configuration of $\mathscr{A}$ reachable after reading w. All processes in $\mathscr{A}$ agree on the set D and the channel sc. Every communication on sc increases the set of dependent channels in the order $<_{\mathrm{sc}}$ until reaching the set D' such that $\mathit{inc}_{<_{\mathrm{sc}}}(D') = \perp$. An additional communication on sc then leads to the switching channel being updated to $\mathrm{sc} \overset{C}{+} 1$.

So after at most 2^n communications on sc the switching channel becomes $\mathrm{sc} \overset{C}{+} 1$. Let w_0 be the word that leads to the switching channel changing.

For every channel, c_i, when the dependent set is $\emptyset$ the sequence $(c_i)^{2^n}$ leads to the change of the switching channel from c_i to $c_i \overset{C}{+} 1$.

Let $\mathrm{sc} = c^0, c^1, \ldots, c^k$ be the sequence of switching channels ending $c^k \overset{C}{+} 1 = c_{n+1}$.

It follows that $w_0 \cdot (c^1)^{2^n} \cdots (c^k)^{2^n}$ leads $\mathscr{A}$ to setting the switching channel to c_{n+1}. At that point all processes in $\mathscr{A}$ are in their initial states except for their assigned channel, which is shifted by one. Namely, process p_k ends up in state $(c_k \overset{(C \setminus \{c_{n-1}\})}{-} 1, c_{n+1}, \emptyset, c_k)$.

Now let $w_{\mathrm{loop}} = c_{n+1}^{2^n} \cdot c_1^{2^n} \cdots c_n^{2^n}$. Each application of w_{loop} again shifts assigned channels by one. So after $n-1$ applications, each process finishes in its initial state. From this configuration the residue language is $\mathscr{L}(\mathscr{A})$. □

Using repetitiveness we can strengthen our result as follows. A process is *trivial* if its initial state lies in a bottom strongly connected component that is complete w.r.t. $\mathit{dom}^{-1}(p)$. Given an AA $\mathscr{B} = ((S_p)_{p \in \mathbb{P}}, (s_p^0)_{p \in \mathbb{P}}, (\delta_a)_{a \in \Sigma})$ and an alternative initial configuration $\vec{t} = (t_p^0)_{p \in \mathbb{P}}$ we denote by $\mathscr{B}(\vec{t})$ the AA $\mathscr{B}(\vec{t}) = ((S_p)_{p \in \mathbb{P}}, (t_p^0)_{p \in \mathbb{P}}, (\delta_a)_{a \in \Sigma})$.

Theorem 7. *Let $\mathscr{B}$ be an AA such that $\mathscr{L}(\mathscr{B}) = \mathscr{L}(\mathscr{A})$. There exists an alternative initial configuration $\vec{t} = (t_0)_{p \in \mathbb{P}}$ such that $\mathscr{L}(\mathscr{B}(\vec{t})) = \mathscr{L}(\mathscr{A})$ and each process in $\mathscr{B}(\vec{t})$ is either fully-listening or trivial.*

Proof. Let $\mathscr{B}$ be an AA equivalent to $\mathscr{A}$. By Theorem 5 there exists a word w such that after reading w all processes in $\mathscr{B}$ that are not fully listening reached a bottom SCC, where they accept all communications on all channels they are listening to. By Lemma 6, there exists a word w' such that $ww' \backslash \mathscr{L}(\mathscr{B}) = \mathscr{L}(\mathscr{B})$. Let $\vec{t} = (t_p^0)_{p \in \mathbb{P}}$ be the states that processes in $\mathscr{B}$ reach after reading ww'. Then $\mathscr{L}(\mathscr{B}(\vec{t})) = \mathscr{L}(\mathscr{B})$. The theorem follows. □

5 Conclusion and Discussion

We study the addition of reconfiguration of communication to asynchronous automata. We show that in terms of expressiveness, the addition does not change the power of the model: every language recognized distributively by automata with reconfigurable communication can be recognized essentially by

the same automata with fixed communication. For deterministic automata this also means that the two are bisimilar. The same is (obviously) true in the other direction. However, the cost of conversion is in disseminating widely all the information and leaving it up to the processes whether to use it or not. We also show that this total dissemination cannot be avoided. Processes who do not get access to the full information about the computation become irrelevant and in fact do not participate in the distributed computation.

The issues of mobile and reconfigurable communication raise a question regarding "how much" communication is performed in a computation. Given a language recognized by an asynchronous automaton (distributively), the independence relation between letters is fixed by the language. It follows that two distributed systems in the form of asynchronous automata accepting (distributively) the same language must have the same independence relation between letters. However, this does not mean that they agree on the distribution of the alphabet. In case of two different distributed alphabets, what makes one better than the other? This question becomes even more important with systems with reconfigurable communication interfaces. Particularly, in reconfigurable asynchronous automata, the connectivity changes from state to state, which makes comparison even harder. How does one measure (and later reduce or minimize) the amount of communication in a system while maintaining the same behavior? We note that for the system in Section 4, the maximal number of channels a process is connected to is four regardless of how many channels are in the system. Dually, the asynchronous automaton for the same language requires every process that participates meaningfully in the interaction to have number of connections equivalent to the parameter n. Is less connectivity better than more connectivity?

The issues of "who is connected" and "with whom information is shared" also have implications for security and privacy. Reconfiguration allowed us to share communication only with those who "need to know". Fixed topology forced us to disseminate information widely. If we intend to use language models and models of concurrency in applications that involve security and privacy we need a way to reason about dissemination of information and comparing formalisms also based on knowledge and information.

Acknowledgments
We are grateful to Y. Abd Alrahman and L. Di Stefano for fruitful discussions and suggestions.

References

[1] Yehia Abd Alrahman, Rocco De Nicola & Michele Loreti (2019): *A calculus for collective-adaptive systems and its behavioural theory*. Information and Computation 268, p. 104457, doi:10.1016/j.ic.2019.104457.

[2] Yehia Abd Alrahman, Rocco De Nicola, Michele Loreti, Francesco Tiezzi & Roberto Vigo (2015): *A calculus for attribute-based communication*. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, pp. 1840–1845, doi:10.1145/2695664.2695668.

[3] Yehia Abd Alrahman & Nir Piterman (2021): *Modelling and verification of reconfigurable multi-agent systems*. Auton. Agents Multi Agent Syst. 35(2), p. 47, doi:10.1007/s10458-021-09521-x.

[4] Blaise Genest, Hugo Gimbert, Anca Muscholl & Igor Walukiewicz (2010): *Optimal Zielonka-type construction of deterministic asynchronous automata*. In: Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II 37, Springer, pp. 52–63, doi:10.1007/978-3-642-14162-1_5.

[5] Blaise Genest & Anca Muscholl (2006): *Constructing exponential-size deterministic Zielonka automata*. In: Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33, Springer, pp. 565–576, doi:10.1007/11787006_48.

[6] Siddharth Krishna & Anca Muscholl (2013): *A quadratic construction for Zielonka automata with acyclic communication structure*. Theoretical Computer Science 503, pp. 109–114, doi:10.1016/j.tcs.2013.07.015.

[7] Madhavan Mukund, K Narayan Kumar & Milind Sohoni (2000): *Synthesizing distributed finite-state systems from MSCs*. In: International Conference on Concurrency Theory, Springer, pp. 521–535, doi:10.1007/3-540-44618-4_37.

[8] Madhavan Mukund & Milind Sohoni (1997): *Keeping track of the latest gossip in a distributed system*. Distributed Computing 10, pp. 137–148, doi:10.1007/s004460050031.

[9] Wieslaw Zielonka (1987): *Notes on Finite Asynchronous Automata*. RAIRO Theor. Informatics Appl. 21(2), pp. 99–135, doi:10.1051/ita/1987210200991.

A Game-Theoretic Approach for Security Control Selection*

Dylan Léveillé Jason Jaskolka

Department of Systems and Computer Engineering
Carleton University, Ottawa, ON, Canada

dylan.leveille@carleton.ca jason.jaskolka@carleton.ca

Selecting the combination of security controls that will most effectively protect a system's assets is a difficult task. If the wrong controls are selected, the system may be left vulnerable to cyber-attacks that can impact the confidentiality, integrity and availability of critical data and services. In practical settings, it is not possible to select and implement every control possible. Instead considerations, such as budget, effectiveness, and dependencies among various controls, must be considered to choose a combination of security controls that best achieve a set of system security objectives. In this paper, we propose a game-theoretic approach for selecting effective combinations of security controls based on expected attacker profiles and a set budget. The control selection problem is set up as a two-person zero-sum one-shot game. Valid control combinations for selection are generated using an algebraic formalism to account for dependencies among selected controls. We demonstrate the proposed approach on an illustrative financial system used in government departments under four different scenarios. The results illustrate how a security analyst can use the proposed approach to guide and support decision-making in the control selection activity when developing secure systems.

1 Introduction

With computers becoming more interconnected than ever, there emerges an even greater need to secure computer systems and to effectively manage security risks. Security risks are mitigated by the implementation of a set of security controls. A *security control* refers to a safeguard or countermeasure prescribed for an information system or an organization designed to protect the confidentiality, integrity, and availability of its information and to meet a set of defined security requirements [28].

Control selection is an activity commonly found as part of a risk management process [10], a systems engineering process [28], the Risk Management Framework [13], the Cybersecurity Framework [25], or the Privacy Framework [24]. Control selection involves selecting and documenting the security controls necessary to protect the information system and organization commensurate with risk to organizational and system operations and assets, individuals, other organizations, and the nation [13].

During the control selection activity, security analysts typically select security controls from standardized security control catalogues, such as NIST SP 800-53 [15], ITSG-33 [8], ISO 27002 [11], CIS Critical Security Controls [4], and MITRE D3FEND™ [16], among others. However, selecting combinations of controls from these catalogues can be difficult for several reasons. First, these control catalogues are large, and many possible controls could be selected to mitigate the risks identified for a given system. In practical settings, it is not possible to select and implement every control possible. Considerations such as budget, effectiveness, and dependencies among various controls, must be considered to choose a combination of security controls that best achieve a set of system security objectives. Second, control selection is largely a human-oriented activity. The dynamics between security analysts (defenders) strategizing to protect critical systems and assets and achieve a set of security objectives, and attackers aiming

*Funded in part by the Human-Centric Cybersecurity Partnership under the SSHRC Partnership Grants program.

A. Achilleos and A. Francalanza (Eds.): Fifteenth International Symposium on Games, Automata, Logics, and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 103–119, doi:10.4204/EPTCS.409.11

© D. Léveillé & J. Jaskolka
This work is licensed under the Creative Commons Attribution License.

to impact critical systems and assets and violate those same security objectives must be considered when deciding on the most effective and cost-efficient combination of security controls. Although numerous optimization-based solutions are adept at accounting for various properties of the controls themselves, they fail to capture the human element that is inherently part of the control selection activity.

To address the above mentioned challenges, we propose a game-theoretic approach for security control selection. The human aspects of the control selection problem, as well as the large space of possible control combinations, and their dependencies and constraints, lends itself well to an application of game theory. Specifically, we set up a two-person zero-sum one-shot game which is played by a security analyst. The analyst selects their strategy based on an attacker profile, characterized by the expected targeted assets and security objectives. Each analyst strategy corresponds to a combination of security controls from a chosen control catalogue that are capable of achieving the security objectives. Valid control combinations are generated using an algebraic formalism (akin to product family algebra [9]) to account for dependencies among selected controls. The outcome of the game is a combination of suggested security controls that can effectively defend against the considered attacker profile. Using an illustrative governmental finance system, we demonstrate the proposed approach under four different scenarios.

The rest of this paper is organized as follows. Section 2 provides an overview of existing works on the topic of control selection and of game theory applications in cybersecurity. Section 3 presents the proposed game-theoretic approach for control selection. Section 4 provides an illustrative example demonstrating the application of the proposed approach. Section 5 discusses the benefits and potential difficulties with the proposed approach. Lastly, Section 6 concludes and briefly discusses future work.

2 Related Work

Many existing approaches to support the security control selection activity are based on setting and solving optimization problems. For example, for each considered control, Yevseyeva et al. [36] assign a probability of "survival" for each possible threat (i.e., the probability that the threat persists in the presence of the control). Probabilities are also assigned for the expected loss of successful attacks. The goal of the proposed approach is to minimize this expected loss, under constraints such as cost and system resources. Similarly, Almeida and Respício [1] also assign probabilities to controls based on their expected performance in mitigating certain vulnerabilities. For the proposed approach, the goal is to find the optimal controls for the system that will minimize an objective function accounting for both loss and cost. A different approach was proposed by Dewri et al. [5] where systems are modelled as trees, in which the leaf nodes represent possible attacks. Controls therefore mitigate one or many leaf nodes. With the attack impact, attack frequency, and cost of each control known, the optimal controls can be found by optimization. A similar tree-like approach was also proposed by Park and Huh [27]. While optimization-based approaches can account for important considerations and constraints such as cost and effectiveness, they depend heavily on assumptions about probabilities for threat likelihoods, or control success rates. Such probabilities are not likely to be accurately known in a practical setting.

Several other approaches for control selection that are not based on optimization have also been proposed. Bettaieb et al. [2] presented an approach where a machine learning model is trained with historical data from previous security assessments to make predictions using certain features of interest from a given security assessment to determine optimal controls. However, using historic data to determine how to protect a system has several limitations as every system is unique and may operate in widely different environments. In another work, Kiesling et al. [18] proposed a simulation-based approach to determine the optimal controls for a system. To do this, expected attacks are simulated on different

components of the system using different possible control combinations to find the optimal ones. This approach is noteworthy as it simply uses the properties of the controls and of the current system (such as different threats) to find the most optimal control combinations and does not depend on any probabilities.

Several works have explored the use of game theory for addressing cybersecurity challenges. For example, Nassar et al. [23] proposed a technique which focused on evaluating a system's network security with the help of a game model. Smith et al. [32] used game theory to verify the security of hardware designs. Wang et al. [35] presented a network attack-defence game to help secure a computer network. However, game theory has yet to be utilized for security control selection.

In contrast to existing work, the proposed approach aims to leverage game theory to address the shortcomings of current control selection approaches by placing a central focus on possible attacker behaviours, while also taking into account the considerations and constraints that limit the selection of certain combinations of security controls to effectively mitigate the threats to a system.

3 The Proposed Approach

In this section, we present our proposed game-theoretic approach for security control selection. An overview of the approach is shown in Figure 1. The approach consists of two main stages shown as swim lanes and six steps shown in blue. All steps are to be conducted by a security analyst. A detailed description of each step of the proposed approach is provided in the sections below.

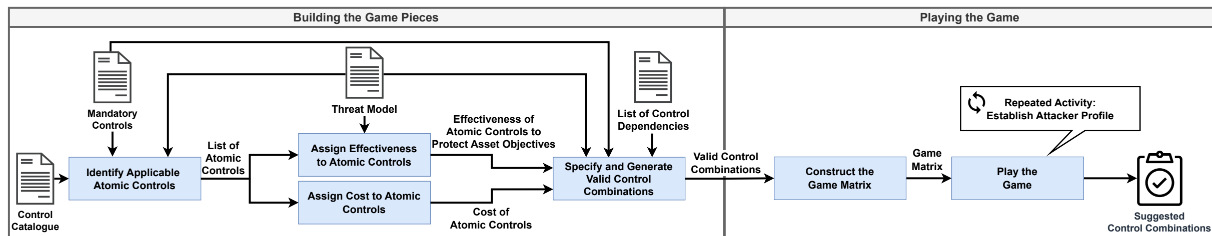


Figure 1: An overview of the proposed game-theoretic approach for security control selection

3.1 Identify Applicable Atomic Controls

The approach starts with the security analyst identifying applicable atomic controls from a given security control catalogue. In our context, the *atomic controls* are the smallest (indivisible) security controls that can be selected from a control catalogue. We say that a control is *applicable* to a system if it could provide any form of protection from the threats to the assets of the system.

We assume that a list of threats and assets are available to the security analyst in the form of a threat model. A threat model is defined as "a structured representation of all the information that affects the security of an application" [6]. Threat models typically include identified system threats and their impact on the assets within the system [6, 22]. The threat model can be obtained by applying a well-known threat modelling methodology such as STRIDE [21] or PASTA [34].

To determine the applicability of an atomic control, the security analyst must carefully consider each atomic control from the given control catalogue and decide if the control can mitigate the identified threats to the assets. Additionally, certain organizational needs or standards and regulations for the system's application domain may require that specific security controls be present in the system. The security analyst must therefore ensure that these *mandatory controls* are included as part of the set of

applicable controls identified. This is a manual process. However, it should be noted that the effort required for this activity is reasonable as security control catalogues are typically separated by control families which help guide an analyst in finding suitable controls [29]. Instructions and guidance for performing this task is well documented by ISO 27005 [12] and NIST SP 800-53B [14].

At the end of this step, the analyst will have a set of applicable atomic controls for the system. Note that the combination of suggested controls found by applying the proposed approach will be a subset of the controls gathered in this initial step.

3.2 Assign Effectiveness to Atomic Controls

For each identified atomic control, the analyst proceeds by assigning an effectiveness of the control at satisfying each *security objective* on each asset in the system. Security objectives represent the security needs of the assets on the system, such as confidentiality, integrity, and availability [3]. These objectives are normally included as part of the threat impacts described in the threat model. It is important to remember that the goal of the proposed approach is to create a game. In every game, there needs to be strategies, and payoffs defined for each strategy. Assigning the effectiveness of each atomic control therefore defines the payoffs of each atomic control in the game.

To perform this step of the approach, the atomic payoff matrix presented in Table 1 must be completed. The rows represent each atomic control that was identified in the previous step (denoted $C_1,\ldots,C_N$). The columns represent the security objectives for each asset (denoted $O_1,\ldots,O_M$). We expect the analyst to assign a value between 0 and 1 in each cell of this matrix. A value of 0 means that the atomic control is not effective at satisfying the specified objective for an asset, while a value of 1 means that the atomic control is completely effective at satisfying the specified objective for an asset. Each payoff value is therefore normalized. Provided that the rating scheme is selected and used consistently throughout the approach, the analyst is free to choose any method for assigning the effectiveness values for the atomic payoff matrix. For example, the analyst may choose to use a quantitative approach as in the Defect Detection and Prevention (DDP) risk reduction strategy developed by NASA [7], or they may alternatively choose to use a qualitative rating mapped to quantitative values as in [19].

Table 1: General form of the atomic payoff matrix

	Asset 1			*Asset 2*			...	*Asset X*		
	O_1	...	O_M	O_1	...	O_M	...	O_1	...	O_M
C_1										
$\vdots$										
C_N										

At the end of this step, the analyst will have the effectiveness of each applicable atomic control for satisfying each security objective on each asset in the system.

3.3 Assign Cost to Atomic Controls

In practical settings, cost or time constraints limit how many controls can be part of a system; if there are too many controls they may exceed a certain budget or cannot be implemented in reasonable time. In fact, without such constraints, there could technically be no limitations on the number of controls that can be selected for a system, and the best solution would be to select them all.

At the same time as assigning effectiveness, the analyst will also need to assign a cost for each identified atomic control. We expect the analyst to assign a cost from the set of real numbers $\mathbb{R}$. The units for cost could be represented as dollars, thousands of dollars, or any other form of currency as long as the same units are consistently used for all cost values. Furthermore, no units could be used if desired. Without units, costs simply represent an implementation effort.

After this step, the analyst will have the cost associated with each applicable atomic control.

3.4 Specify and Generate Valid Control Combinations

Given a set of applicable atomic controls, the analyst needs to specify and generate the set of valid control combinations that satisfies their constraints. To formally capture these constraints, we have decided to use an algebraic specification based on product family algebra [9] to specify and generate valid combinations of security controls.

Product family algebra extends the mathematical notions of semirings to describe and manipulate product families. A semiring is an algebraic structure $(S, +, \cdot, 0, 1)$ consisting of a set S with a commutative and associative binary operator $+$ and an associative binary operator $\cdot$. An element $0 \in S$ is the identity element with respect to $+$, while an element $1 \in S$ is the identity element with respect to $\cdot$. Additionally, $\cdot$ distributes over $+$ and element 0 annihilates S with respect to $\cdot$. A semiring is commutative if $\cdot$ is commutative and a semiring is idempotent if $+$ is idempotent.

For ease of presentation, we recast the vocabulary of product family engineering into the vocabulary of security controls by first defining a security control algebra to express families of security control combinations generated from a set of atomic controls.

Definition 1 (Security Control Algebra) *A* security control algebra *is a commutative idempotent semiring* $\mathscr{C} \stackrel{\text{def}}{=} (C, \oplus, \odot, 0, 1)$ *where each element of the semiring* $c \in C$ *is a security control family.*

In a security control algebra, the operator $\oplus$ is interpreted as a choice between two security control families and the operator $\odot$ is interpreted as a mandatory composition of two security control families[1]. The element 0 represents a non-implementable security control combination that cannot exist and the element 1 represents the empty security control combination which has no controls. A security control family is called a *security control combination* if it is indivisible with regard to the choice operator $\oplus$. Additionally, it is called a *proper security control combination* if $c \neq 0$. A security control combination is an *atomic control* if is it is indivisible with regard to the mandatory composition operator $\odot$. Optional controls are expressed as a choice between the controls and the empty security control combination 1. A list of optional controls $c_1, \dots, c_n$ is denoted by $opt[c_1, \dots, c_n] \stackrel{\text{def}}{=} (c_1 \oplus 1) \odot \cdots \odot (c_n \oplus 1)$.

For two security control families c_1 and c_2 in a security control algebra, the *refinement relation* ($\sqsubseteq$) is defined as $c_1 \sqsubseteq c_2 \stackrel{\text{def}}{\iff} \exists (c_3 \mid: c_1 \leq c_2 \odot c_3)$ where $\leq$ is the natural semiring order (i.e., $c_1 \leq c_2 \stackrel{\text{def}}{\iff} c_1 \oplus c_2 = c_2$). To specify constraints, such as dependencies between controls, we use the requirement relation.

Definition 2 (Requirement Relation [9]) *For elements* c_1, c_2, c_3, c_4 *and security control combination* x *in a security control algebra, the requirement relation* ($\rightarrow$) *is defined inductively as:*

$$
\begin{aligned}
c_1 \xrightarrow{x} c_2 \quad &\stackrel{\text{def}}{\iff} \quad x \sqsubseteq c_1 \implies x \sqsubseteq c_2 \\
c_1 \xrightarrow{c_3 \oplus c_4} c_2 \quad &\stackrel{\text{def}}{\iff} \quad c_1 \xrightarrow{c_3} c_2 \;\wedge\; c_1 \xrightarrow{c_4} c_2
\end{aligned}
$$

[1] When the context is clear, we omit the mandatory composition operator $\odot$ when specifying security control algebra terms.

For elements c_1, c_2 and x, the requirement relation $c_1 \xrightarrow{x} c_2$ can be read as "c_1 requires c_2 within x."

With this setting, all security control combinations can be specified algebraically by expressing the mandatory and optional controls as terms of a security control algebra along with requirement relations describing control dependencies.

The resulting specification serves as the basis for generating all possible proper security control combinations. However, not all control combinations are possible as some may exceed our defined budget. To make this determination we first define how to calculate the cost of a proper security control combination. In what follows, let $P \subseteq C$ be the set of all proper security control combinations in a security control algebra $\mathscr{C}$.

Definition 3 (Cost of a Proper Security Control Combination) *The cost of a proper security control combination $Cost : P \rightarrow \mathbb{R}$ is a function defined inductively for any proper security control combinations $a, b \in P$ in a security control algebra $\mathscr{C}$ as:*

$$\begin{aligned} Cost(1) &= 0 \\ Cost(a) &= G(a) \text{ if } a \text{ is atomic} \\ Cost(a \odot b) &= Cost(a) + Cost(b) \end{aligned}$$

where G is a function that returns the cost assigned to an atomic control (see Section 3.3).

Now that we can compute the cost of a proper security control combination, we determine the set of valid security control combinations. A *valid security control combination* is a proper security control combination that does not exceed the prescribed cost budget. The validity of a control combination is formalized in the following rule.

Definition 4 (Budget Rule) *For any $p \in P$ and budget B:*

$$Valid(p) \iff Cost(p) \leq B$$

After this step, the analyst will have a set of valid security control combinations that satisfy the prescribed budget. These valid security control combinations become the strategies that an analyst can select when playing the game.

3.5 Construct the Game Matrix

In this step, the analyst constructs the game matrix. The general form of the game matrix can be seen in Table 2. The rows represent the valid security control combinations found from the last step (denoted $Combo_1, \ldots, Combo_N$). The columns represent the security objectives for each asset (denoted $O_1, \ldots, O_M$). Note that the game matrix is identical in style to that of the atomic payoff matrix (see Table 1). The game matrix simply has control combinations as rows rather than atomic controls. In the game, the strategies of the security analyst will be the valid security control combinations, while the strategies of the attacker will be each security objective that could be violated on every asset.

Each outcome in a game is tied to a payoff [33]. In our game, the payoffs are represented from the perspective of the analyst and represent the effectiveness of the security control combinations towards every asset's security objectives. Just as cost was defined inductively, we can define a proper control combination's effectiveness towards an asset's security objective in a similar manner.

Table 2: General form of the game matrix

	Asset 1			*Asset 2*			$\dots$	*Asset X*		
	O_1	$\dots$	O_M	O_1	$\dots$	O_M	$\dots$	O_1	$\dots$	O_M
$Combo_1$										
$\vdots$										
$Combo_N$										

Definition 5 (Effectiveness of a Proper Security Control Combination) *The effectiveness of a proper security control combination towards an asset's security objective $Eff : P \to \mathbb{R}$ is a function defined inductively for any proper security control combinations $a,b \in P$ in a security control algebra $\mathscr{C}$ as:*

$$\begin{aligned} Eff(1) &= 0 \\ Eff(a) &= E(a) \text{ if } a \text{ is atomic} \\ Eff(a \odot b) &= 1-(1-Eff(a))(1-Eff(b)) \end{aligned}$$

where E is a function that returns the effectiveness assigned to an atomic control for an asset's security objective (see Section 3.2).

With Definition 5, the payoff values in the game matrix can be calculated. Note that the calculation of the effectiveness of a security control combination is inspired from the combined effectiveness calculation as part of NASA's DDP approach [7].

After this step, the analyst will have the game matrix so that they can proceed to play the game.

3.6 Play the Game

The game is a *two-person zero-sum one-shot game*. The game is played by *two persons*: the security analyst and the attacker. The attacker may embody one or multiple entities, but acts as a unified adversary. The goal of the security analyst is to select the security control combination that will best protect the security objectives for the assets they believe will be targeted by the attacker. Only one security control combination can be selected, hence it is a *one-shot* game. On the other hand, the goal of the attacker is to attack assets and violate corresponding security objectives. An attacker could attack one or many assets and violate one or more objectives from a series of attacks. Regardless, an attacker will select which assets and objectives they will target and will commit to attacking the selected assets and objectives. The attacker will naturally prefer attacking assets which are not properly defended, i.e., those for which there are minimally effective security controls. The effectiveness values in the game matrix (payoffs) do not directly correlate to a loss to the attacker. However, it is easy to see that the higher the values, the more difficult it is for an attacker to conduct a successful attack leading to corresponding security objective violations. Therefore, what the security analyst gains in effectiveness is what the attacker loses in their ability to successfully conduct their attack; hence, it is a *zero-sum* game. Note that this game is strictly non-cooperative; the analyst and attacker are competing directly and would never want to cooperate.

Using the game matrix, the analyst must select a strategy (i.e., a valid security control combination) to play that will best protect the system assets and security objectives that they believe are most important. To do this, an analyst must establish the expected attacker profile. An *attacker profile* is an expected set of the assets and corresponding security objectives targeted by the attacker. One can imagine different classes of attackers having different capabilities, and different targets, thereby establishing different attacker profiles. In the context of a game, an attacker profile corresponds to guessing the attacker strategy

so that it can be defended. This consideration of the dynamics of the analyst and the attacker strategies in this game is what differentiates it from existing security control selection approaches.

It is impossible to know exactly which security objectives on which assets will be attacked, so assumptions must be made. One way to do this is to determine where most of the critical information flows in the system and which assets may be prone to more attacks (i.e., have more expected threats). The combination of these ideas can help localize assets that are more attractive for attacks, and therefore puts the security objectives of these assets at higher risk of violation. Another way to do this is to consider the risk to each asset and corresponding security objectives for the identified threats to the system (which we consider known to the analyst). In this case, prioritizing defence of assets and security objectives targeted by high risk threats may be a good approach. Regardless, once the attacker profile is determined, then the suggested strategy (i.e., the most effective security control combination) can be found.

Regardless of the approach taken to establish the attacker profile, it will articulate the objectives that are expected to be violated by an attacker. For this work, we establish an attacker profile by considering and prioritizing different attacker objectives. Attacker objectives correspond to a set of security objectives for some assets that are equally expected to be targeted by an attacker. Within an attacker profile, several attacker objectives may be prioritized according to their perceived likelihood of being targeted by the attacker to obtain a priority order for the objectives. For example, the security analyst could establish an attacker profile in which the attacker has two ordered attacker objectives: (1) to target the confidentiality of two specific assets equally, and (2) to target the integrity of two other assets equally. The security analyst may consider as many attacker objectives as they desire when developing an attacker profile. The suggested analyst strategies for an attacker profile will be those which maximize the *total effectiveness* across each attacker objectives (i.e., the sum of the effectiveness returned by Definition 5 for the security objectives in the attacker objectives is maximized in the priority order). To better understand this concept, an example of the strategies found by playing the game with an attacker profile with two ordered attacker objectives is visualized in Figure 2.

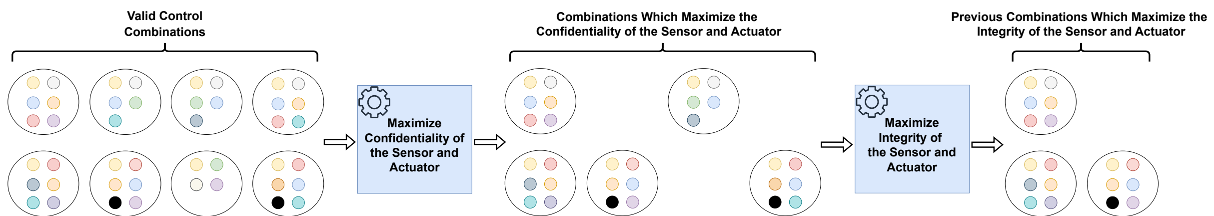


Figure 2: Finding the suggested controls for an attacker profile with multiple ordered attacker objectives

In this example, there are initially eight valid security control combinations. Each security control combination has a unique set of controls (denoted by the different coloured dots in the figure). Only two assets exist in this system; a Sensor and an Actuator. The attacker profile has two ordered attacker objectives: (1) the confidentiality of the Sensor and the Actuator and then (2) the integrity of the Sensor and the Actuator. From all valid control combinations, the control combinations which maximize the first set of attacker objectives is found, yielding five different combinations. From these five combinations, the combinations which maximize the second set of attacker objectives is found, yielding three control combinations. As there are no more ordered attacker objectives, the resulting control combinations are all considered equally valid, and represent the suggested strategies. Note that since the suggested strategies are derived through a series of maximization problems, it may be possible for more than one strategy to be the most effective for a given attacker profile.

At the end of this step, the analyst will obtain at least one strategy that best protects against the considered attacker profile and that corresponds to the suggested security control combinations to be implemented in the system. It is important to remember that this approach is a game. Therefore, as with any game, it is recommended that the game be re-constructed with different maximum budget values and re-played with different attacker profiles (as illustrated in Figure 1). This can help gauge and compare the control combinations that should be used for the system under different constraints and goals.

4 Illustrative Example

In this section, we demonstrate how the approach presented in Section 3 could be applied to support the control selection activity for an illustrative example system. Suppose a security analyst needs to select a combination of cost-effective security controls to protect a financial system used by the Canadian government called *Firebird*. An overview of the system architecture is shown in Figure 3. *Firebird* allows financial analysts to enter data about financial transactions and view those transactions through a user interface. Many identical interfaces may exist. The interfaces communicate over a 5G channel to a central processing system to process the commands from the analyst. A database stores the financial transactions data used by the central processing system. Both the processing system and database are located in an internal government network. The security analyst will apply the proposed game-theoretic approach for security control selection for the *Firebird* system as described in the following sections.

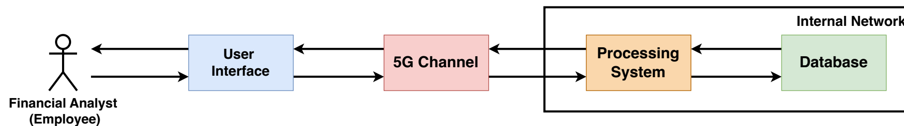


Figure 3: An overview of the *Firebird* system architecture

4.1 Identify Applicable Atomic Controls

Considering the *Firebird* system architecture shown in Figure 3, there are four primary assets: the user interface, the 5G channel, the processing system, and the database. For simplicity and brevity, suppose that the analyst is primarily focused on addressing threats to the user interface and the database and threats to the 5G channel and processing system are being handled by another analyst. Also, it has been pre-determined by the security analyst's government department that they are primarily concerned with the confidentiality (C), integrity (I), and availability (A) security objectives for the system assets.

Because *Firebird* is a Canadian government system, the analyst selects controls from the ITSG-33 control catalogue[2]. To comply with departmental requirements, it was decided by the security analyst's government department that the input validation control (i.e., *SI-10* in ITSG-33) must be present in the system. This is because improper input validation in any system can result in potentially severe consequences [17, 30, 31].

The analyst is provided with the fragment of the threat model consisting of the assets, threats, and violated security objectives for the system as shown the first three columns of Table 3. By consulting the control catalogue, the analyst decides which of the atomic controls are relevant in protecting the system by referring to the identified threats in the threat model. The applicable atomic controls found for each threat can also be seen as part of Table 3. Notice that the mandatory control (*SI-10: Input Validation*) is

[2]ITSG-33 is the standard control catalogue to assist security practitioners in their efforts to protect information systems in compliance with applicable Government of Canada legislation, policies, directives, and standards [8].

included in the gathered set of atomic controls; the other controls therefore represent optional controls that may or may not be included as part of the suggested controls of this approach.

Table 3: Threat model and applicable atomic controls for *Firebird*

Assets	Threats	Security Objectives Violated	Applicable Atomic Controls
User Interface	• Commands received from unknown sources	• Confidentiality • Integrity	• *AC-4: Information Flow Enforcement*
	• Improper/malicious commands entered	• Confidentiality • Integrity	• *SI-10: Input Validation*
	• Employee freely accesses and changes features provided in the interface	• Confidentiality • Integrity	• *AC-3: Access Enforcement* • *AC-6: Least Privilege*
Database	• SQL injection from an improper analyst input changes or retrieves data	• Confidentiality • Integrity	• *AC-4: Information Flow Enforcement* • *SI-10: Input Validation*
	• Employee freely inspects data in the database	• Confidentiality	• *AC-6: Least Privilege*

4.2 Assign Effectiveness to Atomic Controls

The analyst must now assign the effectiveness values for each identified applicable atomic control at mitigating the threats and protecting the security objectives listed in Table 3. The analyst has elected to assign qualitative ratings for the effectiveness of each atomic control that are mapped to quantitative values. The considered ratings and corresponding values are adopted and adapted from the metrics in the Common Vulnerability Scoring System (CVSS) [20] and include: *None* (0.0), *Low* (0.2), *Medium* (0.5), *High* (0.8), and *Very High* (0.9). No rating was assigned to the value of 1 as it is unrealistic to expect a single control to fully protect a security objective.

With these metrics, the analyst develops the atomic payoff matrix, as illustrated in Table 4. Note that none of the identified controls protect asset availability; this is fine as no threats towards availability were identified in the threat model (see Table 3). Therefore, protecting availability is not required.

Table 4: Atomic payoff matrix for *Firebird*

	Database			*User Interface*		
	C	*I*	*A*	*C*	*I*	*A*
SI-10: Input Validation	Medium	Very High	None	Medium	High	None
AC-3: Access Enforcement	None	None	None	Medium	High	None
AC-4: Information Flow Enforcement	Medium	Medium	None	Medium	Low	None
AC-6: Least Privilege	High	None	None	Medium	Low	None

4.3 Assign Cost to Atomic Controls

The analyst also assigns a cost for each identified atomic control as shown in Table 5. No units were used for each cost as it was decided that cost could best be represented as a unit of effort for this particular system. Additionally, the analyst's department has allocated a total budget (expressed as effort) of $B = 15$.

Table 5: Atomic control costs for *Firebird*

Control	Cost
SI-10: Input Validation	5
AC-3: Access Enforcement	6
AC-4: Information Flow Enforcement	4
AC-6: Least Privilege	3

4.4 Specify and Generate Valid Control Combinations

Next, the analyst must determine the valid security control combinations that could be considered for the system. To do this, they use security control algebra to specify the security control family from the mandatory and optional atomic controls identified in the previous steps. Recall that *SI-10: Input Validation* is a mandatory control and that *AC-3: Access Enforcement*, *AC-4: Information Flow Enforcement* and *AC-6: Least Privilege* are optional controls.

Suppose the security analyst has determined that to implement any access enforcement policy (such as Role-Based Access Control) a least privilege approach to protecting the data in the system must first be implemented. Therefore there is a dependency between *AC-3: Access Enforcement* and *AC-6: Least Privilege*. The analyst must consider this dependency in the specification of the security control family.

Denoting the security control family as F, the security control family for this example is specified as the following security control algebra term and requirement relation.

$$F \quad = \quad \mathit{SI\text{-}10} \odot \mathit{opt}\big[\mathit{AC\text{-}3}, \mathit{AC\text{-}4}, \mathit{AC\text{-}6}\big] \qquad \text{such that} \qquad \mathit{AC\text{-}3} \xrightarrow{F} \mathit{AC\text{-}6}$$

The possible security control combinations are generated by expanding the specification of the security control family F subject to the requirement relation. The possible security control combinations for F along with their costs calculated using Definition 3 are shown in Table 6. Note that the security control combinations *SI-10 AC-3* and *SI-10 AC-3 AC-4* are not part of the security control family F because they do not respect the specified requirement relation.

The analyst now determines the validity of the possible security control combinations according to the Budget Rule (Definition 4). Recall that the total budget B is 15. Therefore, applying the Budget Rule for each security control combination, it is easy to see that all control combinations, except for *Combo 6*, satisfy the rule and are therefore valid. As a result, *Combo 6* is no longer considered.

Table 6: Security control combination costs for *Firebird*

ID	Security Control Combination	Cost
Combo 1	*SI-10*	5
Combo 2	*SI-10 AC-4*	9
Combo 3	*SI-10 AC-6*	8
Combo 4	*SI-10 AC-3 AC-6*	14
Combo 5	*SI-10 AC-4 AC-6*	12
Combo 6	*SI-10 AC-3 AC-4 AC-6*	18

4.5 Construct the Game Matrix

Now that the analyst knows all of the valid security control combinations, the game matrix can be constructed. The payoff of each valid security control combination for each asset's security objectives is found by applying Definition 5. The resulting game matrix is shown in Table 7.

4.6 Play the Game

With the game matrix constructed, the security analyst can now find a suggested security combination to protect the security objectives of the considered assets for the system. To do this, the security analyst can play the game considering different attacker profiles captured by the scenarios described below. Table 8 presents the total effectiveness of each strategy in the game for each of the attacker objectives used in each scenario. Noteworthy effectiveness values are highlighted in bold. For strategies that have been excluded for specific attacker objectives, the corresponding effectiveness is noted as "N/A".

Table 7: Game matrix for *Firebird*

	Database			*User Interface*		
	C	*I*	*A*	*C*	*I*	*A*
Combo 1	0.5	0.9	0.0	0.5	0.8	0.0
Combo 2	0.75	0.95	0.0	0.75	0.84	0.0
Combo 3	0.9	0.9	0.0	0.75	0.84	0.0
Combo 4	0.9	0.9	0.0	0.875	0.968	0.0
Combo 5	0.95	0.95	0.0	0.875	0.872	0.0

Table 8: Total effectiveness of game strategies against different attacker objectives for *Firebird*

	Scenario 1	**Scenario 2**	**Scenario 3**		**Scenario 4**	
	AO1.1	*AO2.1*	*AO3.1*	*AO3.2*	*AO4.1*	*AO4.2*
Combo 1	1.0	2.7	0.5	N/A	0.9	N/A
Combo 2	1.50	3.29	0.75	N/A	**0.95**	1.59
Combo 3	1.65	3.39	0.75	N/A	0.9	N/A
Combo 4	1.775	3.643	**0.875**	**0.968**	0.9	N/A
Combo 5	**1.825**	**3.647**	**0.875**	0.872	**0.95**	**1.822**

Scenario 1: This scenario considers an attacker profile where the attacker equally targets the confidentiality of the database and the confidentiality of the user interface (*AO1.1*). By playing the game against this attacker, the suggested security control combination to implement is *Combo 5* because it has the greatest total effectiveness (1.825) for defending against the attacker objectives. Given that all identified threats impact the confidentiality of both assets, the suggested combination includes the optional controls which maximize confidentiality across both assets. While *AC-3* does not provide any protection to the confidentiality of the database, both *AC-4* and *AC-6* protect confidentiality across both assets. Given that *SI-10* is mandatory, *Combo 5* is the logical choice. Note that by disregarding *AC-3*, the threat related to "employee freely accesses and changes features provided in the interface" on the user interface is mitigated only through *AC-6*. Since *AC-6* is not as effective as *AC-3* for protecting integrity, the user interface's integrity is at higher risk of being violated. However, this is an acceptable risk given that the expected behaviour of the attacker is not interested in violating any integrity objectives.

Scenario 2: This scenario considers an attacker profile where the attacker equally targets all of the objectives (confidentiality, integrity, and availability) of each asset (database and user interface) (*AO2.1*). By playing the game against this attacker, the suggested security control combination to implement is *Combo 5* because it has the greatest total effectiveness (3.647) for defending against the attacker objectives. Given that this attacker profile aims to violate all security objectives on all assets, the suggested combination includes the optional controls which maximize the confidentiality and integrity across both assets. *AC-4* stands out in this regard, as it effectively safeguards all security objectives unlike *AC-3* and *AC-6*. While *AC-3* is not effective towards any of the database security objectives, *AC-6* at least offers protection towards the confidentiality of the database. Given that *SI-10* is mandatory, *Combo 5* is again the logical choice. Note that by disregarding *AC-4*, the same (acceptable) risk is imposed on the system as in Scenario 1. Also note that an assumed attacker profile targeting all objectives leads to a strategy that best balances the security objectives across all assets.

Scenario 3: This scenario considers an attacker profile where the attacker has two ordered attacker objectives to target: the confidentiality of the user interface (*AO3.1*) and then the integrity of the user

interface (*AO3.2*). By playing the game against this attacker, the suggested security control is determined by first considering how to best defend against the highest priority attacker objectives. This leaves *Combo 4* and *Combo 5* since they each have the greatest total effectiveness (0.875) for protecting the confidentiality of the user interface. Given that all controls provide the same effectiveness for the confidentiality of the user interface, any valid combination which maximizes this security objective is ideal. We now consider the next highest priority attacker objective from these possible security control combinations. Now, the suggested security control combination to implement is *Combo 4* because it has a greater total effectiveness (0.968 versus 0.872 for *Combo 5*) for protecting the integrity of the user interface. Given that *Combo 4* and *Combo 5* differ by only one control, and that *AC-3* is more effective at protecting the integrity of the user interface than *AC-4*, it follows that *Combo 4* is the logical choice. Note that by disregarding *AC-4*, the threat related to "commands received from unknown sources" on the user interface is not addressed. However, this is an acceptable risk given the expected attacker profile.

Scenario 4: This scenario considers an attacker profile where the attacker has two ordered attacker objectives to target: the integrity of the database (*AO4.1*) and then equally the confidentiality of the database and the integrity of the user interface (*AO4.2*). By playing the game against this attacker, the suggested security control is determined by first considering how to best defend against the highest priority attacker objective. This leaves *Combo 2* and *Combo 5* since they each have the greatest total effectiveness (0.95) for protecting the integrity of the database. These combinations are logical as *AC-4* is the only optional control that protects the integrity of the database. The next highest priority attacker objective must then be considered for these possible security control combinations. Now, the suggested security control combination to implement is *Combo 5* because it has a greater total effectiveness (1.822 versus 1.59 for *Combo 2*) for protecting confidentiality of the database and the integrity of the user interface. Given that *Combo 5* additionally contains *AC-6* which protects against the second set of attacker objectives, it follows that *Combo 5* is the logical choice. Again, as *Combo 5* disregards *AC-4*, the same risk is imposed on the system as in Scenario 1 and Scenario 2. However, this is an acceptable risk given the expected attacker profile.

This illustrative example and the corresponding scenarios highlight the fact that security control selection can indeed be seen as a game, and that the suggested controls to use depends greatly on the expected attacker profiles. The proposed approach considers all the constraints and considerations required to perform control selection, and in contrast to existing approaches, also takes into consideration the expected attacker behaviours; an important factor to this problem that helps justify controls of interest and the risk to leave certain threats unaddressed.

5 Discussion

Approaching security control selection as a game emphasizes the human element in deciding how to most effectively protect a system's assets under various considerations such as budgetary constraints. Selecting controls for a system is indeed a human-centric problem as the large number of potential controls to use from a control catalogue can be overwhelming and could lead to many mistakes in the chosen combination of security controls selected for the system. To expand on this point, selecting security controls exclusively on technical considerations while overlooking attacker behaviours is a fundamentally flawed approach in addressing this issue, as ultimately, it is humans which are conducting attacks. Given that the proposed approach emphasizes the need to reflect on potential attacker behaviours, it prioritizes

the human-centric aspect to find its solutions. Additionally, viewing this problem as a game captures the opposing dynamics of the attacker and analyst, aligning with the real-world motivations of both actors.

Unfortunately, a limitation with many existing game-theoretic approaches from addressing cyber-security challenges is their heavy reliance on assumptions. Specifically, game theory depends on assumptions on the players, such as the players knowing every strategy available to them, knowing the probabilities of every move, and knowing the payoff functions [26]. Compared to existing works, the proposed approach can be used practically as it does not rely on assigning probabilities for the likelihood of attacks succeeding, and instead focuses on general assumptions about which security objectives could be violated by the attacker. Security analysts cannot realistically predict exact probabilities of attack, but they can make informed assumptions regarding which system components might be more attractive to attackers. These considerations ensure that the proposed approach is systematic, repeatable, and realistic, thereby minimizing the influence of human bias on the results of the game and eliminating many of the required assumptions with existing game-theoretic approaches.

While the proposed approach may seem limited by the security analyst's certainty in the effectiveness values for each atomic control, a sensitivity analysis was performed on the effectiveness metrics used in Section 4 and revealed that the results of the approach were not sensitive to these values. The full details of this analysis are omitted due to space limitations. In any case, to allow the analyst to express their uncertainty, it is possible to allow them to provide multiple values when assigning the effectiveness for atomic controls. The modifications to the game under these conditions is left for future work. The proposed approach also currently requires manual effort on the part of the security analyst. While some tasks are unavoidably manual (e.g.,, selecting applicable atomic controls and assigning effectiveness and costs to those controls), the rest of the approach can be supported with automated tools. Lastly, as atomic controls are considered indivisible components, we assume costs can be independently assigned to each atomic control. From a business perspective, this assumption may not always hold as the aggregation of certain controls could result in lower total costs to implement some control combinations. We argue that this limitation is unlikely to occur unless the controls are provided by third party vendors. In such cases, the cost function outlined in Definition 3 can be adapted to combine costs in a different manner.

6 Conclusions and Future Work

Ensuring effective security controls are selected for a system can greatly impact its security. In this work, a game-theoretic approach to security control selection is proposed in which a game is played by a security analyst to determine security controls which best mitigate expected attacker profiles. To create the game, the controls which are believed to secure the system must first be gathered. Following this, the effectiveness and cost of each of these controls is determined. After every possible control combination is generated, the effectiveness and cost of each combination is calculated and the game matrix can be constructed. The game can be played with many different expected attacker profiles and will suggest unique sets of controls for each. The suggested controls can help make a security analyst feel more confident in their decision to implement some controls over others.

In future work, we aim to extend the approach to consider more than one effectiveness value for each control to account for uncertainties in the effectiveness values assigned by the analyst. This would result in the game being played with more than one game matrix. The suggested controls from these different matrices could be compared to guide a security analyst in selecting the controls for the system. Additionally, to support the calculation of the game outcomes for large systems with many controls, we aim to develop software tools to automate aspects of the approach.

References

[1] Luís Almeida & Ana Respício (2018): *Decision support for selecting information security controls*. Journal of Decision Systems 27, pp. 173–180, doi:10.1080/12460125.2018.1468177.

[2] Seifeddine Bettaieb, Seung Yeob Shin, Mehrdad Sabetzadeh, Lionel C. Briand, Michael Garceau & Antoine Meyers (2020): *Using machine learning to assist with the selection of security controls during security assessment*. Empirical Software Engineering 25(4), pp. 2550–2582, doi:10.1007/s10664-020-09814-x.

[3] Jennifer Cawthra, Michael Ekstrom, Lauren Lusty, Julian Sexton & John Sweetnam (2020): *Data Integrity: Detecting and Responding to Ransomware and Other Destructive Events*. Special Publication (NIST SP) 1800-26, National Institute of Standards and Technology, doi:10.6028/NIST.SP.1800-26.

[4] Center for Information Security (2021): *CIS Critical Security Controls – Version 8*. `https://www.cisecurity.org/controls/v8` [Accessed: 2024-06-21].

[5] Rinku Dewri, Nayot Poolsappasit, Indrajit Ray & Darrell Whitley (2007): *Optimal security hardening using multi-objective optimization on attack tree models of networks*. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, Association for Computing Machinery, New York, NY, USA, pp. 204–213, doi:10.1145/1315245.1315272.

[6] Victoria Drake: *Threat Modeling*. `https://owasp.org/www-community/Threat_Modeling` [Accessed: 2023-12-11].

[7] Martin S. Feather, Steven L. Cornford, Kenneth A. Hicks & Kenneth R. Johnson: (2005): *Applications of tool support for risk-informed requirements reasoning*. `https://www.researchgate.net/publication/220403935_Applications_of_tool_support_for_risk-informed_requirements_reasoning` [Accessed: 2024-06-21].

[8] Government of Canada (2014): *IT Security Risk Management: A Lifecycle Approach – Security Control Catalogue*. `https://www.cisecurity.org/controls/v8` [Accessed: 2024-06-21].

[9] Peter Höfner, Ridha Khedri & Bernhard Möller (2011): *An Algebra of Product Families*. Software and Systems Modeling 10(2), pp. 161–182, doi:10.1007/s10270-009-0127-2.

[10] International Organization for Standardization (2018): *ISO/IEC 31000:2018 Risk Management – Guidelines*. `https://www.iso.org/standard/65694.html` [Accessed: 2024-06-21].

[11] International Organization for Standardization (2022): *ISO/IEC 27002:2022 Information security, cybersecurity and privacy protection – Information security controls*. `https://www.iso.org/standard/75652.html` [Accessed: 2024-06-21].

[12] International Organization for Standardization (2022): *ISO/IEC 27005:2022 Information security, cybersecurity and privacy protection – Guidance on managing information security risks*. `https://www.iso.org/standard/80585.html` [Accessed: 2023-12-11].

[13] Joint Task Force Interagency Working Group (2018): *Risk Management Framework for Information Systems and Organizations: A System Life Cycle Approach for Security and Privacy*. Special Publication (NIST SP) 800-37 Revision 2, National Institute of Standards and Technology, doi:10.6028/NIST.SP.800-37r2.

[14] Joint Task Force Interagency Working Group (2020): *Control Baselines for Information Systems and Organizations*. Special Publication (NIST SP) 800-53B, National Institute of Standards and Technology, doi:10.6028/nist.sp.800-53b.

[15] Joint Task Force Interagency Working Group (2020): *Security and Privacy Controls for Information Systems and Organizations*. Special Publication (NIST SP) 800-53 Revision 5, National Institute of Standards and Technology, doi:10.6028/NIST.SP.800-53r5.

[16] Peter Kaloroumakis & Michael Smith (2020): *Toward a Knowledge Graph of Cybersecurity Countermeasures*. `https://apps.dtic.mil/sti/citations/AD1156977` [Accessed: 2024-06-21].

[17] Osamah Ibrahim Khalaf, Munsif Sokiyna, Youseef Alotaibi, Abdulmajeed Alsufyani & Saleh Alghamdi (2021): *Web Attack Detection Using the Input Validation Method: DPDA Theory*. Computers, Materials & Continua 68(3), doi:10.32604/cmc.2021.016099.

[18] Elmar Kiesling, Andreas Ekelhart, Bernhard Grill, Christine Strauss & Christian Stummer (2016): *Selecting security control portfolios: a multi-objective simulation-optimization approach*. *EURO Journal on Decision Processes* 4(1-2), pp. 85–117, doi:10.1007/s40070-016-0055-7.

[19] Qixu Liu & Yuqing Zhang (2011): *VRSS: A New System for Rating and Scoring Vulnerabilities*. *Computer Communications* 34, pp. 264–273, doi:10.1016/j.comcom.2010.04.006.

[20] Peter Mell, Karen Scarfone & Sasha Romanosky (2007): *The Common Vulnerability Scoring System (CVSS) and Its Applicability to Federal Agency Systems*. NIST Interagency Report 7435, National Institute of Standards and Technology, doi:10.6028/NIST.IR.7435.

[21] Microsoft (2022): *Microsoft Threat Modeling Tool – Threats*. `https://learn.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats` [Accessed: 2024-06-21].

[22] Murugiah Souppaya and Karen Scarfone (2016): *Guide to Data-Centric System Threat Modeling*. `https://csrc.nist.gov/pubs/sp/800/154/ipd` [Accessed: 2024-06-21].

[23] Mohamed Nassar, Joseph Khoury, Abdelkarim Erradi & Elias Bou-Harb (2021): *Game Theoretical Model for Cybersecurity Risk Assessment of Industrial Control Systems*. In: *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pp. 1–7, doi:10.1109/NTMS49979.2021.9432668.

[24] National Institute of Standards and Technology (2020): *The NIST Privacy Framework: A Tool for Improving Privacy through Enterprise Risk Management*. Cybersecurity White Papers (CSWP) 10, National Institute of Standards and Technology, doi:10.6028/nist.cswp.10.

[25] National Institute of Standards and Technology (2024): *The NIST Cybersecurity Framework (CSF) 2.0*. Cybersecurity White Papers (CSWP) 29, National Institute of Standards and Technology, doi:10.6028/NIST.CSWP.29.

[26] Guillermo Owen (2015): *Game Theory*. In James D. Wright, editor: *International Encyclopedia of the Social & Behavioral Sciences (Second Edition)*, second edition edition, Elsevier, Oxford, pp. 573–581, doi:10.1016/B978-0-08-097086-8.43045-X.

[27] Jun Young Park & Eui Nam Huh (2020): *A cost-optimization scheme using security vulnerability measurement for efficient security enhancement*. *Journal of Information Processing Systems* 16(1), pp. 61–82, doi:10.3745/JIPS.02.0128.

[28] Ron Ross, Victoria Pillitteri, Richard Graubart, Deborah Bodeau & Rosalie Mcquaid (2021): *Developing Cyber-Resilient Systems: A Systems Security Engineering Approach*. Special Publication (NIST SP) 800-160, Volume 2 Revision 1, National Institute of Standards and Technology, doi:10.6028/NIST.SP.800-160v2r1.

[29] Quentin Rouland, Stojanche Gjorcheski & Jason Jaskolka (2023): *Eliciting a Security Architecture Requirements Baseline from Standards and Regulations*. In: *2023 IEEE 31st International Requirements Engineering Conference Workshops*, REW, Hannover, Germany, pp. 224–229, doi:10.1109/rew57809.2023.00045.

[30] Theodoor Scholte, Davide Balzarotti & Engin Kirda (2012): *Have things changed now? An empirical study on input validation vulnerabilities in web applications*. *Computers & Security* 31(3), pp. 344–356, doi:10.1016/j.cose.2011.12.013.

[31] Theodoor Scholte, William Robertson, Davide Balzarotti & Engin Kirda (2012): *Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis*. In: *2012 IEEE 36th Annual Computer Software and Applications Conference*, pp. 233–243, doi:10.1109/COMPSAC.2012.34.

[32] Andrew M. Smith, Jackson R. Mayo, Vivian Kammler, Robert C. Armstrong & Yevgeniy Vorobeychik (2017): *Using computational game theory to guide verification and security in hardware designs*. In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 110–115, doi:10.1109/HST.2017.7951808.

[33] Philip D. Straffin (1993): *Game Theory and Strategy*, second edition. The Mathematical Association of America.

[34] Tony UcedaVélez & Marco M. Morana (2015): *Risk Centric Threat Modeling: Process for Attack Simulation and Threat Analysis*, first edition. John Wiley & Sons, doi:10.1002/9781118988374.

[35] Baoyi Wang, Jianqiang Cai, Shaomin Zhang & Jun Li (2010): *A network security assessment model based on attack-defense game theory*. In: *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, 3, pp. V3–639–V3–643, doi:10.1109/ICCASM.2010.5620536.

[36] Iryna Yevseyeva, Vitor Basto-Fernandes, Michael Emmerich & Aad Van Moorsel (2015): *Selecting Optimal Subset of Security Controls*. *Procedia Computer Science* 64, pp. 1035–1042, doi:10.1016/j.procs.2015.08.625.

Epistemic Skills
Logical Dynamics of Knowing and Forgetting

Xiaolong Liang
School of Philosophy
Shanxi University
Taiyuan, Shanxi, P.R. China
lianghillon@gmail.com

Yì N. Wáng*
Department of Philosophy (Zhuhai)
Sun Yat-sen University
Zhuhai, Guangdong, P.R. China
ynw@xixilogic.org

We present a type of epistemic logics that encapsulates both the dynamics of acquiring knowledge (knowing) and losing information (forgetting), alongside the integration of group knowledge concepts. Our approach is underpinned by a system of weighted models, which introduces an "epistemic skills" metric to effectively represent the epistemic abilities associated with knowledge update. In this framework, the acquisition of knowledge is modeled as a result of upskilling, whereas forgetting is by downskilling. Additionally, our framework allows us to explore the concept of "knowability," which can be defined as the potential to acquire knowledge through upskilling, and facilitates a nuanced understanding of the distinctions between epistemic de re and de dicto expressions. We study the computational complexity of model checking problems for these logics, providing insights into both the theoretical underpinnings and practical implications of our approach.

1 Introduction

The study of epistemic logic has become a prolific area within applied modal logic, since its inception as a formal methodology in epistemology [34, 21], and its subsequent application in computer science [16, 27]. A longstanding focus of this field has been to elucidate various forms of group knowledge, with mutual knowledge (what everyone knows), common knowledge, and distributed knowledge being particularly well-known concepts.

On top of this has been an exploration of actions that bring about changes in knowledge, such as the effect of public announcements. This inquiry has given rise to the subfield of dynamic epistemic logic [13], a discipline that incorporates update modalities into its language to depict knowledge updates, with Public Announcement Logic [30] and Action Model Logic [7] being popular approaches (the first can be viewed as a specific instance of the broader framework of the latter). Extensions of Public Announcement Logic that incorporate the concept of knowability have then garnered significant interest [5, 2]. These extensions delve into the nuanced understanding of what it means for something to be knowable in a dynamic informational context.

The literature presents a diverse array of approaches to model the phenomenon of forgetting within the frameworks of both classical and non-classical logics. Among these approaches, two prominent categories emerge: syntactical and semantical strategies for representing knowledge contraction. Syntactical strategies, such as those delineated by the AGM paradigm [3], typically involve the removal of formulas from an agent's knowledge base, akin to belief contraction. On the other hand, semantical strategies focus on the modification of the interpretation of knowledge. This can include various methods such as erasing the truth values assigned to atomic propositions [26, 22, 12, 36]. Another semantical method involves updating the set of propositions that an agent is aware of [15].

*Corresponding author.

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 120–137, doi:10.4204/EPTCS.409.12

© Liang and Wáng
This work is licensed under the
Creative Commons Attribution License.

In this study, we propose a unified logical framework designed to model group knowledge, processes that may lead to knowledge update and epistemic necessity and possibility. Our approach is based on weighted modal logic [23, 20]. We extend this foundation by introducing the concept of *epistemic skills*, utilizing weights assigned to the edges in our model to represent the specific skills required to distinguish between pairs of possible worlds. This differentiation introduces a measure of similarity, aligning our work with recent developments in epistemic logic that employ concepts of similarity or distance[28, 14].

Traditionally understood notions such as mutual and common knowledge are preserved in their classical interpretations within our framework. Additionally, we incorporate distributed and field knowledge seamlessly. Our model explicitly defines the skill set each agent possesses, and by leveraging update modalities, we model the acquisition, loss, revision of knowledge as results of upskilling, downskilling and reskilling, respectively. By focusing on operations that modify one's skills, we broaden our analysis to include the concepts of knowability and forgetfulness. In keeping with the perspectives suggested in [5], our guiding principles are: *the knowable is what becomes known after upskilling,* and conversely, *the forgettable is what becomes unknown upon downskilling*. This framework also allows for a more nuanced understanding of the *de re* and *de dicto* distinctions in epistemic sentences.

The structure of the paper is as follows: Section 2 is dedicated to presenting the formal syntax and semantics of our proposed logics. This section also includes a discussion on the use of *epistemic de re* and *de dicto* expressions within our framework. The subsequent section delves into an in-depth analysis of the computational complexity associated with the model checking problems in these logics. The paper concludes with Section 4, where we offer our concluding remarks and reflections on the study.

2 Logics

We extend classical epistemic logic [16, 27] with a mechanism of epistemic skills in the models, allowing us a consistent way of modeling *knowing* and *forgetting*, as well as various notions of group knowledge (such as, *distributed knowledge* and *field knowledge*).

We fix three countably infinite sets before the introduction of formal languages. Namely, P for the *set of atomic propositions* (*atoms* for short), A for the *set of agents* and S for the *set of epistemic skills* (capabilities, professions, or privileges). For simplicity, these sets are unchanged throughout the paper, although it is also possible to treat them as changeable parameters of each of the languages.

2.1 Syntax

The biggest language that we introduce now, named $\mathcal{L}_{CDEF+-=\equiv\boxplus\boxminus\Box}$, has its grammar given as follows:

$$\varphi ::= p \mid \neg\varphi \mid (\varphi \to \varphi) \mid K_a\varphi \mid C_G\varphi \mid D_G\varphi \mid E_G\varphi \mid F_G\varphi \mid \\ (+_S)_a\varphi \mid (-_S)_a\varphi \mid (=_S)_a\varphi \mid (\equiv_b)_a\varphi \mid \boxplus_a\varphi \mid \boxminus_a\varphi \mid \Box_a\varphi$$

where $p \in \mathsf{P}$, $a, b \in \mathsf{A}$, $G \subseteq \mathsf{A}$ is a finite nonempty group, and $S \subseteq \mathsf{S}$ is a finite nonempty skill set.

As the name shows, we are interested in some of its sublanguages. The basic language $\mathcal{L}$ allows a grammar that builds recursively from atomic propositions with Boolean operators (we choose negation and implication to be primitives) and the modal operator K_a (with $a \in \mathsf{A}$) which is used to characterize *individual knowledge*. Namely, $\mathcal{L}$ is the formal language for classical multi-agent epistemic logic.

Four types of modalities, C_G, D_G, E_G and F_G, are introduced for *common knowledge*, *distributed knowledge*, *mutual knowledge* and *field knowledge*, respectively. In naming a language that extends the basic language, we use combinations of the letters C, D, E and F to indicate the inclusion common,

distributed, mutual or field knowledge operators. For example, $\mathcal{L}_{DF}$ denotes the language that extends the basic language with distributed and field knowledge.

We consider four update modalities, $(+_S)_a$, $(-_S)_a$, $(=_S)_a$ and $(\equiv_b)_a$, where $a,b \in \mathsf{A}$ and S is a finite nonempty subset of S, which are intended to mean the action of agent a's expansion with skills S (*upskilling*), subtraction of skills S (*downskilling*), assigning skill set S (*reskilling*) and *learning* from agent b, respectively. These operators are self dual, as one can verify after the semantics is introduced.

Another three operators, $\boxplus_a$, $\boxminus_a$ and $\Box_a$, are used to mean the action of a's addition, subtraction and modification of an arbitrary skill set, respectively. Their dual operators are written as $\oplus_a$, $\ominus_a$ and $\Diamond_a$, respectively, but treated to be non-primitive.

We shall use the symbols $+$, $-$, $=$, $\equiv$, $\boxplus$, $\boxminus$ and $\Box$ in subscript to signal the introduction of each of the update operators or quantifiers. For example, $\mathcal{L}_{F+\boxplus}$ stands for the language that extends the basic language with field knowledge and the operators $(+_S)_a$ and $\boxplus_a$ (for any $a \in \mathsf{A}$ and $S \subseteq \mathsf{S}$).

As a result, we reach as many as 2^{11} $(= 2048)$ languages in total, though many of the combinations may not be of our focus. Other Boolean operators are defined just as in classical logic. When we refer to a *formula*, we are indicating an element of one of these languages, and its specific reference will depend on the context unless otherwise specified.

2.2 Semantics

We introduce a type of models for the interpretation of the languages.

Definition 1. *A* model *is a quadruple* (W,E,C,β) *where:*

- W *is a nonempty set of (possible) worlds or states;*
- $E : W \times W \to \wp(\mathsf{S})$*, an* edge function*, assigns each pair of worlds a skill set;*
- $C : \mathsf{A} \to \wp(\mathsf{S})$ *is a* capability function *that assigns a skill set to each agent;*
- $\beta : W \to \wp(\mathsf{P})$ *is a valuation.*

and satisfies the following two conditions:

- *Positivity: for all* $w,u \in W$*, if* $E(w,u) = \mathsf{S}$*, then* $w = u$*;*
- *Symmetry: for all* $w,u \in W$*,* $E(w,u) = E(u,w)$*.*

In the above definition, the function E assigns a skill set to each edge (a pair of worlds), indicating that only individuals with skills outside the set can distinguish between the pair of worlds. The criteria for satisfaction are defined as follows.

Definition 2. *Given a formula* φ*, a model* $M = (W,E,C,\beta)$ *and* $w \in W$*, we say* φ *is* true *or* satisfied *at* w *in* M*, denoted* $M,w \models \varphi$*, if the following hold inductively:*

$M,w \models p \iff p \in \beta(w)$

$M,w \models \neg\psi \iff$ *not* $M,w \models \psi$

$M,w \models (\psi \to \chi) \iff$ *if* $M,w \models \psi$ *then* $M,w \models \chi$

$M,w \models K_a\psi \iff$ *for all* $u \in W$*, if* $C(a) \subseteq E(w,u)$ *then* $M,u \models \psi$

$M,w \models E_G\psi \iff M,w \models K_a\psi$ *for all* $a \in G$

$M,w \models C_G\psi \iff$ *for all positive integers* n*,* $M,w \models E_G^n\psi$*, with* $E_G^1\psi := E_G\psi$ *and* $E_G^n\psi := E_G^1 E_G^{n-1}\psi$

$M,w \models D_G\psi \iff$ *for all* $u \in W$*, if* $\bigcup_{a\in G} C(a) \subseteq E(w,u)$ *then* $M,u \models \psi$

$M,w \models F_G\psi \iff$ *for all* $u \in W$*, if* $\bigcap_{a\in G} C(a) \subseteq E(w,u)$ *then* $M,u \models \psi$

$M,w \models (+_S)_a\psi \Leftrightarrow (W,E,C^{a+S},\beta),w \models \psi,$ *with* $C^{a+S}(a) = C(a) \cup S$ *and* $(\forall x \in \mathsf{A} \setminus \{a\})\ C^{a+S}(x) = C(x)$
$M,w \models (-_S)_a\psi \Leftrightarrow (W,E,C^{a-S},\beta),w \models \psi,$ *with* $C^{a-S}(a) = C(a) \setminus S$ *and* $(\forall x \in \mathsf{A} \setminus \{a\})\ C^{a-S}(x) = C(x)$
$M,w \models (=_S)_a\psi \Leftrightarrow (W,E,C^{a=S},\beta),w \models \psi,$ *with* $C^{a=S}(a) = S$ *and* $(\forall x \in \mathsf{A} \setminus \{a\})\ C^{a=S}(x) = C(x)$
$M,w \models (\equiv_b)_a\psi \Leftrightarrow (W,E,C^{a\equiv b},\beta),w \models \psi,$ *with* $C^{a\equiv b}(a) = C(b)$ *and* $(\forall x \in \mathsf{A} \setminus \{a\})\ C^{a\equiv b}(x) = C(x)$
$M,w \models \boxplus_a\psi \quad \Leftrightarrow$ *for all finite nonempty* $S \subseteq \mathsf{S}$, $M,w \models (+_S)_a\psi$
$M,w \models \boxminus_a\psi \quad \Leftrightarrow$ *for all finite nonempty* $S \subseteq \mathsf{S}$, $M,w \models (-_S)_a\psi$
$M,w \models \square_a\psi \quad \Leftrightarrow$ *for all finite nonempty* $S \subseteq \mathsf{S}$, $M,w \models (=_S)_a\psi$.

Given that G is a finite group, it is clear that the formula $E_G\psi$ is logically equivalent to the $\bigwedge_{a\in G} K_a\psi$. However, this equivalence impacts both the succinctness of the language and the complexity of model checking. Consequently, $E_G\psi$ cannot be treated merely as a straightforward rewriting of $\bigwedge_{a\in G} K_a\psi$.

Note that although $(=_S)_a\varphi$ is not a legal formula when S is the empty set $\emptyset$, we can regard it as a defined formula, i.e., $(=_\emptyset)_a\varphi := (=_S)_a(-_S)_a\varphi$ (for any qualified set S). In the mean time, it is not hard to verify that both $(+_\emptyset)_a\varphi$ and $(-_\emptyset)_a\varphi$, if allowed, are equivalent to φ, so there is no need to worry about the cases with empty sets.

The logics (i.e., the sets of valid formulas) that are defined by the above semantics and correspond to our languages will bear the same names, but will be denoted using upright roman typeface, e.g., L, $\mathrm{L}_{F+\boxplus}$ and $\mathrm{L}_{CDEF+-=\equiv\boxplus\boxminus\square}$.

2.3 Representation of a model and truths within it

In this section, we describe an exemplary model and demonstrate several true formulas applicable within this model. Let $s_1, s_2, s_3, s_4, s_5 \in \mathsf{S}$ represent epistemic skills, and $a, b, c \in \mathsf{A}$ denote agents. The model $M = (W,E,C,\beta)$ is defined as follows:

- $W = \{w_1, w_2, w_3, w_4, w_5\}$ is the set of possible worlds.
- $E : W \times W \to \wp(\mathsf{S})$ is the symmetric closure that satisfies the following:
  - $E(w_1,w_1) = E(w_2,w_2) = E(w_3,w_3) = E(w_4,w_4) = E(w_5,w_5) = \{s_1,s_2,s_3,s_4\}$,
  - $E(w_1,w_2) = E(w_3,w_5) = \{s_1,s_4\}$,
  - $E(w_1,w_3) = E(w_2,w_5) = \{s_1,s_2,s_3\}$,
  - $E(w_1,w_4) = \emptyset$,
  - $E(w_1,w_5) = E(w_2,w_3) = \{s_1\}$,
  - $E(w_2,w_4) = \{s_2,s_3\}$,
  - $E(w_3,w_4) = \{s_4\}$,
  - $E(w_4,w_5) = \{s_2,s_3,s_4\}$.
- C is the capability function that assigns a skill set to each agent, a, b and c:
  - $C(a) = \{s_1,s_2,s_3\}$,
  - $C(b) = \{s_2,s_3,s_4\}$,
  - $C(c) = \{s_4\}$.
- β assigns sets of propositions to each world:
  - $\beta(w_1) = \{p_1,p_2\}$
  - $\beta(w_2) = \{p_1,p_3\}$
  - $\beta(w_3) = \{p_1,p_2,p_4\}$
  - $\beta(w_4) = \{p_3,p_4\}$
  - $\beta(w_5) = \{p_1,p_3,p_4\}$.

The fact that M is a model can be easily verified, and using a diagram to represent M is often helpful (see Figure 1). In the diagram, nodes represent worlds, and undirected edges represent accessibility relations. Each edge is labeled with the skill set that facilitate accessibility between two worlds. If an edge is labeled with an empty set, it indicates no accessibility (as between w_1 and w_4), and such edges are not drawn in the diagram. This helps in visualizing the connections and structure of the model more clearly.

Readers can verify the following logical truths in model $M = (W,E,C,\beta)$ given above:

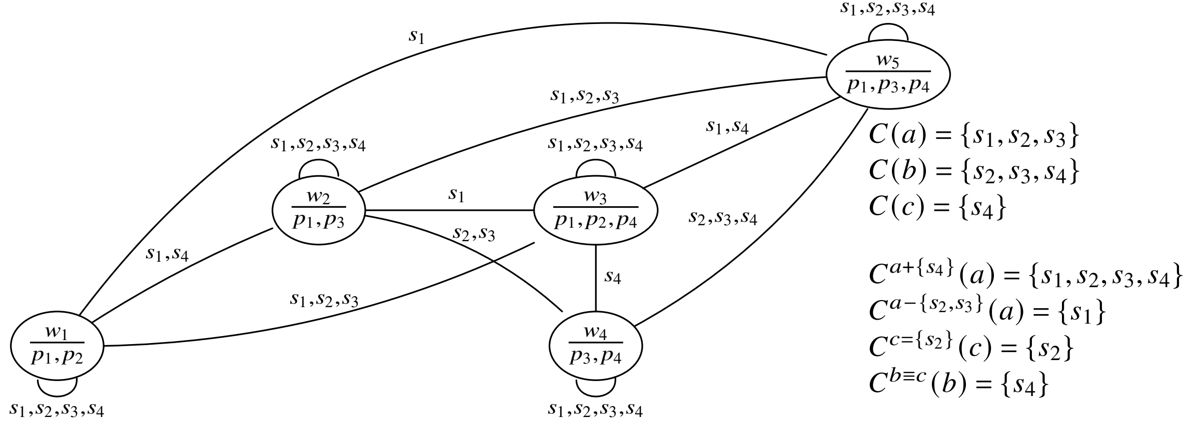


Figure 1: Illustration of the model M.

1. $M,w_2 \models K_a p_3$, indicating that in world w_2, agent a knows proposition p_3.
2. $M,w_4 \models \neg K_b p_1 \wedge \neg K_b \neg p_1$, meaning that in world w_4, agent b does not know whether proposition p_1 is true or false.
3. $M,w_3 \models K_c(K_a p_3 \vee K_a \neg p_3)$, demonstrating that in world w_3, agent c knows whether agent a knows proposition p_3.
4. $M,w_4 \models E_{\{a,b\}}(p_3 \wedge p_4)$, showing that both agents a and b know propositions p_3 and p_4 in w_4.
5. $M,w_5 \models (\neg C_{\{a,c\}} p_1 \wedge \neg C_{\{a,c\}} \neg p_1) \wedge (\neg C_{\{a,c\}} p_2 \wedge \neg C_{\{a,c\}} \neg p_2)$, indicating that neither the truth nor the falsity of propositions p_1 and p_2 are common knowledge between a and c in world w_5.
6. $M,w_4 \models D_{\{a,b\}}(\neg p_1 \wedge p_4)$, indicating that in world w_4, the knowledge that proposition p_1 is false and p_4 is true is distributed between agents a and b.
7. $M,w_4 \models \neg F_{\{a,b\}} \neg p_1 \wedge \neg F_{\{a,b\}} p_4$, showing that in world w_4, neither $\neg p_1$ nor p_4 are field knowledge for agents a and b.
8. $M,w_5 \models \neg K_a p_4 \wedge (+_{\{s_4\}})_a K_a p_4$. Here, in world w_5, agent a initially does not know p_4, but can learn it upon acquiring skill s_4.
9. $M,w_2 \models K_a p_3 \wedge (-_{\{s_2,s_3\}})_a \neg K_a p_3$, indicating that in world w_2, agent a knows p_3 but would forget it if she loses skills s_2 and s_3.
10. $M,w_1 \models E_{\{a,b\}}(\neg K_c p_2 \wedge (=_{\{s_2\}})_c K_c p_2))$. This means that in world w_1, it is mutual knowledge between agents a and b that, c does not know p_2, but would know it if her skill set is $\{s_2\}$.
11. $M,w_1 \models (\equiv_c)_b \bigwedge_{p \in \{p_1,\dots,p_4\}} (F_{\{b,c\}} p \leftrightarrow K_b p)$. This result suggests that in world w_1, if agent b changes her skill set to match that of agent c, her knowledge will align with the field knowledge shared between them.
12. $M,w_5 \models \oplus_a K_a p_4$, indicating that in world w_5, there exists a potential skill upgrade under which agent a can come to know p_4.
13. $M,w_3 \models \ominus_b \bigwedge_{p \in \{p_1,\dots,p_4\}} (\neg C_{\{a,b\}} p \wedge \neg C_{\{a,b\}} \neg p)$, highlighting that in world w_3, it is possible through some downskilling for agent b to reach a state where no propositions p_1 through p_4 are common knowledge between agents a and b.
14. $M,w_2 \models K_c p_1 \wedge \neg K_c p_3 \wedge \Diamond_c(\neg K_c p_1 \wedge K_c p_3)$, indicating that in world w_2, it is currently the case that agent c knows p_1 and does not know p_3, but there is a possible skill update which would make agent c unaware of p_1 while becoming aware of p_3.

2.4 Epistemic de re and de dicto

The distinction between epistemic *de re* and *de dicto* modalities was already discussed in [34], with *de re* modalities concerning whether a specific thing possesses or lacks a certain property, and *de dicto* modalities concerning whether a proposition is true or false. Subsequently, as [31] suggests, this distinction becomes more apparent in a formal language when quantifiers over terms are utilized. In the realm of epistemic logic, a *de re* sentence can be expressed as, "There exists a term x such that an entity knows that x possesses or lacks a certain property." Conversely, a *de dicto* sentence can be formulated as, "An entity knows that there exists a term which possesses or lacks a certain property."

In the logics introduced in this paper, we are not only able to distinguish between *de re* and *de dicto* modalities, but can also delineate two specific types of de re sentences (compare with the case in Group Announcement Logic [2, Section 6]):

- Knowing de dicto: "Agent a knows (with her current skills) that there exists a set S of skills such that, with S, she can achieve φ in world w of model (W,E,C,β)."
  Formally, this is expressed as: $(\forall u \in W)[C(a) \subseteq E(w,u) \Rightarrow (\exists S \subseteq \mathsf{S})\ (W,E,C^{a+S},\beta),u \models \varphi]$.
- Explicitly knowing de re: "There exists a set S of skills such that agent a knows with her current skill set, that with S in addition, she can achieve φ in world w of model (W,E,C,β)."
  Formally, this is represented as: $(\exists S \subseteq \mathsf{S})(\forall u \in W)[C(a) \subseteq E(w,u) \Rightarrow (W,E,C^{a+S},\beta),u \models \varphi]$.
- Implicitly knowing de re: "There exists a set S of skills such that agent a knows, with the addition of S to her skill set, that she can achieve φ in world w of model (W,E,C,β)."
  Formally, this is depicted as: $(\exists S \subseteq \mathsf{S})(\forall u \in W)[C^{a+S}(a) \subseteq E(w,u) \Rightarrow (W,E,C^{a+S},\beta),u \models \varphi]$.

Although the distinction between *de dicto* and *de re* knowledge remains clear, the nuanced difference between *implicit* and *explicit de re* knowledge hinges on whether the skills from the skill set S are included for the agent to formulate her knowledge.

These distinctions elucidate the complex interplay between knowledge and capabilities in dynamic epistemic scenarios, highlighting subtle differences in how agents process information depending on their skill sets and the nature of their knowledge. All three types of knowledge are expressible using the formal languages we have introduced. Here is how each type is represented:

Proposition 3.

1. *Knowledge de dicto is expressed by the formula* $K_a \oplus_a \varphi$.
2. *Explicit knowledge de re is expressed by the formula* $(\equiv_a)_c \oplus_c K_a (\equiv_c)_a \varphi$ *(where c is not in* φ*).*
3. *Implicit knowledge de re is expressed by the formula* $\oplus_a K_a \varphi$.

Proof. Clauses 1 and 3 are straightforwardly validated by the semantics. We focus here on clause 2, where c is any agent not appearing in φ:

$$
\begin{aligned}
& (\exists S \subseteq \mathsf{S})(\forall u \in W)\ C(a) \subseteq E(w,u) \Rightarrow (W,E,C^{a+S},\beta),u \models \varphi \\
\iff\ & (\exists S \subseteq \mathsf{S})(\forall u \in W)\ C(a) \subseteq E(w,u) \Rightarrow (W,E,((C^{c\equiv a})^{c+S})^{a\equiv c},\beta),u \models \varphi \\
\iff\ & (\exists S \subseteq \mathsf{S})(\forall u \in W)\ C(a) \subseteq E(w,u) \Rightarrow (W,E,(C^{c\equiv a})^{c+S},\beta),u \models (\equiv_c)_a \varphi \\
\iff\ & (\exists S \subseteq \mathsf{S})(W,E,(C^{c\equiv a})^{c+S},\beta),w \models K_a (\equiv_c)_a \varphi \\
\iff\ & (W,E,C^{c\equiv a},\beta),w \models \oplus_c K_a (\equiv_c)_a \varphi \\
\iff\ & (W,E,C,\beta),w \models (\equiv_a)_c \oplus_c K_a (\equiv_c)_a \varphi.
\end{aligned}
$$

Examples of different types of knowledge expressed by formulas The formula $D_G \diamondplus_a \diamondminus_b \varphi$ says that, "It is group G's distributed knowledge that, with the addition of certain skills by agent a, it becomes achievable that, even with the loss of certain skills by agent b, φ can still be achieved." This use pertains to be *de dicto*. The formula $(\equiv_a)_c \Diamond_c K_a (\equiv_c)_a \varphi$ (where c does not appear in φ) expresses that, "There exists a skill set such that agent a knows that, with exactly this skill set, a can achieve φ." This use is *explicitly de re*. The formula $\Diamond_a K_a \varphi$ says that, "There is an update of agent a's skill set through which a knows she can make φ true." This is *implicitly de re*.

Remark 1. *For simplicity, the initial definitions of* knowledge de dicto*,* explicit *and* implict knowledge de re *have been presented primarily for individual knowledge using the operator K_a and the actions of knowing represented by the quantifier $\boxplus_a$. These concepts can be readily extended to include:*

- Group knowledge*, utilizing operators such as C_G, D_G, E_G and F_G,*
- Quantifiers over actions of downskilling *and* reskilling*, represented by $\boxminus_a$ and $\Box_a$ respectively,*
- Nested actions *and* dynamic changes *among agents.*

For example, the formula $F_G \diamondminus_{a_1} \diamondplus_{a_2} \Diamond_{a_3} \diamondminus_{a_4} \varphi$ represents an epistemic de dicto *statement involving field knowledge and multiple actions (upskilling, downskilling and reskilling) among different agents. The expression $(\equiv_{d_1})_{c_1} \diamondplus_{c_1} (\equiv_{d_2})_{c_2} \diamondplus_{c_2} (\equiv_{d_3})_{c_3} \diamondplus_{c_3} E_I (\equiv_{c_1})_{d_1} (\equiv_{c_2})_{d_2} (\equiv_{c_3})_{d_3} \varphi$ captures* explicit knowledge de re *involving nested contexts and multiple agents, linked to mutual knowledge. Similarly, the formula $\diamondplus_{b_1} \Diamond_{b_2} \diamondminus_{b_3} D_H \varphi$ illustrates* implicit knowledge de re *involving a sequence of updates and distributed knowledge. In these examples, the agents $a_1, a_2, a_3, a_4, b_1, b_2, b_3, c_1, c_2, c_3, d_1, d_2, d_3$ are not specifically restricted to being within or outside the groups G, H or I. This flexibility allows for a broad application of the concepts across various contexts and group dynamics.*

In dynamic epistemic logic, the distinction between *knowing de dicto* and *knowing de re* is enriched through the use of quantifiers for updates, closely aligning with the philosophical inquiries into *knowing that* versus *knowing how*. While previous solutions such as those presented in [5, 2, 6] primarily adopt a syntactical approach, our logic introduces a semantical perspective, providing an alternative to the topological semantics discussed by [33, 9].

3 Complexity of Model Checking

In this section, we study the computational complexity of the model checking problem for the logics introduced in the preceding sections. The *model checking problem* for a logic involves verifying whether a specified formula φ, within a given finite model M and at a particular world w in the model, holds true; formally, whether $M, w \models \varphi$.

3.1 The input

We define the measure of the input. The *length* of a formula φ, denoted $|\varphi|$, is defined to be the number of symbols that occur in φ (including the symbols for brackets), just as in [16, Section 3.1]; or more precisely defined inductively by the structure of φ, i.e., when φ is:

- An atomic formula p: $|p| = 1$;
- Negation $\neg\psi$: $|\neg\psi| = |\psi| + 1$;
- Implication $(\psi \to \chi)$: $|(\psi \to \chi)| = |\psi| + |\chi| + 3$;

- Individual knowledge $K_a\psi$: $|K_a\psi| = |\psi| + 2$;
- Group knowledge: $|C_G\psi| = |\psi| + 2|G| + 2$, and similarly for $D_G\psi$, $E_G\psi$ and $F_G\psi$; e.g., $|(p \to C_{\{a,b,c\}}q)| = 13$;
- An update modality: $|(+_S)_a\psi| = 2|S| + |\psi| + 5$, similarly for $(-_S)_a\psi$ and $(=_S)_a\psi$, and $|(\equiv_b)_a\psi| = |\psi| + 5$;
- A quantifier: $|\boxplus_a \psi| = |\psi| + 2$, and also for $\boxminus_a\psi$ and $\Box_a\psi$.

The *size* of a model $M = (W, E, C, \beta)$, denoted $|M|$, is defined as the sum of the following components:

- $|W|$: the size of the domain; [1]
- $|E|$: since E consists of triples (w, u, S) where $w, u \in W$ and $S \subseteq \mathsf{S}$, the size of E is determined by the number of the symbols used to denote this set;
- $|C|$ with respect to a given set A of agents: C is composed pairs (a, S) where $a \in A$ and $S \subseteq \mathsf{S}$; the size of C is the count of all symbols used for its representation; [2]
- $|\beta|$: the function β consists of pairs (w, Φ) where $w \in W$ and $\Phi \subseteq \mathsf{P}$; the size of β is the number of the symbols used to represent this set.

Finally, for formula φ and model M (with a designated world w), the *size of the input* is $|\varphi| + |M| + 1$.

3.2 Model checking for logics without quantifiers: in P

We commence by presenting a polynomial-time algorithm designed to ascertain the truth of classical epistemic formulas in a specified world within a given model, addressing the model checking problem for L. Subsequently, we enhance the algorithm to incorporate group knowledge modalities. This extension allows us to establish that the model checking problem for L_{CDEF} fall within the complexity class P. We then proceed to further broaden our results to encompass update modalities, achieving the results for the model checking problems for $\mathrm{L}_{CDEF+-=\equiv}$ and all of its sublolgics.

3.2.1 Model checking in L

Given a model $M = (W, E, C, \beta)$, a world $w \in W$ and a formula φ, we decide whether $M, w \models \varphi$. In order to do so, we present an algorithm (Algorithm 1) for calculating $Val(M, \varphi)$, the *truth set* of φ in M, i.e., $\{x \in W \mid M, x \models \varphi\}$. The question about whether $M, w \models \varphi$ holds is thus reduced to the membership testing in $Val(M, \varphi)$, which takes at most $|W|$ steps in addition to the time costs on computing $Val(M, \varphi)$.

It is not hard to verify that $Val(M, \varphi)$ is indeed the set of worlds of M at which φ is true. In particular, in the case for the K_a operator,

$$\begin{aligned} M, w \models K_a\psi &\iff \forall y \in W : C(a) \subseteq E(w, y) \Rightarrow M, y \models \psi \\ &\iff \forall y \in W : C(a) \subseteq E(w, y) \Rightarrow y \in Val(M, \psi) \qquad \text{(IH)} \\ &\iff w \in \{x \in W \mid \forall y \in W : C(a) \subseteq E(x, y) \Rightarrow y \in Val(M, \psi)\} \end{aligned}$$

The cost for computing $Val(M, \varphi)$ is in polynomial time. In the case for $K_a\psi$—the most time-consuming case here—there are two while-loops over W, and checking $C(a) \subseteq E(x, y)$ costs at most

[1] Model checking is typically impractical for infinite sets due to computational limitations; therefore, we restrict our analysis to finite sets. This is consistently applied in the following discussions as well.

[2] Theoretically, the function C maps each agent (from an infinite set) to a specific skill set. This mapping is not feasible with finite input, but in practical scenarios, we limit the number of agents. It is essential to ensure that the set A includes all agents relevant to the formula being checked.

Algorithm 1 Function $Val(M,\varphi)$: computing the truth set for basic formulas

```
Input: model M = (W, E, C, β) and formula φ
Output: {x | M, x ⊨ φ}
 1: Initialize: tmpVal ← ∅
 2: if φ = p then return {x ∈ W | p ∈ β(x)}
 3: else if φ = ¬ψ then return W \ Val(M, ψ)
 4: else if φ = ψ → χ then
 5:     return (W \ Val(M, ψ)) ∪ Val(M, χ)
 6: else if φ = K_a ψ then
 7:     for all x ∈ W do
 8:         Initialize: n ← true
 9:         for all y ∈ W do
10:             if C(a) ⊆ E(x, y) and y ∉ Val(M, ψ) then
                    n ← false
11:         if n = true then tmpVal ← tmpVal ∪ {x}
12:     return tmpVal ▷ This returns {x ∈ W | ∀y ∈ W : C(a) ⊆ E(x, y) ⇒ y ∈ Val(M, ψ)}
```

$|C|\cdot|E|$ steps, and the membership checking $y \notin Val(M,\psi)$ (when $Val(M,\psi)$ is at hand) takes at most $|W|$ steps; so this case costs at most $|W|^2\cdot(|C|\cdot|E|+|W|)$. Moreover, the algorithm for computing $Val(M,\varphi)$ calls itself recursively, but only for a subformula of φ, and the maximum number of recursion is bounded by $|\varphi|$, i.e., the length of φ. So the total time cost for computing $Val(M,\varphi)$ is $|W|^2\cdot(|C|\cdot|E|+|W|)\cdot|\varphi|$. Considering the input size, we find that the total time cost is within $O(n^5)$. So the following lemma holds.

Lemma 4. *The model checking problem for* L *is in P.*

3.2.2 Model checking group knowledge

Building on the previous result, we now aim to encompass scenarios that include group knowledge. To facilitate this extension, we will first introduce a definition and a couple of lemmas that underpin it.

Definition 5. *For a formula φ, let $A_\varphi = \{G \mid$ "E_G" or "C_G" appears in $\varphi\}$. For a model $M = (W,E,C,\beta)$,*

- *For all worlds $w,u \in W$, $E_\varphi(w,u) = E(w,u) \cup \{G \in A_\varphi \mid (\exists a \in G)\ C(a) \subseteq E(w,u)\}$,*
- *For all worlds $w,u \in W$, $E^+_\varphi(w,u) = E_\varphi(w,u) \cup \{G \in A_\varphi \mid (\exists n \geq 1)(\exists w_0,\ldots,w_n \in W)\ w_0 = w$ and $w_n = u$ and $G \in \bigcap_{0\leq i<n} E_\varphi(w_i,w_{i+1})\}$,*

where without loss of generality we assume that $A_\varphi \cap \mathsf{A} = \emptyset$. For short, we write M^+_φ for (W,E^+_φ,C,β).

It should be noted that the above definition involves an abuse of notation by treating groups of agents as skills. To ensure formal correctness, a one-to-one mapping can be defined from each group to a new skill in S.

Proposition 6. *For any model M and any formula φ, M^+_φ is a model.* □

Lemma 7. *Given formulas φ and χ, a group G, a model M and a world w of M:*

1. *$M,w \models \varphi$ iff $M^+_\chi,w \models \varphi$;*
2. *If "C_G" appears in χ, then $M,w \models C_G\varphi$ iff $M,u \models \varphi$ for any world u such that $G \in E^+_\chi(w,u)$.*

Proof. 1. For any agent a, formula χ and world w,u, we have $C(a) \subseteq E(w,u)$ iff $C(a) \subseteq E_\chi(w,u)$ iff $C(a) \subseteq E^+_\chi(w,u)$. Thus it is easy to verify that (M,w) and (M^+_χ,w) satisfy exactly the same formulas.

2. We first verify the base case for $C_G\varphi$:

$$\begin{array}{rcl}
M,w \models E_G\varphi & \Longleftrightarrow & \text{for any } a \in G,\ M,w \models K_a\varphi \\
& \Longleftrightarrow & \text{for any } a \in G \text{ and } u \in W,\ C(a) \subseteq E(w,u) \text{ implies } M,u \models \varphi \\
& \Longleftrightarrow & \text{for any } u \in W \text{ and } a \in G,\ C(a) \subseteq E(w,u) \text{ implies } M,u \models \varphi \\
& \Longleftrightarrow & \text{for any } u \in W,\ M,u \models \varphi \text{ if } C(a) \subseteq E(w,u) \text{ for some } a \in G \\
& \Longleftrightarrow & \text{for any } u \in W,\ G \in E_\chi(w,u) \text{ implies } M,u \models \varphi \\
& \Longleftrightarrow & M,u \models \varphi \text{ for any world } u \text{ such that } G \in E_\chi(w,u)
\end{array}$$

$$\begin{array}{rcl}
\text{and so}\quad M,w \models C_G\varphi & \Longleftrightarrow & M,w \models E^k_G\varphi \text{ for all } k \in \mathbb{N}^+ \\
& \Longleftrightarrow & M,u \models \varphi \text{ for any world } u \text{ such that } G \in E^+_\chi(w,u) \qquad (*)
\end{array}$$

where $(*)$ can be shown as follows: Suppose $M, w \not\models E_G^n \varphi$ for some $n \in \mathbb{N}^+$, then by induction on n, we have $w_1, \ldots, w_n \in W$ such that $M, w_n \not\models \varphi$ and $G \in E_\chi(w, w_1) \cap \bigcap_{1 \le i < n} E_\chi(w_i, w_{i+1})$. Hence $M, w_n \not\models \varphi$ and $G \in E_\chi^+(w, w_n)$. Suppose $M, u \not\models \varphi$ for a world u such that $G \in E_\chi^+(w, u)$, w.l.o.g, assume that there exist $w_0, \ldots, w_n \in W$ such that $w_0 = w$, $w_n = u$, $G \in \bigcap_{0 \le i < n} E_\chi(w_i, w_{i+1})$ and $M, w_n \not\models \varphi$. Thus using the above result n times we have $M, w \not\models E_G^n \varphi$. □

Lemma 8. *The model checking problem for* L_{CDEF} *(hence for all of its sublogics) is in P.*

Proof. It suffices to provide a polynomial algorithm for the types of formulas $C_G\psi$, $D_G\psi$, $E_G\psi$ and $F_G\psi$. The details are given in Algorithm 2. As in the proof of Lemma 4, checking $C(a) \subseteq E(t, u)$ costs at most

Algorithm 2 Function $Val(M, \varphi)$ extended: cases with group knowledge operators

1: **Initialize:** $temVal \leftarrow \emptyset$
2: **if** ... **then** ... ▹ *Same as in Algorithm 1*
3: **else if** $\varphi = C_G\psi$ **then**
4: **for all** $x \in W$ **do**
5: **Initialize:** $n \leftarrow$ **true**
6: **for all** $y \in W$ **do**
7: **if** $G \in E_\varphi^+(x, y)$ **and** $y \notin Val(M, \psi)$ **then**
8: $n \leftarrow$ **false**
9: **if** $n =$ **true then**
10: $tmpVal \leftarrow tmpVal \cup \{x\}$
11: **return** $tmpVal$ ▹ *Returns* $\{x \in W \mid \forall y \in W : G \in E_\varphi^+(x, y) \Rightarrow y \in Val(M, \psi)\}$
12: **else if** $\varphi = D_G\psi$ **then**
13: **for all** $x \in W$ **do**
14: **Initialize:** $n \leftarrow$ **true**
15: **for all** $y \in W$ **do**
16: **if** $\bigcup_{a \in G} C(a) \subseteq E(x, y)$ **and** $y \notin Val(M, \psi)$ **then**
17: $n \leftarrow$ **false**
18: **if** $n =$ **true then**
19: $tmpVal \leftarrow tmpVal \cup \{x\}$
20: **return** $tmpVal$ ▹ *Returns* $\{x \in W \mid \forall y \in W : \bigcup_{a \in G} C(a) \subseteq E(x, y) \Rightarrow y \in Val(M, \psi)\}$
21: **else if** $\varphi = E_G\psi$ **then**
22: **for all** $x \in W$ **do**
23: **initialize** $n \leftarrow$ **true**
24: **for all** $y \in W$ **do**
25: **if** $G \in E_\varphi(x, y)$ **and** $y \notin Val(M, \psi)$ **then**
26: $n \leftarrow$ **false**
27: **if** $n =$ **true then** $tmpVal \leftarrow tmpVal \cup \{x\}$
28: **return** $tmpVal$ ▹ *Returns* $\{t \in W \mid \forall u \in W : G \in E_\varphi(t, u) \Rightarrow u \in Val(M, \psi)\}$
29: **else if** $\varphi = F_G\psi$ **then**
30: **for all** $x \in W$ **do**
31: **Initialize:** $n \leftarrow$ **true**
32: **for all** $y \in W$ **do**
33: **if** $\bigcap_{a \in G} C(a) \subseteq E(x, y)$ **and** $y \notin Val(M, \psi)$ **then**
34: $n \leftarrow$ **false**
35: **if** $n =$ **true then** $tmpVal \leftarrow tmpVal \cup \{x\}$
36: **return** $tmpVal$ ▹ *Returns* $\{x \in W \mid \forall y \in W : \bigcap_{a \in G} C(a) \subseteq E(x, y) \Rightarrow y \in Val(M, \psi)\}$

$|C| \cdot |E|$ steps, here we furthermore need to calculate the cost caused by group knowledge operators.

For D_G and F_G, notice that the number of agents in any group G that appears in φ is less than $|\varphi|$, so checking $\bigcup_{a \in G} C(a) \subseteq E(t, u)$ and $\bigcap_{a \in G} C(a) \subseteq E(t, u)$ costs at most $|C| \cdot |E| \cdot |\varphi|$ steps. Thus for the logics extended with these operators, the complexity for model checking would not go beyond P.

For E_G and C_G, we need to ensure that there is a polynomial-time algorithm for computing $E_\varphi(w, u)$ and $E_\varphi^+(w, u)$ and checking whether G is an element of them. By Definition 5 and Lemma 7, computing the set A_φ costs at most $|\varphi|$ steps, since there are at most $|\varphi|$ modalities appearing in φ; moreover, the size of G is at most $|\varphi|$. To compute $E_\varphi(w, u)$ for any given w and u, it costs at most $|E|$ steps to compute $E(w, u)$ and at most $|\varphi|^2 \cdot |C| \cdot |E|$ steps to check for every $G \in A_\varphi$ whether there exists $a \in G$ such that $C(a) \subseteq E(w, u)$. So the cost of computing the whole function E_φ can be finished in at most $|W|^2 \cdot (|E| + |\varphi|^2 \cdot |C| \cdot |E|)$ steps. Now we consider the computation of E_φ^+. Assume that we have a string that describes E_φ, then we check for all pairs $(x, y), (y, z) \in W^2$ whether there exists a "G" appearing in φ such that $G \in E_\varphi(x, y) \cap E_\varphi(y, z)$; if it is, we add G as a member of $E_\varphi(x, z)$. Keep doing this until E_φ does not change any more. Every round of checking takes at most $2|\varphi|^2 \cdot |W|^3$ steps, and it will be stable

in at most $|\varphi| \cdot |W|^2$ rounds. Then we obtain the function E^+_φ as we want. Every membership checking for $G \in E^+_\varphi(w, v)$ is finished in polynomial steps. So the whole process is still in P. □

3.2.3 Model checking formulas with update modalities

As we address the case involving update modalities, let us consider a model $M = (W, E, C, \beta)$ and a world $w \in W$, and examine the formulas $(+_S)_a\psi$, $(-_S)_a\psi$, $(=_S)_a\psi$ and $(\equiv_b)_a\psi$. According to the semantics provided in Definition 2,

$$M, w \models (+_S)_a\psi \iff M^{a+S}, w \models \psi$$

where $M^{a+S} = (W, E, C^{a+S}, \beta)$ is defined such that

$$C^{a+S}(x) = \begin{cases} C(x), & \text{if } x \neq a \\ C(a) \cup S, & \text{if } x = a \end{cases}$$

From this, we deduce that verifying whether $M, w \models (+_S)_a\psi$ is reducible to checking if $M^{a+S}, w \models \psi$, effectively eliminating the leftmost update modality from consideration. An algorithm that invokes the model checking algorithm on the latter can be executed in linear time since it involves generating the updated model M^{a+S} directly from the original model M and considering the new formula ψ, which is a substring of the original formula. Hence, the complete algorithm, including the invocation of the model checking, will conclude within polynomial time.

The cases with $(-_S)_a\psi$, $(=_S)_a\psi$ and $(\equiv_b)_a\psi$ follow a similar process, with the distinction that each involves a different modification to the model. Nonetheless, the computational cost remains within P for both scenarios. This leads us to the following theorem:

Theorem 9. *The model checking problems for* $\mathrm{L}_{CDEF+-=\equiv}$ *and all of its sublogics are in P.*

3.3 Model checking quantified formulas: PSPACE complete

We demonstrate the PSPACE hardness by reducing, in polynomial time, the problem of undirected edge geography (UEG) – a variant of the generalized geography [32, 25] – to the model checking problem for any of $\mathrm{L}_{\boxplus}$, $\mathrm{L}_{\boxminus}$ or $\mathrm{L}_{\Box}$, since UEG is a game for which determining a winning strategy is known to be PSPACE complete [17]. The PSPACE upper bound is established using a polynomial space algorithm that builds upon the algorithms introduced earlier.

Let $G = (D, R)$ be an undirected graph; i.e., D is a finite nonempty set, and R is a symmetric and irreflexive relation on D. Given a node $d \in D$, the pair (G, d) is referred to as a *rooted undirected graph*. The undirected edge geography (UEG) game on (G, d) involves two players, and unfolds as follows.

1. Player I's Move: Player I starts by selecting edge $\{d, d_1\} \in R$. If no such edge exists, the game ends and Player II wins as Player I cannot make a valid move.
2. Player II's Move: After Player I"s move selecting an edge $\{d_i, d_{i+1}\}$, Player II must choose an edge $\{d_{i+1}, d_{i+2}\}$ that has not been chosen in previous moves. If Player II cannot make such a move, the game ends and Player I wins.
3. Alternating Turns: After Player II's move selecting an edge $\{d_j, d_{j+1}\}$, it is Player I's turn again to choose an edge $\{d_{j+1}, d_{j+2}\}$ not previously chosen. If Player I cannot make such a move, the game ends and Player II wins.
4. Repeat Step 2: The game continues by alternating turns following the process described in step 2.

Alternatively, UEG game on (G,d) can be recursively defined by modifying the graph after each move:

- The current player chooses an edge $\{d,d'\} \in R$; if this is impossible, he loses, and the game ends.
- The game then proceeds with the opposing player starting a new game on (G',d') where $G' = (D, R \setminus \{\{d,d'\}\})$.

The *UEG problem*, based on a rooted undirected graph, aims to determine whether Player I possesses a winning strategy.

Definition 10 (induced model)**.** *Let $G = (D,R)$ represent an undirected graph. For each edge $\{x,y\} \in R$, assign a unique epistemic skill $s_{\{x,y\}} \in \mathsf{S}$ (ensuring that $s_{\{x',y'\}} \neq s_{\{x'',y''\}}$ for any distinct unordered pairs $\{x',y'\}$ and $\{x'',y''\}$), and for each node $x \in D$, assign a unique atomic proposition $p_x \in \mathsf{P}$ (ensuring that $p_{x'} \neq p_{x''}$ for any distinct nodes x' and x'').*

Define the induced model M_G *as the tuple (D,E,C,β) where:*

- *E: For every $x,y \in D$, if $\{x,y\} \in R$, then $E(x,y) = \{s_{\{x,y\}}\}$; otherwise, $E(x,y) = \emptyset$;*
- *C: For all agents a, $C(a) = \emptyset$;*
- *β: For each node $x \in D$, $\beta(x) = \{p_x\}$.*

This model M_G is well-defined and compactly represents the relationships and properties within the graph G. The size of E is $O(|D|^2)$ due to the pairwise relationship between nodes, while the size of β is $O(|D|)$, reflecting the unique property assignment per node. The size of C remains $O(|D|)$, given that only a limited number of agents are actually be utilized, as confirmed by the following definition and the definition of the size of the input.

Definition 11 (induced formula)**.** *Let $G = (D,R)$ be an undirected graph. Consider n agents $a_1,\dots,a_n \in \mathsf{A}$, where n is the smallest positive even number greater than or equal to $|R|$. For each i where $1 \leq i \leq n$ (for φ_i, only consider even numbers i), define:*

$$
\begin{aligned}
\psi_i &:= \neg K_{a_i}\bot \wedge \textstyle\bigvee_{x\in D} K_{a_i} p_x \\
\chi_i &:= \textstyle\bigvee_{x,y\in D \text{ with } x\neq y, 1\leq j<i} (p_x \wedge \hat{K}_{a_j} p_y \wedge K_{a_i} p_y) \\
\varphi_i &:= \quad \boxplus\!\!\!\!\diamond_{a_1}(\psi_1 \wedge \neg\chi_1 \wedge K_{a_1} \boxplus_{a_2} (\neg\psi_2 \vee \chi_2 \vee \\
&\qquad\quad \hat{K}_{a_2} \Diamond\!\!\!\!+_{a_3}(\psi_3 \wedge \neg\chi_3 \wedge K_{a_3} \boxplus_{a_4} (\neg\psi_4 \vee \chi_4 \vee \\
&\qquad\qquad \hat{K}_{a_4} \Diamond\!\!\!\!+_{a_5}(\psi_5 \wedge \neg\chi_5 \wedge K_{a_5} \boxplus_{a_6} (\neg\psi_6 \vee \chi_6 \vee \\
&\qquad\qquad\quad \cdots \\
&\qquad\qquad\qquad \hat{K}_{a_{i-2}} \Diamond\!\!\!\!+_{a_{i-1}}(\psi_{i-1} \wedge \neg\chi_{i-1} \wedge K_{a_{i-1}} \boxplus_{a_i} (\neg\psi_i \vee \chi_i))\cdots)))))).
\end{aligned}
$$

In the above, $\hat{K}_a$ is the dual of K_a. The induced formula φ_G *for the graph G is defined as φ_n.*

Let us try to understand the induced formula. In a game, each agent a_i plays in the i-th move. The formulas ψ_i represents the condition where the player a_i at the i-th move chooses exactly one edge from the current node. The formula χ_i captures the scenario where the edge chosen by player a_i at the i-th move has been selected in a previous move, thus representing an invalid game move under the new edge rule. The conjunction $\psi_i \wedge \neg\chi_i$ ensures that each move in the game involves selecting a new, unvisited edge. As for complexity, the length of ψ_i is in $O(|D|)$, as it involves a disjunction over each node in D. The length of χ_i is in $O(|D|^2 \cdot |R|)$. The overall formula φ_G thus has its length in $O(|D|^2 \cdot |R|^2)$.

The formula φ_G constructs a logical framework that mirrors the gameplay in an undirected graph:

- $\Diamond\!\!\!\!+_{a_1}$: Indicates the potential for player a_1 to make a valid move by upskilling.
- $\psi_1 \wedge \neg\chi_1$: Ensures that a_1's choice is a new edge (valid move).

- $K_{a_1}\boxplus_{a_2}$: Player a_1 ensures that no matter how player a_2 responds (upskills), the game's next state must be described by the formula that follows. And that formula describes that either a_2 does not find a new edge to choose (leading to the end the game), or, if a_2 chooses a new edge, then the formula starting with $\hat{K}_{a_3}\oplus_{a_3}$ must hold, indicating a situation similar to the first clause above (but for a_3).

This recursive and intertwined structure of φ_G effectively captures the strategic progression of the game, with each player's move affecting the possible moves of the next player, all within the framework of an undirected graph where each node represents a game state or choice.

We now introduce a lemma that establishes a connection between the undirected edge geography problem and the logics we have developed.

Lemma 12. *For any rooted undirected graph (G,d), Player I has a wining strategy in the undirected edge geography game on (G,d), if and only if $M_G,d \models \varphi_G$.*

Proof. We show the lemma by induction on $|R|$. Base case $|R| = 0$, $n = 2$. For any $x,y \in D$, $\{x,y\} \notin R$. Player I loses in this case. Let $M_G = (D,E,C,\beta)$ be the induced model. Then $E(x,y) = \emptyset$ for any $x,y \in D$. We need to show $M_G,d \not\models \varphi_G$. For any finite non-empty $S \subseteq \mathsf{S}$, consider the model $M' = (D,E,C^{a_1+S},\beta)$. Since $\varphi_G = \varphi_2 = \oplus_{a_1}(\psi_1 \wedge \neg\chi_1 \wedge K_{a_1}\boxplus_{a_2}(\neg\psi_2 \vee \chi_2))$, where $\psi_1 = \neg K_{a_1}\bot \wedge \bigvee_{x\in D} K_{a_1}p_x$, $\chi_1 = \bot$, $\psi_2 = \neg K_{a_2}\bot \wedge \bigvee_{x\in D} K_{a_2}p_x$, and $\chi_2 = \bigvee_{x\neq y\in D}(p_x \wedge \hat{K}_{a_1}p_y \wedge K_{a_2}p_y)$. It is clear that $M',d \not\models \psi_1$, since $M',d \models K_{a_1}\bot$. It follows that $M',d \not\models \psi_1 \wedge \neg\chi_1 \wedge K_{a_1}\boxplus_{a_2}(\neg\psi_2 \vee \chi_2)$. Since S is arbitrary, we have $M_G,d \not\models \varphi_G$.

Base case $|R| = 1$, and so $n = 2$. Let $\{d,d'\}$ be the unique edge in R. Let $M_G = (D,E,C,\beta)$ be the induced model. $E(d,d') = E(d',d) = \{s_{\{d,d'\}}\}$, and $E(x,y) = \emptyset$ otherwise. Player I has a winning strategy in this case, and we show that $M_G,d \models \varphi_G$. Consider $S = \{s_{\{d,d'\}}\}$. Let $M' = (D,E,C^{a_1+S},\beta)$, with $\varphi_G = \varphi_2 = \oplus_{a_1}(\psi_1 \wedge \neg\chi_1 \wedge K_{a_1}\boxplus_{a_2}(\neg\psi_2 \vee \chi_2))$, where:

- $\psi_1 = \neg K_{a_1}\bot \wedge \bigvee_{x\in D} K_{a_1}p_x$ ($M',d \models \psi_1$, for $M',d \models \neg K_{a_1}\bot \wedge K_{a_1}p_{d'}$)
- $\chi_1 = \bot$ ($M',d \models \neg\chi_1$)
- $\psi_2 = \neg K_{a_2}\bot \wedge \bigvee_{x\in D} K_{a_2}p_x$
- $\chi_2 = (p_d \wedge \hat{K}_{a_1}p_{d'} \wedge K_{a_2}p_{d'}) \vee (p_{d'} \wedge \hat{K}_{a_1}p_d \wedge K_{a_2}p_d) \vee \bigvee_{x\neq y\in D\setminus\{d,d'\}}(p_x \wedge \hat{K}_{a_1}p_y \wedge K_{a_2}p_y)$.

For any finite nonempty $S' \subseteq \mathsf{S}$, let $M'' = (D,E,(C^{a_1+S})^{a_2+S'},\beta)$, we have one of the following cases:

(1) $S' \not\subseteq S$, then $\forall x \in D$, $(C^{a_1+S})^{a_2+S'}(a_2) \not\subseteq E(d,x)$, hence $M'',d' \models \neg\psi_2$, for $M'',d' \models K_{a_2}\bot$.

(2) $S' \subseteq S$, then $M'',d' \models p_{d'} \wedge \hat{K}_{a_1}p_d \wedge K_{a_2}p_d$. Thus, $M'',d' \models \chi_2$ for its right disjunct is satisfied.

In both case $M'',d' \models \neg\psi_2 \vee \chi_2$, and so $M',d' \models \boxplus_{a_2}(\neg\psi_2 \vee \chi_2)$, and $M',d \models K_{a_2}\boxplus_{a_2}(\neg\psi_2 \vee \chi_2)$. Together with the verifications above, we have $M_G,d \models \varphi_G$.

The case $|R| = k$. The direction from left to right. Suppose that Player I has a winning strategy, by which she chooses in the first move $\{d,d'\}$. Let $M_G = (D,E,C,\beta)$ be the induced model. We need to show that $M_G,d \models \varphi_G$, where $\varphi_G = \oplus_{a_1}(\psi_1 \wedge \neg\chi_1 \wedge K_{a_1}\varphi_{G,\boxplus_{a_2}})$, in which $\varphi_{G,\boxplus_{a_2}}$ is the subformula of φ_G beginning with $\boxplus_{a_2}$ (see Def. 11). Consider $S = \{s_{\{d,d'\}}\}$, and let $M' = (D,E,C^{a_1+S},\beta)$:

- $\psi_1 = \neg K_{a_1}\bot \wedge \bigvee_{x\in D} K_{a_1}p_x$ ($M',d \models \psi_1$, for $M',d \models \neg K_{a_1}\bot \wedge K_{a_1}p_{d'}$)
- $\chi_1 = \bot$ ($M',d \models \neg\chi_1$)

Now we show $M', d \models K_{a_1}\varphi_{G,\boxplus_{a_2}}$; namely, $M', d' \models \varphi_{G,\boxplus_{a_2}}$, where $\varphi_{G,\boxplus_{a_2}} = \boxplus_{a_2}(\neg\psi_2 \vee \chi_2 \vee \hat{K}_{a_2}\varphi_{G,\oplus_{a_3}})$ in which $\varphi_{G,\oplus_{a_3}}$ is the subformula of φ_G beginning wtih $\oplus_{a_3}$. For any finite nonempty $S' \subseteq \mathsf{S}$, let $M'' = (D, E, (C^{a_1+S})^{a_2+S'}, \beta)$, and it suffices to show that

$$M'', d' \models \neg\psi_2 \vee \chi_2 \vee \hat{K}_{a_2}\varphi_{G,\oplus_{a_3}}, \tag{†}$$

where $\psi_2 = \neg K_{a_2}\bot \wedge \bigvee_{x\in D} K_{a_2} p_x$ and $\chi_2 = \bigvee_{x\neq y\in D}(p_x \wedge \hat{K}_{a_1} p_y \wedge K_{a_2} p_y)$. Consider the possible cases:

(1) There does not exist $x \in D$ such that $S' \subseteq E(d', x)$, or

(2) There exists $d'' \in D$ such that $S' \subseteq E(d', d'')$ (note that S' must be singleton).

In case (1), $M'', d' \models K_{a_2}\bot$, so $M'', d' \models \neg\psi_2$, hence (†) holds. In case (2), Player I has a winning strategy in the continued game on (G_2, d'') with $G_2 = (D, R \setminus \{\{d, d'\}, \{d', d''\}\})$ (note that d'' cannot be d or d'). It suffices to show the following result:

$$M'', d'' \models \varphi_{G,\oplus_{a_3}} \iff M_{G_2}, d'' \models \varphi_{G_2}. \tag{‡}$$

Since $M_{G_2}, d'' \models \varphi_{G_2}$ holds by the induction hypothesis, by (‡), we have $M'', d'' \models \varphi_{G,\oplus_{a_3}}$. This makes the rightmost disjunct of (†) true in M'', d', and completes the whole proof.

Let $M_{G_2} = (D, E_2, C, \beta)$. To see (‡), $M'', d'' \models \varphi_{G,\oplus_{a_3}}$, i.e., $(D, E, (C^{a_1+S})^{a_2+S'}, \beta), d'' \models \varphi_{G,\oplus_{a_3}}$

$\iff$ $(D, E_2, (C^{a_1+S})^{a_2+S'}, \beta), d'' \models \varphi'_{G,\oplus_{a_3}}$, where $\varphi'_{G,\oplus_{a_3}}$ is adapted from $\varphi_{G,\oplus_{a_3}}$ by the following:

- Delete all occurrences of $\bigvee_{x\neq y\in D}(p_x \wedge \hat{K}_{a_1} p_y \wedge K_{a_i} p_y)$ from $\varphi_{G,\oplus_{a_3}}$
- Delete all occurrences of $\bigvee_{x\neq y\in D}(p_x \wedge \hat{K}_{a_2} p_y \wedge K_{a_i} p_y)$ from $\varphi_{G,\oplus_{a_3}}$

(This equivalence holds since $E(d, d') = E(d', d) = \emptyset$, which implies that any formulas $\hat{K}_{a_1}\varphi$ and $\hat{K}_{a_2}\varphi$ are false in any world x of model (D, E_2, C', β), where C' is any capability function updated from $(C^{a_1+S})^{a_2+S'}$ without changing the capabilities of a_1 and a_2.)

$\iff$ $(D, E_2, C, \beta), d'' \models \varphi''_{G,\oplus_{a_3}}$, where $\varphi''_{G,\oplus_{a_3}}$ a variant of $\varphi'_{G,\oplus_{a_3}}$ by replacing any a_{i+2} with a_i,
(This holds since $(C^{a_1+S})^{a_2+S_2}(a_{i+2}) = C(a_i) = \emptyset$; note that a_1 and a_2 does not exist in $\varphi'_{G,\oplus_{a_3}}$.)

$\iff$ $M_{G_2}, d'' \models \varphi_{G_2}$, i.e., $(D, E_2, C, \beta), d'' \models \varphi_{G_2}$ (since $\varphi_{G_2} = \varphi''_{G,\oplus_{a_3}}$)

From right to left. If Player I does not have a winning strategy, we must show that $M_G, d \not\models \varphi_G$. Let the induced model M_G be (D, E, C, β). Since Player I does not have a winning strategy, then:

(a) There is no $x \in D$ such that $\{d, x\} \in R$, and Player I loses in this case; or

(b) Player I does not have a winning strategy by choosing in the first move any $x \in D \setminus \{d\}$ such that $\{d, x\} \in R$.

For case (a): Since $E(d, x) = \emptyset$ for any x, we get $M_G, d \not\models \varphi_G$ similarly to the case when $|R| = 0$.
For case (b): Consider an arbitrary finite nonempty $S \subseteq \mathsf{S}$. Then:

(1) For all $x \in D$, $S \not\subseteq E(d, x)$; or

(2) Theres exists $d' \in D$ such that $S \subseteq E(d, d')$ (note that d' cannot be d).

We need to show $M_G, d \not\models \varphi_G$ where φ_G is given in Def. 11. Let $M' = (D, E, C^{a_1+S}, \beta)$. In case (1), since $M', d \models K_{a_1}\bot$, $M', d \not\models \psi_1$ (with $\psi_1 = \neg K_{a_1}\bot \wedge \bigvee_{x\in D} K_{a_1} p_x$), and so $M, d \not\models \varphi_G$.

In case (2) (under the case (b)), there must exist $d'' \in D \setminus \{d, d'\}$ such that Player I does not have a winning strategy in the game on (G_2, d'') where $G_2 = (D, R \setminus \{\{d, d'\}, \{d', d''\}\})$; for otherwise Player I

has a winning strategy (this is also the case when there is no such a d''), leading to a contradiction. Let $S' = \{s_{\{d',d''\}}\}$, then $S' \subseteq E(d',d'')$. Let $M'' = (D,E,(C^{a_1+S})^{a_2+S'},\beta)$. It suffices to show that

$$M'',d' \not\models \neg\psi_2 \vee \chi_2 \vee \hat{K}_{a_2}\varphi_{G,\oplus_{a_3}}, \tag{*}$$

Consider $\psi_2 = \neg K_{a_2}\bot \wedge \bigvee_{x\in D} K_{a_2}p_x$. Since $M'',d' \models \neg K_{a_2}\bot \wedge K_{a_2}p_{d''}$, we have $M'',d' \not\models \neg\psi_2$. As for $\chi_2 = \bigvee_{x\neq y\in D}(p_x \wedge \hat{K}_{a_1}p_y \wedge K_{a_2}p_y)$, since $M'',d' \models \hat{K}_{a_1}p_y \wedge K_{a_2}p_y$ implies $y = d \neq d'' = y$, we have $M'',d' \not\models \chi_2$. Finally we show that $M'',d' \not\models \hat{K}_{a_2}\varphi_{G,\oplus_{a_3}}$. Since there is exact one $x \in D$ (which must be d'' by the definition of S') such that $S' \subseteq E(d',x)$, it suffices to prove $M'',d'' \not\models \varphi_{G,\oplus_{a_3}}$. Note that (‡) from the proof of the converse direction can also be shown here, it suffices to show that $M_{G_2},d'' \not\models \varphi_{G_2}$, and this holds by the induction hypothesis. □

Corollary 13. *Undirected edge geography is polynomial time reducible to the model checking problem for* $\mathrm{L}_{\boxplus}$.

Remark 2. *It is important to note that the reduction discussed previously utilizes only the modalities* $\boxplus$ *and* $\oplus$. *However, we can also perform a reduction using exclusively the modalities* $\Box$ *and* $\Diamond$. *This alternative reduction is structurally similar to the original, with the primary modification being the replacement of* $\boxplus$ *with* $\Box$. *Additionally, a reduction that employs only the modalities* $\boxminus$ *and* $\ominus$ *is also feasible. In this case, we replace* $\boxplus$ *with* $\boxminus$. *Furthermore, there is a requirement to modify the skill set* $C(a_i)$ *to* $\{s_{\{w,v\}} \mid w,v \in D\}$. *The model checking problems for any logics that include at least one of the modalities –* $\boxplus$, $\boxminus$, $\Box$, $\oplus$, $\ominus$ *or* $\Diamond$ *– remain PSPACE hard. This complexity assertion holds even in the absence of additional modalities such as* C_G, D_G, E_G, F_G, $(+_S)_a$, $(-_S)_a$, $(=_S)_a$, *and* $(\equiv_S)_a$.

Lemma 14. *The model checking problem for* $\mathrm{L}_{CDEF+-=\equiv\boxplus\boxminus\Box}$ *is in PSPACE.*

Proof. With the presence of Algorithm 2, it suffices to provide a polynomial space algorithm for the types of formulas $\boxplus_a\varphi$, $\Box_a\varphi$ and $\boxminus_a\varphi$. The details are given in Algorithm 3.

Algorithm 3 Function $Val((W,E,C,\beta),\varphi)$ extended: cases with quantifiers

```
1:  Initialize: temVal ← ∅
2:  Initialize: S1 ← (⋃_{w,v∈W} E(w,v)) ∪
      (⋃_{a appears in φ} C(a))
3:  Initialize: S2 ← S1 ∪ {s}          ▷ Here s ∈ 𝕊 is new for S1
4:  if ... then ...                      ▷ Same as in Algorithm 2
5:  else if φ = ⊞_a ψ then
6:     for all t ∈ W do
7:        Initialize: n ← true
8:        for all S ⊆ S2 do
9:           if S ≠ ∅ and t ∉ Val((W,E,C^{a+S},β),ψ)
               then n ← false
10:       if n = true then tmpVal ← tmpVal ∪ {t}
11:    return tmpVal     ▷ Returns {t ∈ W | ∀S ⊆ S1 :
         t ∈ Val((W,E,C^{a+S},β),ψ)}
12: else if φ = ⊟_a ψ then
13:    for all t ∈ W do
14:       Initialize: n ← true
15:       for all S ⊆ S2 do
16:          if S ≠ ∅ and t ∉ Val((W,E,C^{a−S},β),ψ)
               then n ← false
17:       if n = true then tmpVal ← tmpVal ∪ {t}
18:    return tmpVal     ▷ Returns {t ∈ W | ∀S ⊆ S1 :
         t ∈ Val((W,E,C^{a−S},β),ψ)}
19: else if φ = □_a ψ then
20:    for all t ∈ W do
21:       Initialize: n ← true
22:       for all S ⊆ S2 do
23:          if S ≠ ∅ and t ∉ Val((W,E,C^{a=S},β),ψ)
               then n ← false
24:       if n = true then tmpVal ← tmpVal ∪ {t}
25:    return tmpVal     ▷ Returns {t ∈ W | ∀S ⊆ S1 :
         t ∈ Val((W,E,C^{a=S},β),ψ)}
```

Here we furthermore need to check the space cost caused by the new modalities. But notice that all the space cost of checking $Val((W,E,C,\beta),\varphi)$ is in $O(|M|)$. So the space cost of the algorithm is immediately linear. So the model checking problem for $\mathrm{L}_{CDEF+-=\equiv\boxplus\boxminus\Box}$ is in PSPACE. □

We reach the following result from Corollary 13 (considering that UEG is PSPACE complete) and Lemma 14.

Theorem 15. *The model checking problems for all logics with quantifiers (i.e., at least one of* $\boxplus$*,* $\boxminus$ *and* $\Box$*) that extends the base logic* L *is PSPACE complete.*

4 Discussion

We have developed a variety of logics that incorporate individual and group knowledge, actions such as knowing, forgetting, revising, and learning, as well as the necessity and possibility of these actions. These logics are highly expressive, yet the computational cost for model checking remains manageable. Specifically:

- For logics devoid of quantifiers, the complexity of model checking falls within the class P, aligning with many traditional epistemic logics.
- For logics that include quantifiers, the complexity is PSPACE complete. This matches the complexity found in similar types of logics, such as Group Announcement Logic [2], Coalition Announcement Logic [29, 19, 4], and Subset Space Arbitrary Announcement Logic [6]. [3]

Logicians are deeply interested in the decidability of validity/satisfiability problems for logics that include quantifiers over updates. Known complexities, such as the undecidability of Arbitrary Public Announcement Logic (APAL) and Group Announcement Logic [18, 1], have spurred ongoing research into decidable alternatives [18, 11, 10]. Even the development of variants that are recursively axiomatizable represents an advancement [35, 8], especially given that APAL is not likely to have this feature.

Our research also aims to explore the decidability and computational complexity of satisfiability and validity problems within our logics. Although our ongoing efforts have yielded PSPACE completeness and EXPTIME completeness results for many of our less complex logics (for example, the satisfiability problems for logics devoid of common knowledge, update modalities, and quantifiers are PSPACE complete, whereas those lacking update modalities and quantifiers but incorporating common knowledge are EXPTIME complete), a definitive result for the full logic $\mathrm{L}_{CDEF+-=\equiv\boxplus\boxminus\Box}$ remains elusive. Additionally, while we have successfully axiomatized some of our logics in previous studies [24], an axiomatic system for the full logic is not yet developed. These areas are designated for future exploration and development.

We have introduced a new update modality for learning, $(\equiv_b)_a$, which denotes the action where agent a learns the skills of agent b. This operator essentially replaces a's skill set with that of b. However, we can also devise variants that facilitate skill set modification through incremental learning (e.g., $(\cup_b)_a$) or decremental learning (e.g., $(\cap_b)_a$ "retaining only beneficial skills from b", or $(\backslash_b)_a$ "eliminating undesirable skills of b"). Additionally, the concept of "deskilling," derived from the richness of natural language, refers to a reduction in the skills required to perform a task. This could be modeled as an update action that modifies the edge function, thereby requiring fewer skills to distinguish between worlds, potentially leading to knowledge acquisition. Incorporating these diverse learning modalities does not increase the complexity of the model checking problem, though it may add complexity to the validity problem. The exploration of quantifiers over learning operators presents another intriguing area of study.

Acknowledgements We express our gratitude to the anonymous reviewers for their invaluable comments and suggestions. We acknowledge the financial support by the MOE Project of Humanities and Social Sciences (No. 24YJA72040002) and the National Social Science Fund of China (Grant No. 20&ZD047).

[3] It is worth noting that model checking in Arbitrary Public Announcement Logic is also believed to be PSPACE complete [5]. However, a detailed validation of this claim has not yet been found by us.

References

[1] T. Ågotnes, H. van Ditmarsch & T. French (2016): *The Undecidability of Quantified Announcements*. *Studia Logica* 104(4), pp. 597–640, doi:10.1007/s11225-016-9657-0.

[2] Thomas Ågotnes, Philippe Balbiani, Hans van Ditmarsch & Pablo Seban (2010): *Group Announcement Logic*. *Journal of Applied Logic* 8(1), pp. 62–81, doi:10.1016/j.jal.2008.12.002.

[3] Carlos E. Alchourrón, Peter Gärdenfors & David Makinson (1985): *On the Logic of Theory Change: Partial Meet Contraction and Revision Functions*. *The Journal of Symbolic Logic* 50, pp. 510–530, doi:10.2307/2274239.

[4] Natasha Alechina, Hans van Ditmarsch, Rustam Galimullin & Tuo Wang (2021): *Verification and Strategy Synthesis for Coalition Announcement Logic*. *Journal of Logic, Language and Information* 30(4), pp. 671–700, doi:10.1007/s10849-021-09339-6.

[5] Philippe Balbiani, Alexandru Baltag, Hans van Ditmarsch, Andreas Herzig, Tomohiro Hoshi & Tiago de Lima (2008): *'Knowable' as 'Known after an Announcement'*. *The Review of Symbolic Logic* 1(3), pp. 305–334, doi:10.1017/S1755020308080210.

[6] Philippe Balbiani, Hans van Ditmarsch & Andrey Kudinov (2013): *Subset Space Logic with Arbitrary Announcements*. In: *Proceedings of ICLA* 2013, pp. 233–244, doi:10.1007/978-3-642-36039-8_21.

[7] Alexandru Baltag, Lawrence S. Moss & SLawomir Solecki (1998): *The Logic of Public Announcements, Common Knowledge, and Private Suspicions*. In I. Gilboa, editor: *Proceedings of the 7th Conference on Theoretical Aspects of Rationality and Knowledge* (*TARK* 98), pp. 43–56, doi:10.5555/645876.671885.

[8] Alexandru Baltag, Aybüke Özgün & Ana Lucia Vargas Sandoval (2023): *Arbitrary Public Announcement Logic with Memory*. *Journal of Philosophical Logic* 52(1), pp. 53–110, doi:10.1007/s10992-022-09664-6.

[9] Alexandru Baltag, Aybüke Özgün & Ana Lucia Vargas Sandoval (2017): *Topo-Logic as a Dynamic-Epistemic Logic*. In Alexandru Baltag, Jeremy Seligman & Tomoyuki Yamada, editors: *Logic, Rationality, and Interaction*, Springer Berlin Heidelberg, pp. 330–346, doi:10.1007/978-3-662-55665-8_23.

[10] Hans van Ditmarsch & Tim French (2022): *Quantifying over Boolean announcements*. *Logical Methods in Computer Science* 18(1), doi:10.46298/lmcs-18(1:20)2022.

[11] Hans van Ditmarsch, Tim French & Sophie Pinchinat (2010): *Future Event Logic – Axioms and Complexity*. In Lev D. Beklemishev, Valentin Goranko & Valentin B. Shehtman, editors: *Advances in Modal Logic* 8*, papers from the eighth conference on "Advances in Modal Logic," held in Moscow, Russia,* 24-27 *August* 2010, College Publications, pp. 77–99.

[12] Hans van Ditmarsch, Andreas Herzig, Jérôme Lang & Pierre Marquis (2009): *Introspective Forgetting*. *Synthese* 169(2), pp. 405–423, doi:10.1007/s11229-009-9554-4.

[13] Hans van Ditmarsch, Wiebe van der Hoek & Barteld Kooi (2008): *Dynamic Epistemic Logic*. *Synthese Library* 337, Springer Netherlands, doi:10.1007/978-1-4020-5839-4.

[14] Huimin Dong, Xu Li & Yì N. Wáng (2021): *Weighted Modal Logic in Epistemic and Deontic Contexts*. In Sujata Ghosh & Thomas Icard, editors: *Proceedings of the Eighth International Conference on Logic, Rationality and Interaction* (*LORI* 2021), *Lecture Notes of Theoretical Computer Science* 13039, Springer, pp. 73–87, doi:10.1007/978-3-030-88708-7_6.

[15] Ronald Fagin & Joseph Y. Halpern (1988): *Belief, Awareness, and Limited Reasoning*. *Artificial Intelligence* 34(1), pp. 39–76, doi:10.1016/0004-3702(87)90003-8.

[16] Ronald Fagin, Joseph Y. Halpern, Yoram Moses & Moshe Y. Vardi (1995): *Reasoning about Knowledge*. The MIT Press, doi:10.7551/mitpress/5803.001.0001.

[17] Aviezri S Fraenkel, Edward R Scheinerman & Daniel Ullman (1993): *Undirected Edge Geography*. *Theoretical Computer Science* 112(2), pp. 371–381, doi:10.1016/0304-3975(93)90026-P.

[18] Tim French & Hans van Ditmarsch (2008): *Undecidability for Arbitrary Public Announcement Logic*. In Carlos Areces & Robert Goldblatt, editors: *Advances in Modal Logic*, 7, College Publications, pp. 23–42.

[19] Rustam Galimullin, Natasha Alechina & Hans van Ditmarsch (2018): *Model Checking for Coalition Announcement Logic*. In Frank Trollmann & Anni-Yasmin Turhan, editors: *KI* 2018: *Advances in Artificial Intelligence*, Springer International Publishing, Cham, pp. 11–23, doi:10.1007/978-3-030-00111-7_2.

[20] Mikkel Hansen, Kim Guldstrand Larsen, Radu Mardare & Mathias Ruggaard Pedersen (2018): *Reasoning about Bounds in Weighted Transition Systems*. *Logical Methods in Computer Science* 14(4), pp. 1–32, doi:10.23638/LMCS-14(4:19)2018.

[21] Jaakko Hintikka (1962): *Knowledge and Belief: An Introduction to the Logic of Two Notions*. Cornell University Press, Ithaca, New York.

[22] Jôme Lang, Paolo Liberatore & Pierre Marquis (2003): *Propositional Independence: Formula-Variable Independence and Forgetting*. *Journal of Artificial Intelligence Research* 18(1), pp. 391–443, doi:10.5555/1622420.1622431.

[23] Kim G. Larsen & Radu Mardare (2014): *Complete Proof Systems for Weighted Modal Logic*. *Theoretical Computer Science* 546(12), pp. 164–175, doi:10.1016/j.tcs.2014.03.007.

[24] Xiaolong Liang & Yì N. Wáng (2022): *Epistemic Logic over Weighted Graphs*. In: *Proceedings of the Second International Workshop on Logics for New-Generation AI*, College Publications, pp. 43–58.

[25] David Lichtenstein & Michael Sipser (1980): *GO Is Polynomial-Space Hard*. *Journal of the ACM* 27(2), pp. 393–401, doi:10.1145/322186.322201.

[26] Fangzhen Lin & Ray Reiter (1994): *Forget It!* In: *Working Notes of AAAI Fall Symposium on Relevance*, pp. 154–159.

[27] John-Jules Ch. Meyer & Wiebe van der Hoek (1995): *Epistemic Logic for AI and Computer Science*. Cambridge University Press, doi:10.1017/CBO9780511569852.

[28] Pavel Naumov & Jia Tao (2015): *Logic of Confidence*. *Synthese* 192, pp. 1821–1838, doi:10.1007/s11229-014-0655-3.

[29] Marc Pauly (2002): *A Modal Logic for Coalition Power in Games*. *Journal of Logic Computation* 12(1), pp. 149–166, doi:10.1093/logcom/12.1.149.

[30] Jan A. Plaza (1989): *Logics of Public Communications*. In M. L. Emrich, M. S. Pfeifer, M. Hadzikadic & Z. W. Ras, editors: *Proceedings of the* 4*th International Symposium on Methodologies for Intelligent Systems* (*ISMIS* '89), Oak Ridge National Laboratory, pp. 201–216.

[31] W. V. Quine (1956): *Quantifiers and Propositional Attitudes*. *The Journal of Philosophy* 53(5), pp. 177–187, doi:10.2307/2022451.

[32] Thomas J. Schaefer (1978): *On the Complexity of Some Two-Person Perfect-Information Games*. *Journal of Computer and System Sciences* 16(2), pp. 185–225, doi:10.1016/0022-0000(78)90045-4.

[33] Yì N. Wáng & Thomas Ågotnes (2013): *Subset Space Public Announcement Logic*. In Kamal Lodaya, editor: *Proceedings of ICLA*, *Lecture Notes in Computer Science* 7750, Springer, pp. 245–257, doi:10.1007/978-3-642-36039-8_22.

[34] Georg H. von Wright (1951): *An Essay in Modal Logic*. Studies in Logic and the Foundations of Mathematics, North-Holland Publishing Company.

[35] Kang Xu & Yì N. Wáng (2018): *Group Simple Announcement Logic*. *Studies in Logic* 11(1), pp. 1–22.

[36] Yan Zhang & Yi Zhou (2009): *Knowledge forgetting: Properties and applications*. *Artificial Intelligence* 173(16), pp. 1525–1537, doi:10.1016/j.artint.2009.07.005.

An Evaluation of Massively Parallel Algorithms for DFA Minimization

Jan Martens
Leiden University
The Netherlands
j.j.m.martens@liacs.leidenuniv.nl

Anton Wijs
Eindhoven University of Technology
The Netherlands
a.j.wijs@tue.nl

We study parallel algorithms for the minimization of Deterministic Finite Automata (DFAs). In particular, we implement four different massively parallel algorithms for DFA minimization on Graphics Processing Units (GPUs). Our results confirm the expectations that the algorithm with the theoretically best time complexity is not practically suitable to run on GPUs due to the large amount of resources needed. We empirically verify that parallel partition refinement algorithms from the literature perform better in practice, even though their time complexity is worse. Lastly, we introduce a novel algorithm based on partition refinement with an extra parallel partial transitive closure step and show that on specific benchmarks it has better run-time complexity and performs better in practice.

1 Introduction

In contrast to sequential chips, the processing power of parallel devices keeps increasing. Graphics Processing Units, or GPUs, are examples of such devices. Originating from the need to do simple computations for many (independent) pixels to generate graphics, GPUs have also shown useful as computational powerhouses, and led to general-purpose computing on GPUs (GPGPU). Most convincingly, GPUs have become indispensable in training models for artificial intelligence. Because of the enormous potential of GPUs, it is important to investigate how computational problem solving can be accelerated with them.

Deterministic Finite Automata (DFAs) are one of the simplest computational formalisms. The natural problem of computing a minimal machine that is equivalent to a given machine w.r.t. the input is omnipresent in the field of theoretical computer science. In the case of DFAs the problem has a rich history. The first method that computes a minimal DFA dates back to Moore's framework [13], and is a *partition refinement* algorithm. Later, this algorithm was adapted by Hopcroft [8] to a quasi-linear time algorithm.

The complexity class known as Nick's Class (*NC*) consists of the problems that can be solved in polylogarithmic time with a parallel machine using a polynomial number of parallel processors. It is an open question whether $NC \stackrel{?}{=} P$, but it is widely believed that this is not the case. Similar to the assumption that decision problems not in P are inherently difficult (known as Cobham's thesis), we can think of P-complete problems as being inherently sequential.

The problem of minimizing DFAs is known to be in *NC* [4], which intuitively means it can be efficiently computed in parallel. In contrast, the problem of computing bisimilarity on non-deterministic structures is known to be P-complete [1]. Interestingly, the most efficient sequential algorithms for these two problems, i.e., Hopcroft's algorithm [8] and an algorithm based on Paige-Tarjan [14], respectively, are very similar. In particular, these algorithms are both *partition refinement* algorithms.

The parallel algorithms studied for computing bisimilarity on non-deterministic structures and DFA minimization are also partition refinement algorithms [12, 16, 18, 21]. Since DFA minimization is in

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 138–153, doi:10.4204/EPTCS.409.13

© J.J.M. Martens & A.J. Wijs
This work is licensed under the
Creative Commons Attribution License.

NC, there is a parallel sublinear time algorithm. However, none of these partition refinement algorithms studied have a sublinear run-time. A linear lower bound for the parallel run-time was proven in [6] for any parallel partition refinement algorithm deciding bisimilarity, and this result also directly applies to deterministic structures such as DFA minimization. This means that no partition refinement algorithm can achieve the theoretically optimal run-time on parallel machines. It is therefore interesting to investigate whether there is an algorithm that is not a partition refinement algorithm that performs better in parallel than partition refinement algorithms.

The algorithm introduced in [4] runs in logarithmic time. However, the work is mainly theoretical and the large amount of parallel processors and memory required makes it unlikely to scale well in practice. The main constraint here is the need to compute the transitive closure for the underlying graph of the DFA. It seems hard to find a significant improvement in the number of parallel processors needed.

In this paper we compare implementations of different parallel algorithms for DFA minimization on GPUs, using the various parallel algorithms proposed in the literature as a basis. We establish that the logarithmic runtime complexity with the construction from [4] is not feasible due to the large amount of processors needed. Additionally, we find that on our benchmarks the partition refinement algorithm that uses sorting in each iteration performs better than the naive splitting strategy on the more diverse benchmarks from the VLTS benchmark set,[1] but worse for benchmarks that are known to be hard for partition refinement algorithms. Finally, we show a method of adding a partial transitive closure as a preprocessing step that can significantly increase the performance on benchmarks with a very specific shape.

2 Preliminaries

We write $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ for the set of booleans, $\mathbb{N}$ for the set of natural numbers, and for numbers $i, j \in \mathbb{N}$ we define $[i, j] = \{c \in \mathbb{N} \mid i \leq c \leq j\} \subseteq \mathbb{N}$, the closed interval from i to j. Given an alphabet Σ, a sequence $a_1 a_2 \dots a_n$ of symbols from Σ is called a word. We write Σ^* for the set containing all finite sequences of letters in Σ. The empty-sequence consisting of no symbols is written as ε.

Definition 1. (Deterministic Finite Automaton) A deterministic finite automaton (DFA) $A = (Q, \Sigma, \delta, F, q_0)$ is a five-tuple consisting of:

- a finite set of states Q,
- a finite alphabet Σ,
- a transition function $\delta : Q \times \Sigma \to Q$,
- a set of accepting states $F \subseteq Q$, and
- an initial state $q_0 \in Q$.

We sometimes write $q \xrightarrow{a} q'$ if $\delta(q, a) = q'$. The function $\delta^* : Q \times \Sigma^* \to Q$ extends the transition function to words and is defined inductively for all words in Σ^* as follows:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

Given a DFA $A = (Q, \Sigma, \delta, F, q_0)$, a word $w \in \Sigma^*$ is *accepted* iff $\delta^*(q_0, w) \in F$. The language of a DFA A, notation $\mathscr{L}(A)$, is the set of all words $w \in \Sigma^*$ that are accepted by A.

[1] https://cadp.inria.fr/resources/vlts (visited on: 19-04-2024).

We consider the problem of computing the minimal DFA, i.e., given a DFA A, identifying the DFA A' with the smallest number of states such that $\mathscr{L}(A') = \mathscr{L}(A)$.

Minimizing DFAs consists of combining undistinguishable states and deleting unreachable states. The main part of the problem consists of combining states, and removing unreachable states can be seen as a simple pre-processsing step. For the remainder of the paper, we assume that all the states in a DFA are reachable from its initial state. The algorithms can be seen as computing bisimilarity, or the coarsest set partition problem, without the preprocessing step that removes unreachable states.

Representation. For an input automaton $A = (Q, \Sigma, \delta, F, q_0)$, we assume that the states in Q and letters in Σ are represented by unique indices, i.e., $Q = \{0, \dots, |Q|\}$ and $\Sigma = \{0, \dots |\Sigma|\}$. The transition function δ is represented by $|\Sigma|$ arrays of length $|Q|$, such that for state $q \in Q$ and letter $a \in \Sigma$, $\delta[a][q] = \delta(q, a)$.

The PRAM Model. The complexities we mention assume the model of the *Parallel Random Access Machine (PRAM)*. The PRAM is a natural extension of the RAM model, where parallel processors have access to a shared memory. A PRAM consists of a sequence of processors $P_0, P_1, \dots$ and a function $\mathscr{P}$ that given the size of the input defines a bound on the number of processors used.

Each processor P_i has the natural instructions of a normal RAM and in addition has an instruction to retrieve its unique index i. All processors run the same program in lock-step, using their index to identify the data they need to access. This parallel processing is called single-instruction multiple data (SIMD).

There are many different ways to handle data-races. We assume the concurrent read, concurrent write (CRCW) model following [17], where processors are allowed to read from and write to the same memory location concurrently. After multiple concurrent writes to the same memory location, that location contains the result of one of those writes.

GPUs. While in reality, no device completely adheres to the PRAM model, recent hardware advancements has led to devices that are getting better and better at approximating this model. The GPU, in particular, is a very suitable target platform for PRAM algorithms, as it has been specifically designed for SIMD processing. The performance of GPU programs typically relies on launching tens to hundreds of thousands of threads, as the performance of these programs is often memory-bound: accessing the input data in the GPU's *global* memory, in NVIDIA CUDA terminology, is relatively slow. This latency can be hidden by a GPU via fast context switching between threads. As one thread is waiting for data to be retrieved, another thread is executed in the meantime on the same processor. It is this fast context switching between threads that allows GPUs, typically equipped with several thousands of cores, to virtually execute hundreds of thousands of threads concurrently. In the current work, we employ NVIDIA GPUs, programs for which can be written in CUDA C++.

3 The algorithms

3.1 Transitive closure

The first algorithm we discuss has theoretical polylogarithmic runtime [4]. However, the large amount of memory and parallel processors it uses makes it unlikely to work in practice. Here we confirm this fact.

The idea is rather simple; build a graph with nodes $V = Q \times Q$ and edges E containing $(q, q') \to (p, p')$ iff there is a letter $a \in \Sigma$ such that $\delta(q, a) = p$ and $\delta(q', a) = p'$. Initially, in the array `Apart` we label the nodes $(q, q') \in V$ to be inequivalent if $q_1 \in F \iff q_2 \notin F$. Any two states $q_1, q_2 \in Q$ are not equivalent iff they were initially labelled in `Apart` or there is a path to a labelled node. Computing this reachability of false nodes can be seen as computing the transitive closure in the directed graph (V, E) containing n^2

Algorithm 1 Transitive DFA minimization `trans`.

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$ where $|Q| = n$
Output: The minimal quotient automaton represented in the matrix *Apart*

```
 1: V :: Q × Q                                    ▷ Nodes of graph consisting of pair of states
 2: Apart :: Array[n^2] of type 𝔹
 3: Reach :: Array[n^2][n^2] of type 𝔹            ▷ Represents reachability in V, initially false
 4: do in parallel for (q, q') ∈ V                ▷ Initializes data structures in parallel.
 5:     Apart[(q, q')] := (q ∈ F ⟺ q' ∉ F)        ▷ State initially unequal
 6:     for all a ∈ Σ do
 7:         Reach[(q, q')][(δ(q, a), δ(q', a))] := true
 8: stable := false
 9: while ¬stable do
10:     stable := true
11:     do in parallel for s, t, u ∈ V
12:         if Reach[s][t] and Reach[t][u] and ¬Reach[s][u] then
13:             Reach[s][u] := true
14:     do in parallel for s, t ∈ V
15:         if Reach[s][t] and Apart[t] and ¬Apart[s] then
16:             Apart[s] := true
17:             stable := false
```

nodes. In parallel this computation can be done in polylogarithmic running time using $O(|V|^3)$ parallel processors [9, Chapter 5.5.].

Algorithm 1, which we refer to as `trans`, implements this idea. First, at lines 4–7 (l.4–7), the graph is constructed, inequivalent nodes labelled in the *Apart* data structure and the edges stored in the adjacency matrix *Reach*. Next, the parallel transitive closure of *Reach* is computed and *Apart* updated accordingly. If in an iteration there is no new pair of states labelled *Apart* the algorithm is finished. The minimal automaton is represented in the graph where states $q, q' \in Q$ can be combined if $\neg Apart(q, q')$.

Computing the transitive closure for a directed graph in logarithmic time requires many processors. Our naive implementation requires $|V|^3$ parallel processors. Given a DFA with n states, this means that since $|V| = n^2$, we require n^6 processors. Theoretically, more efficient methods are known for computing the transitive closure, which uses matrix multiplication. Matrix multiplication can be computed with $O(n^\omega)$ operations, where currently $\omega \leq 2.372\ldots$, this means we can compute our transitive closure with $O(|V|^\omega)$ processors. Since these algorithms are non-trivial and already $|V| = n^2$, we believe these improvements would not significantly change the results mentioned here.

3.2 Naive partition refinement

The next algorithm, `naivePR`, is an adaptation of the parallel algorithm for bisimilarity checking of Kripke structures from [12]. The program runs on a PRAM with $\max(n, m)$ processes, where n is the number of states and $m = |\Sigma| * n$ is the number of transitions in the input DFA.

The algorithm applies *partition refinement*: states are initially partitioned into *blocks*, and the algorithm repeatedly splits blocks into smaller blocks until a fix-point is reached. Once this has happened, each block represents one state of the minimized DFA.

When splitting blocks in parallel, one particular challenge is how to identify newly created blocks,

as each new block requires a unique identifier. Algorithm 2 does this by means of a leader election procedure: for each block, one of its states is elected leader, meaning that it is used as an identifier to refer to the block. In this way each iteration of the algorithm takes constant time if performed on a parallel machine that has concurrent writes.

In Algorithm 2, at l.1, an array *block* is initialized that defines for every state in Q its current block (as represented by a leader in Q). An array *new_leader* is defined at l.2 that is used to elect new leaders. At l.3, the initial leaders are selected: one state $q_f \in F$ for the block consisting of all the accepting states $q \in F$, and one state $q_n \in Q \setminus F$ for all the non-accepting states $q' \in Q \setminus F$. The array *block* is subsequently initialised using these leaders (l.4–5).

Next, partition refinement is applied inside the **while**-loop at l.7. The variable *stable* is used to monitor whether a fix-point has been reached, which has happened as soon as no blocks can be split any further. At the start of each iteration through the **while**-loop, *stable* is set to `true` (l.8). Next, all transitions of the DFA are processed in parallel (l.9), and for each transition $q \xrightarrow{a} q'$, it is checked whether $block[q']$ differs from the block that the leader $block[q]$ can reach via an a-transition. If it does, then q should be separated from its leader. At l.11, q is assigned to $new_leader[block[q]]$, the latter being the position where the leader for the new block will be elected. Here, the result of concurrent writes, as allowed by the PRAM CRCW model, is used for leader election.

Subsequently, when l.12 is reached, *new_leader* contains the newly elected leaders: specifically, at $new_leader[block[q]]$, the leader for the new block created by splitting off states from $block[q]$ is stored. In the parallel loop of l.12, the transitions are once more processed in parallel, and whenever a state turns out to differ from its leader regarding block reachability (l.13), the leader of that state is updated (l.14). Finally, since a block has been split, *stable* is set to `false` at l.15.

The largest difference between Algorithm 2 and the original algorithm [12] is that Algorithm 2 splits blocks directly w.r.t. the leader, as opposed to first selecting one particular block as *splitter*, and splitting those blocks in which some states differ w.r.t. their leader concerning the ability to reach the splitter. The reason for this difference is that for DFAs, comparing the outgoing transitions of two states is much more straightforward, as each state has exactly one outgoing transition for every $a \in \Sigma$. In the setting of LTSs, due to non-determinism it is not possible to directly compare the behaviour of a state with the leader state, and hence a fixed splitter is chosen before.

In Algorithm 2, leader election is performed in two phases: in the first phase (l.9–11), states are written to *new_leader* to elect new leaders, and the results are subsequently read at l.12–15. One could argue that this is inefficient, and that it would perhaps be better to combine these two phases. This is possible, but it requires the use of atomic *compare-and-swap* (CAS) operations. This is illustrated in Algorithm 3. In the single loop at l.11–17, new leaders are written to *and* read from *new_leader*. At l.14–15, the use of a compare-and-swap operation is described: in one atomic operation, the current value stored at $new_leader[leader]$ is stored in *new_block*, and it is checked whether $new_leader[leader]$ is equal to the initial value $\perp$, and if it is, it is set to q. Next, at l.16, if *new_block* is equal to $\perp$, it means q has been elected as leader. Otherwise, *new_block* indicates which state is the new leader. Note that for this to work, after execution of the loop at l.11, the values of *new_leader* have to be reset to $\perp$.

In practice, we experienced that a GPU implementation (in CUDA 12.2) of Algorithm 3 exhibits similar runtimes compared to a GPU implementation of Algorithm 2. The benefit of merging the loops seems to be negated by the use of atomic operations. For this reason, when discussing the experiments in Section 4.2, we do not involve Algorithm 3.

Algorithm 2 Parallel leader-election-based algorithm `naivePR`.

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$ where $|Q| = n$
Output: The minimal quotient automaton represented in the array *block*

1: *block* :: Array$[n]$ of type Q
2: *new_leader* :: Array$[n]$ of type Q
3: Select initial leader states $q_f \in F$ and $q_n \in Q \setminus F$
4: **do in parallel for** $q \in Q$
5: $\quad$ $block[q] := (q \in F \ ?\ q_f : q_n)$ ▷ Initialize
6: $stable :=$ `false`
7: **while** $\neg stable$ **do**
8: $\quad$ $stable :=$ `true`
9: $\quad$ **do in parallel for** $q, a \in Q \times \Sigma$
10: $\quad\quad$ **if** $block[\delta(q,a)] \neq block[\delta(block[q],a)]$ **then**
11: $\quad\quad\quad$ $new_leader[block[q]] := q$ ▷ Leader election
12: $\quad$ **do in parallel for** $q, a \in Q \times \Sigma$
13: $\quad\quad$ **if** $block[\delta(q,a)] \neq block[\delta(block[q],a)]$ **then**
14: $\quad\quad\quad$ $block[q] := new_leader[block[q]]$ ▷ Split from leader
15: $\quad\quad\quad$ $stable :=$ `false`

Algorithm 3 Parallel leader-election based `naivePR` with atomics.

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$, where $|Q| = n$
Output: The minimal quotient automaton represented in the array *block*

1: *block* :: Array$[n]$ of type Q
2: *new_block* :: Q
3: *leader* :: Q
4: *new_leader* :: Array$[n]$ of type Q
5: Select initial leader states $q_f \in F$ and $q_n \in Q \setminus F$
6: **do in parallel for** $q \in Q$
7: $\quad$ $new_leader[q] := \perp$ ▷ Initialize
8: $\quad$ $block[q] := (q \in F \ ?\ q_f : q_n)$
9: **while** $\neg stable$ **do**
10: $\quad$ $stable :=$ `true`
11: $\quad$ **do in parallel for** $q, a \in Q \times \Sigma$
12: $\quad\quad$ $leader := block[q]$
13: $\quad\quad$ **if** $block[\delta(q,a)] \neq block[\delta(leader,a)]$ **then**
14: $\quad\quad\quad$ $\{new_block := new_leader[leader];$ ▷ Leader election with CAS
15: $\quad\quad\quad$ $new_leader[leader] = \perp \ ?\ new_leader[leader] := q\}$
16: $\quad\quad\quad$ $block[q] := new_block = \perp \ ?\ q : new_block$ ▷ Split from leader
17: $\quad\quad\quad$ $stable :=$ `false`
18: $\quad$ **do in parallel for** $q \in Q$
19: $\quad\quad$ $new_leader[q] := \perp$

3.3 Sorting arrays

The next algorithm is an algorithm inspired by [16, 18]. Similar to Algorithm 2, this algorithm also performs partition refinement, but instead of doing so using leader elections, it repeatedly computes a *signature* for every state, and sorts the states w.r.t. their signatures. This method allows splitting a block in more than two subblocks with the downside that each iteration takes more than constant parallel time.

The algorithm from [18] uses hashing to construct and compare signatures. Since sorting arrays is a very native operation on GPUs, we follow [16] and use a sorting approach to construct the new blocks.

Algorithm 4 presents this approach as `sortPR`. Again, an array *block* is created (l.1). In addition, an array *state* is used for sorting the states (l.2). The signature of a state consists of a list of block IDs, one for each $a \in \Sigma$: $signature[q][a]$ is equal to q' iff $q \xrightarrow{a} q''$ and $block[q''] = q'$.

The array *new_block* is used to store the results of assigning new blocks to states (l.4). Finally, the current number of blocks is stored at l.5 in *num_blocks*.

Next, at l.6–8, *block* and *state* are initialised. The block consisting of all accepting states is given ID 0, while the other states are assigned to block 1 (l.7). All the states are added to *state* at l.8.

In the loop at l.9–19, the partition refinement is performed until a fix-point has been reached, i.e., the number of blocks has not increased (l.19). In each iteration of this loop, the following is performed. First, in parallel, the signatures are updated (l.11–12). After that, *state* is sorted in parallel, using *signature* to compare states. The comparison function is given at l.20–26. First, states are compared based on the block they reside in. If they reside in the same block, then the blocks they can reach via outgoing transitions are compared. Note that the for loop starting in (l.23) is sequential and requires iterating over the alphabet letters in a fixed order.

Once *state* has been sorted, the parallel *adjacent difference* is computed and stored in *new_block*. The result of this is that $new_block[0] = state[0]$ and for all $0 < i < n$, $new_block[i] = are_neq(state[i], state[i-1])$, with *are_neq* as defined at l.27–31. Once $new_block[0]$ has been reset to 0 (l.15), *new_block* contains only 0's and 1's, with each 1 identifying the start of a new block. At l.16, an *inclusive scan* is performed in parallel, resulting in *new_block* having been updated in such a way that for each $0 \leq i < n$, $new_block[i] = \sum_{0 \leq j \leq i} new_block'[j]$, with new_block' referring to *new_block* at the start of executing l.16.

Now, for all $0 \leq i < n$, $new_block[i]$ contains the new block of state $state[i]$. At l.17–18, *block* is updated in parallel to reflect this. As the largest new block ID can be found at $new_block[n-1]$, this location can be used to determine the new number of blocks at l.19.

In [16] it is shown that on average this algorithm has polylogarithmic run-time complexity. The argument given uses the fact that on uniformly sampled DFAs almost all pairs of states have a shortest distinguishing word of polylogarithmic depth. This fact is attributed to [19]. Although this is true for uniformly sampled DFAs, we like to stress that for many use cases and real-life applications this bound does not apply. For example, in the Fibonacci automata presented in Section 4.2 this is not the case. In the automaton $\texttt{Fib}_i$ containing n states, there is a pair of states for which the shortest distinguishing word, and thus also the number of iterations, has length $n-2$.

3.4 Partition refinement using partial transitive closure

In this section, we present a new algorithm `transPR`. The main idea of the algorithm is to perform partition refinement like the algorithms before, but in the initialization compute the transitive closure on each distinct letter. After this initialization step, we use `naivePR` to complete the minimization.

This approach is presented in Algorithm 5. This is done in a data-parallel way which is also used for prefix sum and finding the end of a linked list [7]. On some DFAs, with a rather specific structure, this

Algorithm 4 Parallel sorting-based algorithm `sortPR`

Input: A DFA $A = (Q, \Sigma, \delta, F, q_0)$, where $|Q| = n$ and $|\Sigma| = k$

Output: The minimal quotient automaton represented in the array *block*

```
1:  block :: Array[n] of type ℕ
2:  state :: Array[n] of type Q
3:  signature :: Array[n][k] of type Q
4:  new_block :: Array[n] of type ℕ
5:  num_blocks := 2
6:  do in parallel for q ∈ Q
7:      block[q] := (q ∈ F ? 0 : 1)                          ▷ Initialize
8:      state[q] := q
9:  repeat
10:     num_blocks := new_block[n − 1] + 1                   ▷ Number of blocks before iteration
11:     do in parallel for q, a ∈ Q × Σ
12:         signature[q][a] := block[δ(q, a)]
13:     sort(state, COMPARE)
14:     new_block := adjacent_diff(state, ARE_NEQ)           ▷ Place 1 for each change
15:     new_block[0] := 0
16:     new_block := inclusive_scan(new_block)               ▷ Compute new block labels
17:     do in parallel for q ∈ Q
18:         block[state[q]] = new_block[q]
19: until new_block[n − 1] + 1 = num_blocks

20: function COMPARE(q1, q2)
21:     if block[q1] > block[q2] then return false
22:     if block[q1] < block[q2] then return true
23:     for all a ∈ Σ do
24:         if signature[q1][a] > signature[q2][a] then return false
25:         if signature[q1][a] < signature[q2][a] then return true
26:     return false

27: function ARE_NEQ(q1, q2)
28:     if block[q1] ≠ block[q2] then return true
29:     for all a ∈ Σ do
30:         if signature[q1][a] ≠ signature[q2][a] then return true
31:     return false
```

method exponentially improves the runtime compared to the other partition refinement algorithms.

Algorithm 5 Parallel partition refinement with transitive closure `transPR`

1: $\Sigma^T := \{a^{2^i} \mid a \in \Sigma, i \in [0, \lfloor \log n \rfloor]\}$
2: $\delta^T :: Q \times \Sigma^T \mapsto Q$
3: $\delta^T(q,a) := \delta(q,a)$ for all $a \in \Sigma$
4: **for all** $i \in [1, \lfloor \log n \rfloor]$ **do**
5: **for all** $a \in \Sigma$ **do**
6: **do in parallel for** $q \in Q$
7: $\delta^T(q, a^{2^i}) := \delta^T(\delta^T(q, a^{2^{i-1}}), a^{2^{i-1}})$
8: Perform `naivePR` on the DFA $A' = (Q, \Sigma^T, \delta^T, F, q_0)$

The algorithm works by adding letters for increasingly large words of the same letters. Given an input DFA $A = (Q, \Sigma, \delta, F, q_0)$, we construct a DFA $A' = (Q, \Sigma^T, \delta^T, F, q_0)$ which has the same set of states and final states, but a larger alphabet Σ^T. The alphabet contains the letters $a^{2^0}, a^{2^1}, \ldots, a^{2^{\lfloor \log n \rfloor}}$ for each original letter $a \in \Sigma$. The transition function is computed such that for each new symbol $a^k \in \Sigma^T$ the transition function $\delta^T(q_1, a^k) = q_k$ if in the original DFA Q there are states $q_1, \ldots q_k \in Q$ such that $\delta(q_i, a) = q_{i+1}$ for each $i \in [1, k]$. This can be computed in a logarithmic number of parallel steps, by using the previously computed transitions, as is done at l.7 of Algorithm 5.

The correctness of this algorithm relies on the fact that equality on states is invariant under the partial closure that is added. Indeed, we can see that the DFA A' obtained in Algorithm 5 is language equivalent to the input DFA A if we consider the alphabet letters added as words. If $\delta^T(q, a_T) = q'$ for some $a_T \in \Sigma^T$, then $a_T = a^{2^j}$ for some $a \in \Sigma$ and $j \in [0, \lfloor \log n \rfloor]$. By construction there is a sequence $q_0, \ldots q_k$ such that $q_0 = q$, $q_{i+1} = \delta(q_i, a)$ and $q_k = q'$.

This approach helps in the case of long paths with the same letter. Consider the DFA A from Figure 1. This DFA accepts all words a^j with $j > 8$. Any parallel partition refinement algorithm would need at least 8 iterations to conclude that q_0 is not the same as q_1. However, building the partial transitive closure only requires a logarithmic number of parallel iterations. With this partial transitive closure added, a partition refinement algorithm can in the first iteration conclude that q_0 is different from q_1 since the transition with a^8 leads to different states.

4 Experiments

In this section, we discuss the results of our implementations. We benchmarked the implementations with respect to three families of DFAs: Fibonacci DFAs from [3], bit-splitters $\mathscr{B}_k$ derived from [6], and DFAs derived from a subset of the VLTS benchmark set.

4.1 Benchmarks

Fibonacci DFAs: The first family of DFAs we use for benchmarking consists of so-called Fibonacci automata. These are simple automata with only a unary alphabet. However, they exhibit very particular behaviour. As witnessed in [3], these automata are notoriously hard for partition refinement and the number of iterations of any partition refinement algorithm is n. The automata are called Fibonacci automata due to the close correspondence with Fibonacci words over the binary alphabet, which are

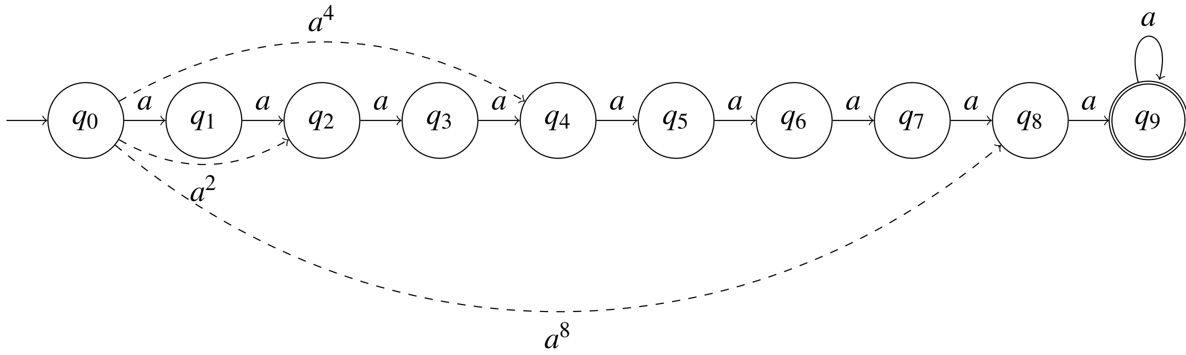


Figure 1: The DFA $A = (\{q_0,\ldots,q_9\},\{a\},\delta,\{q_9\},q_0)$ with the extra partial transitive closure from q_0 added in dashed arrows.

defined inductively as follows: the base cases are $w_0 = 1$, and $w_1 = 0$, and for every $i \in \mathbb{N}$, $w_{n+1} = w_n w_{n-1}$. This gives the following sequence:

$$\begin{aligned} w_2 &= 01 \\ w_3 &= 010 \\ w_4 &= 01001 \\ w_5 &= 01001010 \\ &\ldots \end{aligned}$$

For every $n \in \mathbb{N}$, we define the automaton $\texttt{Fib}_n = (Q,\{a\},\delta,q_0,F)$ as follows, with $w_n[i]$ referring to the i-th bit in the bit sequence w_n:

- the set of states is $Q = \{q_i \mid i \in [0,|w_n|]\}$;
- the transition function is $\delta(q_i,a) = q_{i+1 \mod |w_n|}$;
- the set of final states is $F = \{q_i \mid q_i \in Q \text{ and } w_n[i] = 1\}$.

Bit-splitters: The second family of automata consists of the so-called *bit-splitters* $\mathscr{B}_n$. For $n \in \mathbb{N}$, the bit-splitter $\mathscr{B}_n$ is a deterministic automaton with 2^n states and an alphabet consisting of $n-1$ symbols. By construction, during partition refinement, every time a block can be split, it is split in two blocks of equal size. This property makes the family inherently hard for partition refinement algorithms. However, the parallel algorithm requires only a logarithmic number of iterations to compute the minimal DFA. The bit splitter $\mathscr{B}_3$ is given in Figure 2.

The family does not contain an initial state, and comes from the setting of Labelled Transition Systems (LTSs). An LTS is a graph structure with a finite number of states and transitions between states, with each transition having an action label.

Since it is a hard example for partition refinement, we use it for this purpose and allow the absence of an initial state. In the following, states σ represent bit sequences of length n, i.e., $\sigma \in \{0,1\}^n$. We define $\mathscr{B}_1 = (Q_1,\Sigma_1,\delta_1,F_1)$, where $Q_1 = \{\texttt{0},\texttt{1}\}$, $\Sigma_1 = \emptyset$ and $F_1 = \{\texttt{1}\}$. Given the automaton $\mathscr{B}_n = (Q_n,\Sigma_n,\delta_n,F_n)$ for some $n \in \mathbb{N}$, we define $\mathscr{B}_{n+1} = (Q_{n+1},\Sigma_{n+1},\delta_{n+1},F_{n+1})$, such that:

- The set of states contains two copies of Q_n, i.e., $Q_{n+1} = \{\texttt{0}\sigma,\texttt{1}\sigma \mid \sigma \in Q_n\}$,

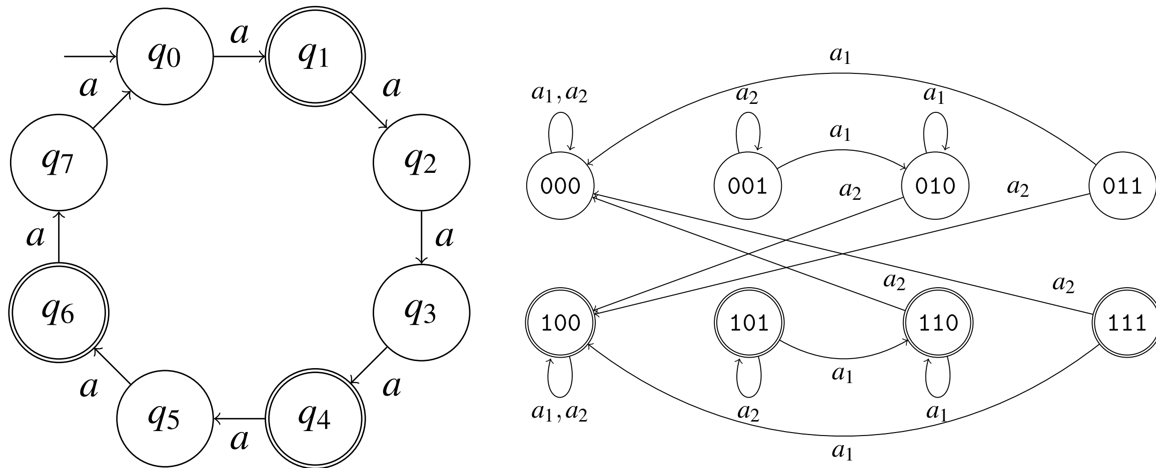


Figure 2: The DFA $\texttt{Fib}_5$ on the left, and the DFA $\mathscr{B}_3$ on the right.

- One fresh symbol $a_n \notin \Sigma_n$ is added to the alphabet: $\Sigma_{n+1} = \Sigma_n \cup \{a_n\}$,
- The transition function δ_{n+1} is defined such that for each $a_m \in \Sigma_n$, and state $\texttt{b}\sigma \in Q_{n+1}$, it maintains the behaviour of $\mathscr{B}_n$, i.e., $\delta_{n+1}(a_m, \texttt{b}\sigma) = \texttt{b}\delta_n(a_m, \sigma)$. For the fresh symbol $a_n \in \Sigma_{n+1} \setminus \Sigma_n$, δ_{n+1} is extended as follows, with $\bar{\texttt{b}}$ being the bit flipped version of $\texttt{b}$, i.e., $\texttt{b} = 0 \iff \bar{\texttt{b}} = 1$:

$$\delta_{n+1}(\texttt{b}\sigma, a_n) = \begin{cases} \bar{\texttt{b}}\texttt{0}^n & \text{If } \sigma[0] = 1, \\ \texttt{b}\sigma & \text{otherwise.} \end{cases}$$

- the set of accepting states is $F_{n+1} = \{\texttt{1}\sigma \mid \sigma \in Q_n\}$.

As previously mentioned, bit-splitter DFAs are constructed in such a way that they are inherently hard to minimize by partition refinement. Each bit-splitter $\mathscr{B}_{n+1}$ combines two copies of the bit-splitter $\mathscr{B}_n$, with the transition function defined in such a way that each possible split divides an existing block in two blocks of the same size. This results in a DFA in which the amount of required work for splitting is large, since each split involves moving many states to the new block. However, because in each split a block is split in two parts of the same size, the number of sequential splits needed is smaller than for the Fibonacci automata.

VLTSs: Lastly, we benchmark our implementations against the VLTS benchmark suite.[2] The VLTS acronym stands for *Very Large Transition Systems*. This suite consists of LTSs that originate from modelling protocols and concurrent systems. Some of the benchmarks are from case studies from industrial systems.

The transition relation of an LTS does not need to be deterministic, nor complete. We turn an LTS into a DFA by first making the LTS deterministic such that each state has at most one outgoing transition for every label, using the powerset construction algorithm [15]. To convert the deterministic LTS to a DFA, we need to complete the transition function and define which states are accepting. We define all the states as accepting and add one new non-accepting state $\perp$. For each state q and label a for which there exists no transition with that label from q, we add a new transition labelled a to $\perp$, i.e. $\delta(q,a) = \perp$.

[2] `https://cadp.inria.fr/resources/vlts` (visited on: 04-2024).

Name	N	Iterations	Time (ms)	Memory(Mb)	#threads
$\texttt{Fib}_4$	8	3	0.3	0	589,824
$\texttt{Fib}_5$	13	4	0.7	0	6,230,016
$\texttt{Fib}_6$	21	5	7.8	0	88,510,464
$\texttt{Fib}_7$	34	5	159.9	0	1,620,545,536
$\texttt{Fib}_8$	55	6	3,034.9	10	27,955,840,000
$\texttt{Fib}_9$	89	7	66,846.7	60	498,865,340,416
$\texttt{Fib}_{10}$	144	t/o	t/o	412	8,943,640,510,464

Table 1: Results of running the algorithm `trans` on the Fibonacci automata.

This completes the transition function and creates a DFA accepting all the words corresponding with a path through the original LTS.

Due to state-space explosion we were not able to make all VLTS benchmarks deterministic. We used all benchmarks for which the computation to make them deterministic took less than ten minutes.

4.2 Results

The algorithms were implemented in CUDA C++ and compiled using the CUDA toolkit 12.2, with the implementation of `sortPR` using the Thrust library for sorting and computing the adjacent differences and inclusive scans [2]. Experiments were conducted on a device running Linux Mint 20, equipped with an NVIDIA TITAN RTX GPU with 24 GB of memory and 4,608 cores. Such a GPU can manage trillions of light-weight threads. Thanks to fast context switching between threads, a GPU can typically handle a few hundred thousand threads as if they execute in parallel.

The reported times are the average of five separate runs. Benchmarks that did not finish within five minutes were aborted, in which case we registered a timeout 't/o'. Benchmarks for which there was not enough memory are indicated by 'OoM'.

Transitive approach: The results of running Algorithm 1 on the Fibonacci automata are given in Table 1. As expected, the number of threads used to compute the transitive closure in parallel grows very quickly. Although the number of iterations of the algorithm is indeed logarithmic, the available parallelism is not sufficient to lead to logarithmic run times. We only use this set of small Fibonacci automata for this algorithm. It already suggests that for a relatively small amount of states (~ 100), obtaining the required resources is already infeasible. The other benchmarks are almost completely out of range of the algorithm.

Partition refinement algorithms: The results of running the parallel partition refinement algorithms, `naivePR` (Algorithm 2), `sortPR` (Algorithm 4), and `transPR` (Algorithm 5) are given in Table 2 and Table 3 for the different benchmarks.

First, we observe in Table 2 that on the Fibonacci automata the `naivePR` performs better than `sortPR`. This can be explained by the fact that the number of iterations is n for both algorithms, while each iteration in `sortPR` is slower than in `naivePR`. Another interesting observation here is that for all benchmarks $\texttt{Fib}_{18},\ldots,\texttt{Fib}_{28}$ the run time of the algorithm `naivePR` scales linearly with the number of states n. Since the number of parallel iterations is $n-2$ for all these benchmarks, each parallel iteration processing up to $\sim 500k$ states took a similar amount of time. In other words, the GPU was able to run around $500k$ threads as if they ran in parallel. This confirms the statement about fast context switching at the beginning of Section 4.2.

	Benchmark metrics			Times (ms)			Iterations		
Name	N	$\|\Sigma\|$	Size output	naivePR	sortPR	transPR	naivePR	sortPR	transPR
$\texttt{Fib}_{20}$	17,711	1	17,711	308.8	3,909.2	1.7	17,710	17,710	14
$\texttt{Fib}_{21}$	28,657	1	28,657	494.2	6,374.2	2.4	28,656	28,656	25
$\texttt{Fib}_{22}$	46,368	1	46,368	778.7	11,712.1	4.1	46,367	46,367	61
$\texttt{Fib}_{23}$	75,025	1	75,025	1,241.3	21,366.6	8.0	75,024	75,024	101
$\texttt{Fib}_{24}$	121,393	1	121,393	2,006.7	34,793.1	12.5	121,392	121,392	104
$\texttt{Fib}_{25}$	196,418	1	196,418	3,251.3	64,411.7	18.3	196,417	196,417	138
$\texttt{Fib}_{26}$	317,811	1	317,811	5,277.8	178,367.4	49.8	317,810	317,810	102
$\texttt{Fib}_{27}$	514,229	1	514,229	8,607.7	t/o	96.1	514,228	t/o	268
$\texttt{Fib}_{28}$	832,040	1	832,040	22,723.0	t/o	178.4	832,039	t/o	299
$\texttt{Fib}_{29}$	1,346,269	1	1,346,269	59,510.8	t/o	726.9	1,346,268	t/o	755
$\texttt{Fib}_{30}$	2,178,309	1	2,178,309	141,601.0	t/o	1,109.3	2,178,308	t/o	914
$\mathscr{B}_{15}$	32,768	14	32,768	0.8	25.8	1.7	14	14	2
$\mathscr{B}_{16}$	65,536	15	65,536	1.4	29.7	3.7	15	15	2
$\mathscr{B}_{17}$	131,072	16	131,072	2.6	54.3	9.4	16	16	2
$\mathscr{B}_{18}$	262,144	17	262,144	5.0	107.2	25.6	17	17	2
$\mathscr{B}_{19}$	524,288	18	524,288	9.6	235.7	60.9	18	18	2
$\mathscr{B}_{20}$	1,048,576	19	1,048,576	19.3	520.2	139.8	19	19	2
$\mathscr{B}_{21}$	2,097,152	20	2,097,152	39.8	1,148.6	312.2	20	20	2
$\mathscr{B}_{22}$	4,194,304	21	4,194,304	82.6	2,538.5	728.7	21	21	2
$\mathscr{B}_{23}$	8,388,608	22	8,388,608	170.3	5,612.7	1,612.1	22	22	2
$\mathscr{B}_{24}$	16,777,216	23	16,777,216	352.6	12,351.8	OoM	23	23	OoM
$\mathscr{B}_{25}$	33,554,432	24	33,554,432	737.4	27,092.2	OoM	24	24	OoM
$\mathscr{B}_{26}$	67,108,864	25	67,108,864	1,541.5	59,203.8	OoM	25	25	OoM

Table 2: Results of running the partition refinement algorithms on the `Fib` and $\mathscr{B}$ benchmark set.

Finally, for the Fibonacci automata, we see that `transPR` performs significantly better on this benchmark. This can be explained by the fact that the partial transitive closure reduces the number of iterations of the algorithm significantly.

The results on the bit-splitter automata in Table 2 show that the improvement of `transPR` does not work on all automata. The high number of alphabet letters together with the structure of the automata make the transitive closure less effective, making `naivePR` much faster.

For the VLTS benchmark set we see the power of `sortPR` in Table 3. In some benchmarks, like 'vasy_69_520' the algorithm performs significantly better. In these examples, it helps that in `sortPR`, in each iteration a block can be split into many subblocks, which is not the case in the other algorithms.

Since the VLTS benchmarks originate from communication protocols and concurrent systems, the success of `sortPR` suggests that for DFAs that represent 'real' systems, this algorithm is a solid choice for efficient DFA minimization. However, the experiments with the Fibonacci and bit-splitter families of DFAs demonstrate room for improvement.

5 Conclusions & Future work

We implemented and compared different parallel algorithms for DFA minimization on GPUs. We find that the *NC* algorithm `trans` with parallel logarithmic run-time does not scale well because of the large number of resources needed. Instead, we find that the partition refinement algorithms perform better.

	Benchmark metrics			Times (ms)			Iterations		
Name	N	$\|\Sigma\|$	Size output	naivePR	sortPR	transPR	naivePR	sortPR	transPR
cwi_1_2	4,448	26	2,416	5.4	66.7	25.1	308	38	621
cwi_2416_17605	503	15	58	0.8	38.2	0.4	40	40	8
cwi_3_14	63	2	63	1.2	9.1	0.4	61	61	8
vasy_0_1	92	2	10	0.2	3.9	0.4	6	5	5
vasy_1_4	6,087	6	29	0.4	8.5	0.9	15	7	20
vasy_10_56	10,850	12	2113	8.7	40.2	30.9	519	33	791
vasy_1112_5290	1,112,491	23	266	135.4	386.8	2,049.2	246	4	231
vasy_157_297	157,605	235	4,290	455.1	1,736.3	11,312.0	1,049	27	1,306
vasy_164_1619	109,911	37	1,025	69.9	50.5	823.4	770	4	766
vasy_166_651	393,147	211	392,175	159,265.6	1,070.6	t/o	175,764	19	t/o
vasy_18_73	419,664	17	31,952	1,586.1	305.2	34,055.2	13,343	27	18,444
vasy_25_25	25,218	25,216	25,218	262,878.6	3,502.7	t/o	25,217	2	t/o
vasy_386_1171	355,790	73	114	36.9	489.4	766.0	58	8	113
vasy_40_60	40,007	3	40,007	331.6	8,391.5	845.2	20,004	20002	20,004
vasy_5_9	5,088	31	138	2.2	14.3	7.0	113	5	124
vasy_574_13561	574,058	141	3,578	2,332.2	976.5	64,312.6	2,351	5	2,634
vasy_6120_11031	3,190,785	125	5,216	13,186.6	21,886.0	t/o	2,373	21	t/o
vasy_65_2621	65,538	72	65,537	2,591.8	38.3	47,568.0	36,575	4	38,999
vasy_66_1302	209,791	81	208,419	42,864.9	96.0	t/o	179,861	8	t/o
vasy_69_520	74,958	135	74,958	7,223.0	124.2	181,611.4	49,723	12	74,667
vasy_720_390	87,741	49	3,279	176.0	57.1	2,961.7	2,936	5	2,950
vasy_8_24	20,306	11	560	5.9	26.8	22.1	282	17	348
vasy_8_38	8,922	81	220	5.7	44.1	31.5	174	5	215
vasy_83_325	393,147	211	392,175	162,495.0	1,074.4	t/o	173,218	19	t/o

Table 3: Results of running the partition refinement algorithms on the VLTS benchmark set.

This might be seen as contradictory since these partition refinement algorithms have inherently linear parallel run-times.

When comparing the different partition refinement algorithms, the structure of the input DFA is of high influence. The trade-off is that in `sortPR` each iteration takes more time than in `naivePR`, but in `sortPR`, an iteration has the potential to lead to a block being split into more than two subblocks. When this happens sufficiently often, fewer iterations are needed. This leads to `sortPR` being slower in cases where the number of iterations is high. In other benchmarks, it leads to fewer iterations and thereby a significant speed-up.

Finally, we showed a way to incorporate a partial transitive closure in partition refinement algorithms. We showed that for a specific class of DFAs this approach leads to logarithmic run-times, where every partition refinement algorithm is inherently linear.

As future work it would be interesting to further investigate sublinear time parallel algorithms for DFA minimization. Specifically, there are two key questions that come to mind. The first question is: what is a reasonable number of parallel processes necessary for a poly-logarithmic time parallel algorithm? It seems feasible to use a similar argument as in [11] to get a superlinear lower bound. However, the gap between the $O(n^{2\omega})$ processors[3] used in Algorithm 1 remains large. The second question is: can a method such as the one presented in this paper using the partial transitive closure be implemented in such a way that the run time will be sublinear with high probability, i.e. any parallel run-time $O(n^{1-\varepsilon})$ for some $\varepsilon \geq 0$. A good starting point would be the recent work on parallel reachability algorithms [20, 5, 10].

[3] If matrix multiplication can be computed in time $O(n^{\omega})$, currently best known bounds $\omega \leq 2.372\ldots$.

References

[1] J. Balcázar, J. Gabarro & M. Santha (1992): *Deciding bisimilarity is P-complete*. Formal aspects of computing 4(1), pp. 638–648, doi:10.1007/BF03180566.

[2] N. Bell & J. Hoberock (2012): *Thrust: A Productivity-Oriented Library for CUDA*. In: GPU Computing Gems Jade Edition, chapter 26, Morgan Kaufmann Publishers Inc., pp. 359–371, doi:10.1016/C2010-0-68654-8.

[3] G. Castiglione, A. Restivo & M. Sciortino (2008): *Hopcroft's Algorithm and Cyclic Automata*. In C. Martín-Vide, F. Otto & H. Fernau, editors: Proc. of LATA 2008, LNCS 5196, Springer, pp. 172–183, doi:10.1007/978-3-540-88282-4_17.

[4] S. Cho & D.T. Huynh (1992): *The parallel complexity of coarsest set partition problems*. Information Processing Letters 42(2), pp. 89–94, doi:10.1016/0020-0190(92)90095-D.

[5] J.T. Fineman (2018): *Nearly Work-Efficient Parallel Algorithm for Digraph Reachability*. In: Proc. of STOC 2018, ACM, p. 457–470, doi:10.1145/3188745.3188926.

[6] J.F. Groote, J.J.M. Martens & E.P. de Vink (2023): *Lowerbounds for bisimulation by partition refinement*. Logical Methods in Computer Science Volume 19, Issue 2, doi:10.46298/lmcs-19(2:10)2023.

[7] W.D. Hillis & G.L. Steele Jr. (1986): *Data Parallel Algorithms*. Communications of the ACM 29(12), pp. 1170–1183, doi:10.1145/7902.7903.

[8] J. Hopcroft (1971): *An $n\log n$ algorithm for minimizing states in a finite automaton*. In Z. Kohavi & A. Paz, editors: Theory of Machines and Computations, Academic Press, pp. 189–196, doi:10.1016/b978-0-12-417750-5.50022-1.

[9] J. JáJá (1992): *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., USA.

[10] A. Jambulapati, Y.P. Liu & A. Sidford (2019): *Parallel reachability in almost linear work and square root depth*. In: Proc. of FOCS 2019, IEEE, pp. 1664–1686, doi:10.1109/FOCS.2019.00098.

[11] S. Khuller & U. Vishkin (1994): *On the parallel complexity of digraph reachability*. Information Processing Letters 52(5), pp. 239–241, doi:10.1016/0020-0190(94)00153-7.

[12] J.J.M. Martens, J.F. Groote, L.B. Haak, P. Hijma & A.J. Wijs (2022): *Linear parallel algorithms to compute strong and branching bisimilarity*. Software and Systems Modeling, pp. 1–25, doi:10.1007/s10270-022-01060-7.

[13] E.F. Moore (1956): *Gedanken-Experiments on Sequential Machines*. In Claude Shannon & John McCarthy, editors: Automata Studies, Princeton University Press, Princeton, NJ, pp. 129–153, doi:10.1515/9781400882618-006.

[14] R. Paige & R. E. Tarjan (1987): *Three partition refinement algorithms*. SIAM Journal on Computing 16(6), pp. 973–989, doi:10.1137/0216062.

[15] M.O. Rabin & D. Scott (1959): *Finite automata and their decision problems*. IBM Journal of Research and Development 3(2), pp. 114–125, doi:10.1147/rd.32.0114.

[16] B. Ravikumar & X. Xiong (1996): *A Parallel Algorithm for Minimization of Finite Automata*. In: Proceedings of the 10th International Parallel Processing Symposium, IPPS '96, IEEE Computer Society, USA, pp. 187–191, doi:10.1109/IPPS.1996.508056.

[17] L. Stockmeyer & U. Vishkin (1984): *Simulation of parallel random access machines by circuits*. SIAM Journal on Computing 13(2), pp. 409–422, doi:10.1137/0213027.

[18] A. Tewari, U. Srivastava & P. Gupta (2002): *A Parallel DFA Minimization Algorithm*. In: Proc. of HiPC, LNCS 2552, Springer, pp. 34–40, doi:10.1007/3-540-36265-7_4.

[19] B.A. Trakhtenbrot & J.M. Barzdin (1973): *Finite automata: behavior and synthesis*. North-Holland Publishing.

[20] J. Ullman & M. Yannakakis (1990): *High-Probability Parallel Transitive Closure Algorithms*. In: Proc. of SPAA 1990, pp. 200–209, doi:10.1145/97444.97686.

[21] A.J. Wijs (2015): *GPU Accelerated Strong and Branching Bisimilarity Checking*. In C. Baier & C. Tinelli, editors: *Proc. of TACAS*, *LNCS* 9035, Springer, pp. 368–383, doi:10.1007/978-3-662-46681-0_29.

Reachability and Safety Games under TSO Semantics

Stephan Spengler
Uppsala University
Uppsala, Sweden
stephan.spengler@it.uu.se

We consider games played on the transition graph of concurrent programs running under the Total Store Order (TSO) weak memory model. Games are frequently used to model the interaction between a system and its environment, in this case between the concurrent processes and the nondeterministic TSO buffer updates. In our formulation, the game is played by two players, who alternatingly make a move: The *process player* can execute any enabled instruction of the processes, while the *update player* takes care of updating the messages in the buffers that are between each process and the shared memory. We show that the reachability and safety problem of this game reduce to the analysis of single-process (non-concurrent) programs. In particular, they exhibit only finite-state behaviour. Because of this, we introduce different notions of *fairness*, which force the two players to behave in a more realistic way. Both the reachability and safety problem then become undecidable.

1 Introduction

In concurrent programs, different processes interact with each other through the use of shared memory. Programmers usually unconsciously assume that the semantics adhere to the Sequential Consistency (SC) memory model [17]. In SC, the execution of processes can be interleaved, but write instructions are visible in the memory in the exact order in which they were issued. However, most modern architectures, such as Intel x86 [16], SPARC [22], IBM's POWER [15], and ARM [6], implement several relaxations and optimisations that improve memory access latency but break SC assumptions. A standard model that is weaker than SC allows the reordering of reads and writes of the same process, as long as it maintains the appearance of SC from the perspective of each individual process. The implementation of this optimisation adds an unbounded first-in-first-out write buffer between each process and the shared memory. The buffer is used to delay write operations. This model is called Total Store Ordering (TSO) and is a faithful formalisation of SPARC and Intel x86 [19, 21].

Verification under TSO semantics is difficult due to the unboundedness of the buffers. Even if each process can be modelled as a finite-state system, the program itself has a state space of infinite size. The reachability problem for programs running under TSO semantics is to decide whether a target program state is reachable from a given initial state during program execution. If the target state is considered to be a bad state, it is also called the safety problem. Solving reachability and safety helps in deciding if a program is correct, i.e. if it adheres to a specification or if it can avoid states of undefined behaviour. Using alternative but equivalent semantics, it has been shown that the reachability problem is decidable [9, 2, 1]. Furthermore, lossy channel system [4, 14, 3, 20] can be simulated by programs running under TSO semantics [9]. This implies that the reachability problem is non-primitive recursive [20] and the repeated reachability problem is undecidable [3]. Additionally, the termination problem has been shown to be decidable [8] using the framework of well-structured transition systems [14, 4].

In this paper, we consider games played on the transition graph of concurrent programs running under TSO semantics. Formal games provide a framework to reason about the behaviour of a system and

A. Achilleos and A. Francalanza (Eds.): Fifteenth
International Symposium on Games, Automata, Logics,
and Formal Verification (GandALF 2024).
EPTCS 409, 2024, pp. 154–171, doi:10.4204/EPTCS.409.14

© S. Spengler

the interaction between the system and its environment. In particular, they have been extensively used in controller synthesis problems [7, 10, 12, 11, 5]. A previous paper introduces safety games in which two players alternatingly execute instructions of a concurrent program [24]. Motivated by this work, we propose a game setting that more closely models the interplay between a system and the environment: The first player controls the execution of the program instructions, while the second player handles the nondeterministic updates of the store buffers to the shared memory. This model sees the process and the update mechanism as antagonistic, and allows us to reason about the correctness of the program regardless of the update behaviour.

We consider two types of game objectives: In a reachability game, the process player tries to reach a given set of target states, while the update player tries to avoid this; In a safety game, these two roles are reversed. We show that in both cases finding the winner of the game reduces to the analysis of games being played on a program with just one process. Furthermore, we show that these games are bisimilar to finite-state games and thus decidable. In particular, the reachability and safety problem are PSPACE-complete. The reason that the concurrent programs exhibit a finite-state character lies in the optimal behaviour of the two players. If the player that controls the processes has a winning strategy, then she can win by playing in only one process, ignoring all the other processes of the program. On the other hand, if the player controlling the buffer is able to win, she can do so by never letting any write operation reach the memory. In both cases, there is no concurrency in the sense that the processes do not interact or communicate with each other. This is not realistic, since we should be able to assume that if the program runs a sufficiently long duration (1) every process will be executed and (2) every write stored in the buffer will be updated to the memory.

We rectify this issue by introducing two fairness conditions. First, in an infinite run the process player must execute each enabled process infinitely many times. Second, the update player must make sure that each write operation reaches the memory after finitely many steps. We show that both the reachability and safety problem become undecidable with these restrictions. To do so, we use a reduction from perfect channel systems adapted from [24].

Finally, we investigate an alternative TSO semantics in our game setting. The authors of [1] propose a load-buffer semantics for TSO which reverts the direction of the information flow between the processes and the shared memory. In their model, the buffer is filled with values from the memory which can later be read by the process. Using well-structured transition systems, they showed that it is equivalent to the classical TSO semantics with respect to state reachability. We explore whether the equivalence also holds in the two-player game, but come to the conclusion that this is not the case. In particular, we construct a concurrent program that is won by the update player under store-buffer semantics but by the process player under load-buffer semantics.

2 Preliminaries

Transition Systems A *(labelled) transition system* is a triple $\mathcal{T} = \langle \mathsf{C}, \mathsf{L}, \rightarrow \rangle$, where C is a set of *configurations*, L is a set of *labels*, and $\rightarrow \subseteq \mathsf{C} \times \mathsf{L} \times \mathsf{C}$ is a *transition relation*. We usually write $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_2$ if $\langle \mathsf{c}_1, \mathsf{label}, \mathsf{c}_2 \rangle \in \rightarrow$. Furthermore, we write $\mathsf{c}_1 \rightarrow \mathsf{c}_2$ if there exists some label such that $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_2$. A *run* π of $\mathcal{T}$ is a sequence of transitions $\mathsf{c}_0 \xrightarrow{\mathsf{label}_1} \mathsf{c}_1 \xrightarrow{\mathsf{label}_2} \mathsf{c}_2 \ldots \xrightarrow{\mathsf{label}_n} \mathsf{c}_n$. It is also written as $\mathsf{c}_0 \xrightarrow{\pi} \mathsf{c}_n$. A configuration c' is *reachable* from a configuration c, if there exists a run from c to c'.

For a configuration c, we define $\text{Pre}(\mathsf{c}) = \{\mathsf{c}' \mid \mathsf{c}' \rightarrow \mathsf{c}\}$ and $\text{Post}(\mathsf{c}) = \{\mathsf{c}' \mid \mathsf{c} \rightarrow \mathsf{c}'\}$. We extend these notions to sets of configurations C' with $\text{Pre}(\mathsf{C}') = \bigcup_{\mathsf{c} \in \mathsf{C}'} \text{Pre}(\mathsf{c})$ and $\text{Post}(\mathsf{C}') = \bigcup_{\mathsf{c} \in \mathsf{C}'} \text{Post}(\mathsf{c})$.

An *unlabelled transition system* is a transition system without labels. Formally, it is defined as a transition system with a singleton label set. In this case, we omit the labels.

Perfect Channel Systems Given a set of messages M, define the set of channel operations $\mathsf{Op} = \{!\mathsf{m}, ?\mathsf{m} \mid \mathsf{m} \in \mathsf{M}\} \cup \{\mathtt{skip}\}$. A *perfect channel system* (PCS) is a triple $\mathcal{L} = \langle \mathsf{S}, \mathsf{M}, \delta \rangle$, where S is a set of states, M is a set of messages, and $\delta \subseteq \mathsf{S} \times \mathsf{Op} \times \mathsf{S}$ is a transition relation. We write $\mathsf{s}_1 \xrightarrow{\mathsf{op}} \mathsf{s}_2$ if $\langle \mathsf{s}_1, \mathsf{op}, \mathsf{s}_2 \rangle \in \delta$.

Intuitively, a PCS models a finite state automaton that is augmented by a *perfect* (i.e. non-lossy) FIFO buffer, called *channel*. During a *send operation* $!\mathsf{m}$, the channel system appends m to the tail of the channel. A transition $?\mathsf{m}$ is called *receive operation*. It is only enabled if the channel is not empty and m is its oldest message. When the channel system performs this operation, it removes m from the head of the channel. Lastly, a skip operation just changes the state, but does not modify the buffer.

The formal semantics of $\mathcal{L}$ are defined by a transition system $\mathcal{T}_{\mathcal{L}} = \langle \mathsf{C}_{\mathcal{L}}, \mathsf{L}_{\mathcal{L}}, \rightarrow_{\mathcal{L}} \rangle$, where $\mathsf{C}_{\mathcal{L}} = \mathsf{S} \times \mathsf{M}^*$, $\mathsf{L}_{\mathcal{L}} = \mathsf{Op}$ and the transition relation $\rightarrow_{\mathcal{L}}$ is the smallest relation given by:

- If $\mathsf{s}_1 \xrightarrow{!\mathsf{m}} \mathsf{s}_2$ and $\mathsf{w} \in \mathsf{M}^*$, then $\langle \mathsf{s}_1, \mathsf{w} \rangle \xrightarrow{!\mathsf{m}}_{\mathcal{L}} \langle \mathsf{s}_2, \mathsf{m} \cdot \mathsf{w} \rangle$.
- If $\mathsf{s}_1 \xrightarrow{?\mathsf{m}} \mathsf{s}_2$ and $\mathsf{w} \in \mathsf{M}^*$, then $\langle \mathsf{s}_1, \mathsf{w} \cdot \mathsf{m} \rangle \xrightarrow{?\mathsf{m}}_{\mathcal{L}} \langle \mathsf{s}_2, \mathsf{w} \rangle$.
- If $\mathsf{s}_1 \xrightarrow{\mathtt{skip}} \mathsf{s}_2$ and $\mathsf{w} \in \mathsf{M}^*$, then $\langle \mathsf{s}_1, \mathsf{w} \rangle \xrightarrow{\mathtt{skip}}_{\mathcal{L}} \langle \mathsf{s}_2, \mathsf{w} \rangle$.

A state $\mathsf{s}_F \in \mathsf{S}$ is *reachable* from a configuration $\mathsf{c}_0 \in \mathsf{C}_{\mathcal{L}}$, if there exists a configuration $\mathsf{c}_F = \langle \mathsf{s}_F, \mathsf{w}_F \rangle$ such that c_F is reachable from c_0 in $\mathcal{T}_{\mathcal{L}}$. The **state reachability problem** of PCS is, given a perfect channel system $\mathcal{L}$, an initial configuration $\mathsf{c}_0 \in \mathsf{C}_{\mathcal{L}}$ and a final state $\mathsf{s}_F \in \mathsf{S}$, to decide whether s_F is reachable from c_0 in $\mathcal{T}_{\mathcal{L}}$. It is undecidable [13].

3 Concurrent Programs

Syntax Let Dom be a finite data domain and Vars be a finite set of shared variables over Dom. We define the *instruction set* $\mathsf{Instrs} = \{\mathtt{rd}(\mathsf{x}, \mathsf{d}), \mathtt{wr}(\mathsf{x}, \mathsf{d}) \mid \mathsf{x} \in \mathsf{Vars}, \mathsf{d} \in \mathsf{Dom}\} \cup \{\mathtt{skip}, \mathtt{mf}\}$, which are called *read*, *write*, *skip* and *memory fence*, respectively. A process is represented by a finite state labelled transition system. It is given as the triple $\mathsf{Proc} = \langle \mathsf{Q}, \mathsf{Instrs}, \delta \rangle$, where Q is a finite set of *local states* and $\delta \subseteq \mathsf{Q} \times \mathsf{Instrs} \times \mathsf{Q}$ is the transition relation. As with transition systems, we write $\mathsf{q}_1 \xrightarrow{\mathsf{instr}} \mathsf{q}_2$ if $\langle \mathsf{q}_1, \mathsf{instr}, \mathsf{q}_2 \rangle \in \delta$ and $\mathsf{q}_1 \rightarrow \mathsf{q}_2$ if there exists some instr such that $\mathsf{q}_1 \xrightarrow{\mathsf{instr}} \mathsf{q}_2$.

A *concurrent program* is a tuple of processes $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$, where $\mathcal{I}$ is a finite set of process identifiers. For each $\iota \in \mathcal{I}$ we have $\mathsf{Proc}^\iota = \langle \mathsf{Q}^\iota, \mathsf{Instrs}, \delta^\iota \rangle$. A *global* state of $\mathcal{P}$ is a function $\mathcal{S} : \mathcal{I} \rightarrow \bigcup_{\iota \in \mathcal{I}} \mathsf{Q}^\iota$ that maps each process to its local state, i.e $\mathcal{S}(\iota) \in \mathsf{Q}^\iota$.

TSO Semantics Under TSO semantics, the processes of a concurrent program do not interact with the shared memory directly, but indirectly through a FIFO *store buffer* instead. When performing a *write* instruction $\mathtt{wr}(\mathsf{x}, \mathsf{d})$, the process adds a new message $\langle \mathsf{x}, \mathsf{d} \rangle$ to the tail of its store buffer. A *read* instruction $\mathtt{rd}(\mathsf{x}, \mathsf{d})$ works differently depending on the current buffer content of the process. If the buffer contains a write message on variable x, the value d must correspond to the value of the most recent such message. Otherwise, the value is read directly from memory. A *skip* instruction only changes the local state of the process. The *memory fence* instruction is disabled, i.e. it cannot be executed, unless the buffer of the process is empty. Additionally, at any point during the execution, the process can *update* the write

read-own-write
$$\frac{\mathsf{q} \xrightarrow{\texttt{rd(x,d)}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q} \qquad \mathcal{B}(\iota)|_{\{\mathsf{x}\}\times\mathsf{Dom}}=\langle \mathsf{x},\mathsf{d}\rangle\cdot \mathsf{w}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\texttt{rd(x,d)}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow \mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

read-from-memory
$$\frac{\mathsf{q} \xrightarrow{\texttt{rd(x,d)}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q} \qquad \mathcal{B}(\iota)|_{\{\mathsf{x}\}\times\mathsf{Dom}}=\varepsilon \qquad \mathcal{M}(\mathsf{x})=\mathsf{d}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\texttt{rd(x,d)}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow \mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

write
$$\frac{\mathsf{q} \xrightarrow{\texttt{wr(x,d)}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\texttt{wr(x,d)}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow \mathsf{q}'],\mathcal{B}[\iota\leftarrow\langle \mathsf{x},\mathsf{d}\rangle\cdot\mathcal{B}(\iota)],\mathcal{M}\rangle}$$

skip
$$\frac{\mathsf{q} \xrightarrow{\texttt{skip}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q}}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\texttt{skip}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow \mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

memory-fence
$$\frac{\mathsf{q} \xrightarrow{\texttt{mf}} \mathsf{q}' \qquad \mathcal{S}(\iota)=\mathsf{q} \qquad \mathcal{B}(\iota)=\varepsilon}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\texttt{mf}_\iota}_{\mathcal{P}} \langle \mathcal{S}[\iota\leftarrow \mathsf{q}'],\mathcal{B},\mathcal{M}\rangle}$$

update
$$\frac{\mathcal{B}(\iota)=\mathsf{w}\cdot\langle \mathsf{x},\mathsf{d}\rangle}{\langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle \xrightarrow{\texttt{up}_\iota}_{\mathcal{P}} \langle \mathcal{S},\mathcal{B}[\iota\leftarrow \mathsf{w}],\mathcal{M}[\mathsf{x}\leftarrow \mathsf{d}]\rangle}$$

Figure 1: TSO semantics

message at the head of its buffer to the memory. For example, if the oldest message in the buffer is $\langle \mathsf{x},\mathsf{d}\rangle$, it will be removed from the buffer and the memory value of variable x will be updated to contain the value d. This happens in a nondeterministic manner.

Formally, we introduce a TSO *configuration* as a tuple $\mathsf{c} = \langle \mathcal{S},\mathcal{B},\mathcal{M}\rangle$, where:

- $\mathcal{S} : \mathcal{I} \to \bigcup_{\iota\in\mathcal{I}} \mathsf{Q}^\iota$ is a global state of $\mathcal{P}$.
- $\mathcal{B} : \mathcal{I} \to (\mathsf{Vars}\times\mathsf{Dom})^*$ represents the buffer state of each process.
- $\mathcal{M} : \mathsf{Vars} \to \mathsf{Dom}$ represents the memory state of each shared variable.

Given a configuration c, we write $\mathcal{S}(\mathsf{c})$, $\mathcal{B}(\mathsf{c})$ and $\mathcal{M}(\mathsf{c})$ for the global program state, buffer state and memory state of c. The semantics of a concurrent program running under TSO is defined by a transition system $\mathcal{T}_\mathcal{P} = \langle \mathsf{C}_\mathcal{P}, \mathsf{L}_\mathcal{P}, \to_\mathcal{P}\rangle$, where $\mathsf{C}_\mathcal{P}$ is the set of all possible TSO configurations and $\mathsf{L}_\mathcal{P} = \{\mathsf{instr}_\iota \mid \mathsf{instr}\in\mathsf{Instrs}, \iota\in\mathcal{I}\}\cup\{\mathsf{up}_\iota \mid \iota\in\mathcal{I}\}$ is the set of labels. The transition relation $\to_\mathcal{P}$ is given by the rules in Figure 1, where we use $\mathcal{B}(\iota)|_{\{\mathsf{x}\}\times\mathsf{Dom}}$ to denote the restriction of $\mathcal{B}(\iota)$ to write messages on the variable x. Furthermore, we define up^* to be the transitive closure of $\{\mathsf{up}_\iota \mid \iota\in\mathcal{I}\}$, i.e. $\mathsf{c}_1 \xrightarrow{\mathsf{up}^*}_\mathcal{P} \mathsf{c}_2$ if and only if c_2 can be obtained from c_1 by some amount of buffer updates.

A global state $\mathcal{S}_F$ is *reachable* from an initial configuration c_0, if there is a configuration c_F with $\mathcal{S}(\mathsf{c}_F) = \mathcal{S}_F$ such that c_F is reachable from c_0 in $\mathcal{T}_\mathcal{P}$. The **state reachability problem** of TSO is, given a program $\mathcal{P}$, an initial configuration c_0 and a final global state $\mathcal{S}_F$, to decide whether $\mathcal{S}_F$ is reachable from c_0 in $\mathcal{T}_\mathcal{P}$.

4 Games

A *game* is an unlabelled transition system, in which two players A and B take turns making a *move* in the transition system, i.e. changing the state of the game from one configuration to an adjacent one. In

a *reachability game*, the goal of player A is to reach a given set of target states, while player B tries to avoid this. In a *safety game*, the roles are swapped.

Formally, a game is defined as a tuple $\mathcal{G} = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow \rangle$, where C is the set of configurations, C_A and C_B form a partition of C, and $\rightarrow$ is a transition relation on C. For the games considered in this paper, the relation will always be restricted to $\rightarrow \subseteq (\mathsf{C}_A \times \mathsf{C}_B) \cup (\mathsf{C}_B \times \mathsf{C}_A)$, which means that the two players take turns alternatingly.

Plays and Winning Conditions An *infinite play* P of $\mathcal{G}$ is an infinite sequence $\mathsf{c}_0, \mathsf{c}_1, \ldots$ such that $\mathsf{c}_i \rightarrow \mathsf{c}_{i+1}$ for all $i \in \mathbb{N}$. Similarly, a *finite play* is a finite sequence $\mathsf{c}_0, \mathsf{c}_1, \ldots, \mathsf{c}_n$ such that $\mathsf{c}_i \rightarrow \mathsf{c}_{i+1}$ for all $i \in [0, \ldots, n-1]$ and $\mathrm{Post}(\mathsf{c}_n) = \emptyset$, i.e. the play ends in a deadlock. A *winning condition* W is a subset of all infinite plays. We say that player A is the winner of a play, if either the play is infinite and an element of W, or if it is finite and ends in a deadlock for player B, i.e. $\mathsf{c}_n \in \mathsf{C}_B$. Otherwise, player B wins the play.

In this work, we will consider two types of winning conditions. A *reachability condition* is given by a set $\mathsf{C}_R \subseteq \mathsf{C}$ which induces the winning condition $\mathsf{W}_R = \{\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \cdots \mid \exists i \in \mathbb{N} : \mathsf{c}_i \in \mathsf{C}_R\}$, i.e. the set of all plays that visit a configuration in C_R. Accordingly, a *safety condition* is given by a set $\mathsf{C}_S \subseteq \mathsf{C}$ which induces the winning condition $\mathsf{W}_S = \{\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \cdots \mid \forall i \in \mathbb{N} : \mathsf{c}_i \notin \mathsf{C}_S\}$, i.e. the set of all plays that never visit a configuration in C_S. Reachability games and safety games are dual to each other in the sense that a reachability game with winning condition C_R can be seen as a safety game with winning condition $\mathsf{C}_S = \mathsf{C} \setminus \mathsf{C}_R$, where the roles of players A and B are swapped.

Strategies A *strategy* of player A is a partial function $\sigma_A : \mathsf{C}^* \rightharpoonup \mathsf{C}_B$, such that $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n)$ is defined if and only if $\mathsf{c}_0, \ldots, \mathsf{c}_n$ is a prefix of a play, $\mathsf{c}_n \in \mathsf{C}_A$ and $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n) \in \mathrm{Post}(\mathsf{c}_n)$. A strategy σ_A is called *positional*, if it only depends on c_n, i.e. if $\sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_n) = \sigma_A(\mathsf{c}_n)$ for all $(\mathsf{c}_0, \ldots, \mathsf{c}_n)$ on which σ_A is defined. Thus, a positional strategy is usually given as a total function $\sigma_A : \mathsf{C}_A \rightarrow \mathsf{C}_B$. For player B, strategies are defined accordingly.

Two strategies σ_A and σ_B together with an initial configuration c_0 induce a finite or infinite play $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B) = \mathsf{c}_0, \mathsf{c}_1, \ldots$ such that $\mathsf{c}_{i+1} = \sigma_A(\mathsf{c}_0, \ldots, \mathsf{c}_i)$ for all $\mathsf{c}_i \in \mathsf{C}_A$ and $\mathsf{c}_{i+1} = \sigma_B(\mathsf{c}_0, \ldots, \mathsf{c}_i)$ for all $\mathsf{c}_i \in \mathsf{C}_B$. Given a winning condition W, a strategy σ_A is *winning* from a configuration c_0, if for *all* strategies σ_B it holds that player A wins the play $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$. That is, for each σ_B, either $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B) \in \mathsf{W}$ or the play is finite and ends in a deadlock of player B. A configuration c_0 is *winning* for player A if she has a strategy that is winning from c_0. Equivalent notions exist for player B. Given a reachability condition W_R / a safety condition W_S, the **reachability problem / safety problem** for a game $\mathcal{G}$ and a configuration c_0 is to decide whether c_0 is winning for player A.

Lemma 1 (Proposition 2.21 in [18])**.** *In reachability and safety games, every configuration is winning for exactly one player. A player with a winning strategy also has a positional winning strategy.*

Since we only consider reachability and safety games in this paper, all strategies will be positional.

5 Reachability and Safety Games under TSO Semantics

We model the execution of a TSO program as a game between two players: The *process player A* takes the role of the program and decides at each execution step which instruction to execute. The *update player B* is in charge of the buffer message updates in between.

Formally, a TSO program $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$ induces a game $\mathcal{G}(\mathcal{P}) = \langle \mathsf{C}, \mathsf{C}_A, \mathsf{C}_B, \rightarrow \rangle$ as follows. The sets C_A and C_B are copies of the set $\mathsf{C}^{\mathcal{P}}$ of TSO configurations, annotated by *A* and *B*, respectively:

$\mathsf{C}_A := \{\mathsf{c}_A \mid \mathsf{c} \in \mathsf{C}^{\mathcal{P}}\}$ and $\mathsf{C}_B := \{\mathsf{c}_B \mid \mathsf{c} \in \mathsf{C}^{\mathcal{P}}\}$. The transition relation $\rightarrow$ is defined by the following rules:

- **Program** For each transition $\mathsf{c} \xrightarrow{\mathsf{instr}_\iota}_{\mathcal{P}} \mathsf{c}'$ where $\mathsf{c}, \mathsf{c}' \in \mathsf{C}^{\mathcal{P}}$, $\iota \in \mathcal{I}$ and $\mathsf{instr} \in \mathsf{Instrs}$, it holds that $\mathsf{c}_A \xrightarrow{\mathsf{instr}_\iota} \mathsf{c}'_B$. This means that the process player can execute any program instruction.
- **Update** For each transition $\mathsf{c}_B \in \mathsf{C}_B$, it holds that $\mathsf{c}_B \xrightarrow{\mathsf{up}^*} \mathsf{c}'_A$ for all c' with $\mathsf{c} \xrightarrow{\mathsf{up}^*}_{\mathcal{P}} \mathsf{c}'$. This means that the update player can update any amount of buffer messages (including zero) between each of the turns of the process player.

In the remainder of this work, we will consider reachability or safety winning conditions induced by a set of local states $\mathsf{Q}^{\mathcal{P}}_{\mathsf{W}} \subset \mathsf{Q}^{\mathcal{P}}$. The corresponding set of configurations is $\mathsf{C}_{\mathsf{W}} = \{\mathsf{c} = \langle \mathcal{S}, \mathcal{B}, \mathcal{M} \rangle_X \mid X \in \{A, B\} \land \exists \iota : \mathcal{S}(\iota) \in \mathsf{Q}^{\mathcal{P}}_{\mathsf{W}}\}$, that is, the set of all configurations where at least one process is in a state of $\mathsf{Q}^{\mathcal{P}}_{\mathsf{W}}$. The set C_{W} can then induce either a reachability or safety winning condition. In the following, we will assume that the initial configuration (usually named c_0) is not contained in C_{W}, since otherwise the game is decided immediately. Furthermore, we desire that the process player immediately wins when reaching a target state in a reachability game, that is, we do not care whether the play can be extended infinitely or not. Formally, we require that in a reachability game, a process cannot deadlock from a target state, implying that the process player cannot lose after reaching it.

Games on Single-Process Programs This section introduces games on single-process programs which will help us analysing the general case. Given a game induced by a concurrent program $\mathcal{P}$, we compare it to the game on just one of the processes of $\mathcal{P}$. We show that if the process player wins the single-process game, then she also wins the original game. The main idea is that she achieves this by executing exactly the same instructions in both games.

For the remainder of this section, fix a program $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota \in \mathcal{I}}$ and a process index $\iota \in \mathcal{I}$. Let $\mathcal{P}^\iota = \langle \mathsf{Proc}^\iota \rangle$, i.e. the restriction of $\mathcal{P}$ to only the process Proc^ι. Define $\mathcal{G} = \mathcal{G}(\mathcal{P})$ and $\mathcal{G}^\iota = \mathcal{G}(\mathcal{P}^\iota)$, that is, the games induced by $\mathcal{P}$ and $\mathcal{P}^\iota$, respectively. Let Q_{W} induce a reachability or safety winning condition for $\mathcal{G}$ and define the winning condition for $\mathcal{G}^\iota$ through $\mathsf{Q}^\iota_{\mathsf{W}} = \mathsf{Q}_{\mathsf{W}} \cap \mathsf{Q}^\iota$.

Now, fix a configuration $\mathsf{c}_0 \in \mathsf{C} \setminus \mathsf{C}_{\mathsf{W}}$ with empty buffers (i.e. $\mathcal{B}(\mathsf{c}_0) = \langle \varepsilon \rangle_{\iota \in \mathcal{I}}$). For $X \in \{A, B\}$ and a configuration $\mathsf{c} = \langle \mathcal{S}, \mathcal{B}, \mathcal{M} \rangle_X \in \mathsf{C}_X$, let $\mathsf{c} \downarrow^\iota = \langle \mathcal{S}(\iota), \mathcal{B}(\iota), \mathcal{M} \rangle_X \in \mathsf{C}^\iota_X$, which can be understood as the projection of c onto the process Proc^ι. Conversely, for a configuration $\mathsf{c}^\iota \in \mathsf{C}^\iota_X$ of $\mathcal{G}^\iota$, define $\mathsf{c}^\iota \uparrow_{\mathcal{P}} = \langle \mathcal{S}(\mathsf{c}_0)[\iota \leftarrow \mathcal{S}(\mathsf{c}^\iota)], \mathcal{B}(\mathsf{c}_0)[\iota \leftarrow \mathcal{B}(\mathsf{c}^\iota)], \mathcal{M}(\mathsf{c}^\iota) \rangle_X \in \mathsf{C}_X$, that is, the configuration of $\mathcal{G}$ which is like c^ι for the process Proc^ι, but the local states and buffers of all other processes are as in the initial configuration c_0. Note that for all c^ι of $\mathcal{G}^\iota$, it holds that $(\mathsf{c}^\iota \uparrow_{\mathcal{P}}) \downarrow^\iota = \mathsf{c}^\iota$. On the other hand, $(\mathsf{c} \downarrow^\iota) \uparrow_{\mathcal{P}} = \mathsf{c}$ only holds for some c of $\mathcal{G}$.

Lemma 2. *If the process player wins $\mathcal{G}^\iota$ starting from the configuration $\mathsf{c}^\iota_0 = \mathsf{c}_0 \downarrow^\iota$, then she also wins $\mathcal{G}$ starting from c_0.*

Proof idea. Given a winning strategy σ^ι_A for the process player in $\mathcal{G}^\iota$, define a strategy in $\mathcal{G}$ by $\sigma_A(\mathsf{c}) = \sigma^\iota_A(\mathsf{c} \downarrow^\iota) \uparrow_{\mathcal{P}}$. We can show by induction over the number of moves that both strategies force the play to visit the same sequence of local states. Since winning the game is defined in terms of which local states are visited, it follows that σ_A must be a winning strategy. The full proof can be found in the extended version of this paper [23]. □

It is easy to see that the converse statement of Lemma 2 cannot hold for all $\iota \in \mathcal{I}$. Rather, we only show that under certain conditions the process player is able to visit the same local states of a process

Proc^1 in both $\mathcal{G}$ and $\mathcal{G}^\iota$. The strategy to do so will take a specific play in $\mathcal{G}$ and mimic all instructions that have been played in Proc^ι, similar as in the previous proof.

Fix $\iota \in \mathcal{I}$. Let σ_A be a winning strategy for the process player and let σ_B be the strategy of the update player where she never updates any buffer messages to the memory. Consider the play $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ in $\mathcal{G}$. For $k = 1, 2, \ldots$, let $\bar{\mathsf{c}}_k \rightarrow \mathsf{c}_k$ be the k-th time in this play where either the local state or the buffer of Proc^ι changes. This transition is due to some unique instruction $\mathcal{S}(\bar{\mathsf{c}}_k)(\iota) \xrightarrow{\mathsf{instr}_k} \mathcal{S}(\mathsf{c}_k)(\iota)$ in Proc^ι. In particular, it cannot be due to a memory update, since σ_B was chosen that way. Note that there does not necessarily need to be an infinite amount of k with this property. We define the strategy σ_A^ι as follows: Whenever $\mathcal{G}^\iota$ is in the k-th round, the local state of the process is $\mathcal{S}(\bar{\mathsf{c}}_k)(\iota)$ and instr_k is enabled, execute this instruction to move to the unique configuration with local state $\mathcal{S}(\mathsf{c}_k)(\iota)$. Otherwise, make an arbitrary move. Let σ_B^ι be an arbitrary strategy of the update player for $\mathcal{G}^\iota$. After the k-th round of $\mathsf{P}(\mathsf{c}_0^\iota, \sigma_A^\iota, \sigma_B^\iota)$, the game is in some position c_k^ι.

Claim 3. *For all $k \in \mathbb{N}$ for which c_k is defined, it holds that $\mathsf{c}_k \downarrow^\iota \xrightarrow{\mathsf{up}^*} \mathsf{c}_k^\iota$.*

Proof idea. First, we show by induction over k that instr_k is enabled at c_{k-1}^ι. Let $\tilde{\mathsf{c}}_k^\iota$ be such that $\mathsf{c}_{k-1}^\iota \xrightarrow{\mathsf{instr}_k} \tilde{\mathsf{c}}_k^\iota$. We compare the buffer and memory of c_k and $\tilde{\mathsf{c}}_k^\iota$ to conclude $\mathsf{c}_k \downarrow^\iota \xrightarrow{\mathsf{up}^*} \tilde{\mathsf{c}}_k^\iota$. The claim follows from $\tilde{\mathsf{c}}_k^\iota \xrightarrow{\mathsf{up}^*} \mathsf{c}_k^\iota$. The full proof is again in the extended version [23]. □

Concurrent Games We combine the results for single-process games to obtain the following theorem.

Theorem 4. *The process player wins $\mathcal{G}$ starting from a configuration c_0 if and only if she also wins $\mathcal{G}^\iota$ starting from configuration $\mathsf{c}_0 \downarrow^\iota$ for at least one $\iota \in \mathcal{I}$.*

Proof. By Lemma 2, if the process player wins $\mathcal{G}^\iota$ for at least one $\iota \in \mathcal{I}$, then she also wins $\mathcal{G}$. For the other direction, consider the strategies as defined above. What is left to show is that σ_A^ι is winning for at least one $\iota \in \mathcal{I}$.

If the process player has a reachability objective, $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ visits at least one target state in some process Proc^ι. Note again that the update player cannot be deadlocked and therefore the play must be infinite. From Claim 3 it follows that $\mathsf{P}(\mathsf{c}_0^\iota, \sigma_A^\iota, \sigma_B^\iota)$ visits the same local states of Proc^ι than $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ and in particular, it visits the same target state. Since σ_B^ι was chosen arbitrarily, it means that σ_A^ι is a winning strategy. Otherwise, if the process player has a safety objective, $\mathsf{P}(\mathsf{c}_0, \sigma_A, \sigma_B)$ executes an infinite amount of instructions in at least one process Proc^ι, but never visits any of its target states. Using the same arguments as previously, it follows that $\mathsf{P}(\mathsf{c}_0^\iota, \sigma_A^\iota, \sigma_B^\iota)$ is also a winning play and σ_A^ι is a winning strategy. □

Theorem 4 reduces the reachability problem for games on concurrent programs to the single-process case. Although the game $\mathcal{G}^\iota$ still has an infinite amount of configurations, many of them are indistinguishable in the sense that they have the same local state and allow the same sequences of instructions to be executed. Section 6 formally constructs a finite game that is a so-called *bisimulation* of $\mathcal{G}^\iota$. This shows that the reachability and safety problems for TSO games are decidable. In fact, they are PSPACE-complete.

Fairness Conditions In the previous section we have seen that the game played under TSO semantics reduces to the analysis of games on single-process programs. This is somewhat unsatisfying, since those games do not exhibit any concurrent behaviour that arises from the communication between multiple

processes. The underlying reason is the structure of the optimal strategies of the two players: If the process player wins, it is because she only plays in one single process. Otherwise, the update player wins by never updating any buffer messages. Both of these behaviours are not natural in the sense that they will not occur in any reasonable program environment: We should be able to assume that eventually (1) every buffer message will be updated to the memory and (2) every process will execute an instruction. In Section 7 and Section 8 we will impose additional restrictions on the two players to enforce this behaviour.

6 Decidability of Single-Process Games

TSO Views In the single-process program $\mathcal{P}^\iota$, there is no communication between different processes. A read operation of the process Proc^ι on a variable x either reads the initial value from the (shared) memory, or the value of the last write on x done by Proc^ι, if such a write operation has happened. In the latter case, the value of the read operation can come from either the buffer of Proc^ι or directly from the memory. But a single process cannot distinguish between these two cases. To be exact, the information that the process can obtain from the buffer and the memory is the value that Proc^ι can read from each variable, and whether the process can execute a memory fence instruction or not. Together with the local state of Proc^ι at the current configuration, this completely determines the enabled transitions for the process.

We call this concept the *view* of the process on the (concurrent) system and define it formally as a tuple $\mathsf{v} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F} \rangle$, where:

- $\mathcal{S} \in \mathsf{Q}^\iota$ is the local state of Proc^ι.
- $\mathcal{V} : \mathsf{Vars} \to \mathsf{Dom}$ defines which value Proc^ι reads from each variable.
- $\mathcal{F} \in \{\mathsf{true}, \mathsf{false}\}$ represents the possibility to perform a memory fence instruction.

Given a view $\mathsf{v} = \langle \mathcal{S}, \mathcal{V}, \mathcal{F} \rangle$, we write $\mathcal{S}(\mathsf{v})$, $\mathcal{V}(\mathsf{v})$ and $\mathcal{F}(\mathsf{v})$ for the local state $\mathcal{S}$, the value state $\mathcal{V}$ and the fence state $\mathcal{F}$ of v, respectively. The view of a configuration $\mathsf{c} \in \mathsf{C}^\iota$ is denoted by $\mathsf{v}(\mathsf{c})$ and defined in the following way. First, $\mathcal{S}(\mathsf{v}(\mathsf{c})) = \mathcal{S}(\mathsf{c})$. For all $\mathsf{x} \in \mathsf{Vars}$, if $\mathcal{B}(\mathsf{c})|_{\{\mathsf{x}\} \times \mathsf{Dom}} = \langle \mathsf{x}, \mathsf{d} \rangle \cdot \mathsf{w}$, then $\mathcal{V}(\mathsf{v}(\mathsf{c}))(\mathsf{x}) = \mathsf{d}$. Otherwise, $\mathcal{V}(\mathsf{v}(\mathsf{c}))(\mathsf{x}) = \mathcal{M}(\mathsf{c})(\mathsf{x})$. Lastly, $\mathcal{F}(\mathsf{v}(\mathsf{c})) = \mathsf{true}$ if and only if $\mathcal{B}(\mathsf{c}) = \varepsilon$.

For $\mathsf{c}_1, \mathsf{c}_2 \in \mathsf{C}$, if $\mathsf{v}(\mathsf{c}_1) = \mathsf{v}(\mathsf{c}_2)$, then we write $\mathsf{c}_1 \equiv \mathsf{c}_2$. In such a case, the process Proc^ι cannot differentiate between c_1 and c_2 in the sense that the enabled transitions in both configurations are the same. This is shown in Figure 2 and formally captured in Lemma 5.

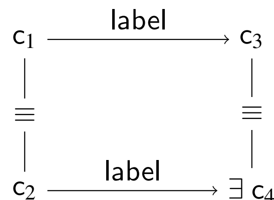


Figure 2: The configurations of Lemma 5.

Lemma 5. *For all* $\mathsf{c}_1, \mathsf{c}_3, \mathsf{c}_2 \in \mathsf{C}^\iota$ *and* $\mathsf{label} \in \mathsf{Instrs} \cup \{\mathtt{up}^*\}$ *with* $\mathsf{c}_1 \equiv \mathsf{c}_2$ *and* $\mathsf{c}_1 \xrightarrow{\mathsf{label}} \mathsf{c}_3$*, there exists a* $\mathsf{c}_4 \in \mathsf{C}^\iota$ *such that* $\mathsf{c}_3 \equiv \mathsf{c}_4$ *and* $\mathsf{c}_2 \xrightarrow{\mathsf{label}} \mathsf{c}_4$.

Proof. If $\mathsf{label} = \mathtt{up}^*$, this clearly holds for $c_4 = c_2$. Otherwise, we first show that $\mathsf{label} \in \mathsf{Instrs}$ is enabled at c_2. Since $c_1 \equiv c_2$, it holds that $\mathcal{S}(c_1) = \mathcal{S}(c_2)$. Furthermore, if $\mathsf{label} = \mathtt{rd}(x, d)$, then $\mathcal{V}(v(c_1))(x) = \mathcal{V}(v(c_2))(x) = d$. Also, if $\mathsf{label} = \mathtt{mf}$, then $\mathcal{B}(v(c_1)) = \varepsilon$ and since $\mathcal{F}(v(c_1)) = \mathcal{F}(v(c_2)) = \mathsf{true}$ it follows that $\mathcal{B}(v(c_2)) = \varepsilon$. From these considerations and the definition of the TSO semantics (see Figure 1), it follows that label is indeed enabled at c_2.

Let c_4 be the unique configuration obtained after executing the transition $\mathcal{S}(c_1) \xrightarrow{\mathsf{label}} \mathcal{S}(c_3)$ at c_2, i.e. $c_2 \xrightarrow{\mathsf{label}} c_4$ and $\mathcal{S}(c_4) = \mathcal{S}(c_3)$. If $\mathsf{label} = \mathtt{wr}(x, d)$, then $\mathcal{V}(v(c_4)) = \mathcal{V}(v(c_3)) = \mathcal{V}(v(c_1))[x \leftarrow d]$ and $\mathcal{F}(v(c_4)) = \mathcal{F}(v(c_3)) = \mathsf{false}$. Otherwise, $\mathcal{V}(v(c_4)) = \mathcal{V}(v(c_3)) = \mathcal{V}(v(c_1))$ and $\mathcal{F}(v(c_4)) = \mathcal{F}(v(c_3)) = \mathcal{F}(v(c_1))$. In all cases it follows that $c_3 \equiv c_4$. □

Bisimulations A *colouring* of a game $\mathcal{G}$ is a function $\lambda : C \to \mathcal{C}$ from the set of configurations into some set of colours $\mathcal{C}$. Consider two games $\mathcal{G}$ and $\mathcal{G}'$ with colouring functions λ and λ', respectively. A *bisimulation* (also called *zig-zag relation*) between $\mathcal{G}$ and $\mathcal{G}'$ is a relation $Z \subseteq C \times C'$ such that for all pair of related configurations $(c_1, c_2) \in Z$ it holds that:

- c_1 and c_2 agree on their colour: $\lambda(c_1) = \lambda'(c_2)$
- (*zig*) for each transition $c_1 \xrightarrow{\mathsf{label}} c_3$ there is a transition $c_2 \xrightarrow{\mathsf{label}} c_4$ such that $(c_3, c_4) \in Z$.
- (*zag*) for each transition $c_2 \xrightarrow{\mathsf{label}} c_4$ there is a transition $c_1 \xrightarrow{\mathsf{label}} c_3$ such that $(c_3, c_4) \in Z$.

We say that two related configurations c_1 and c_2 are *bisimilar* and write $c \approx c'$. We call $\mathcal{G}$ and $\mathcal{G}'$ *bisimilar* if there is a bisimulation between them.

It is common knowledge in game theory that winning strategies are preserved under bisimulations if the colourings are a refinement of the winning condition in the following sense. Consider two bisimilar games $\mathcal{G}$ and $\mathcal{G}'$ with winning conditions given by C_W and C'_W, respectively. Let λ be a colouring function for $\mathcal{G}$ such that the configurations in C_W have different colours than the rest of the configurations, i.e. $\lambda^{-1}(\lambda(C_W)) = C_W$. Define λ' as a colouring function for $\mathcal{G}'$ accordingly.

Lemma 6. *Given two bisimilar configurations $c_0 \in C$ and $c'_0 \in C'$, it holds that c_0 is a winning configuration in $\mathcal{G}$ if and only if c'_0 is a winning configuration in $\mathcal{G}'$.*

We define a game on views $\mathcal{G}^{\mathsf{V}} = \langle \mathsf{V}, \mathsf{V}_A, \mathsf{V}_B, \to_{\mathsf{v}} \rangle$ that is bisimilar to the single-process game $\mathcal{G}^\iota$. Let $\mathsf{V}_X = \{ v(c)_X \mid c \in C^\iota \}$ for $X \in \{A, B\}$ and $\mathsf{V} = \mathsf{V}_A \cup \mathsf{V}_B$. We extend the notation of the function v to game configurations by $v(c_X) = v(c)_X$ for $X \in \{A, B\}$ and $c \in C^\iota$. Now, we can define $\to_{\mathsf{v}}$ by $v(c) \xrightarrow{\mathsf{label}}_{\mathsf{v}} v(c')$ if and only if $c \xrightarrow{\mathsf{label}} c'$ for some $c, c' \in C^\iota$.

Theorem 7. *The relation $Z = \{ (c, v(c)) \mid c \in C^\iota \} \subset C^\iota \times C^{\mathsf{V}}$ is a bisimulation between $\mathcal{G}^\iota$ and $\mathcal{G}^{\mathsf{V}}$ with colouring functions $\lambda^\iota : C^\iota \to \mathsf{V}, c \mapsto v(c)$ and $\lambda^{\mathsf{V}} = \mathrm{id}_{\mathsf{V}}$, respectively.*

Proof. From the definition it follows directly that related configurations agree on their colour and that $\mathcal{G}^{\mathsf{V}}$ can simulate $\mathcal{G}^\iota$. What is left to show is that $\mathcal{G}^\iota$ can also simulate $\mathcal{G}^{\mathsf{V}}$, i.e. that for all $c \approx v$ and $v \xrightarrow{\mathsf{label}}_{\mathsf{v}} \tilde{v}$ there is $c \xrightarrow{\mathsf{label}} \tilde{c}$ with $\tilde{c} \approx \tilde{v}$. The transition $v \xrightarrow{\mathsf{label}}_{\mathsf{v}} \tilde{v}$ is due to a transition $d \xrightarrow{\mathsf{label}} \tilde{d}$ for some $d, \tilde{d} \in C^\iota$ with $d \approx v$ and $\tilde{d} \approx \tilde{v}$. Since $v(c) = v = v(d)$, it follows that $c \equiv d$. Apply Lemma 5 to $c_1 = d$, $c_2 = c$ and $c_3 = \tilde{d}$ to obtain a configuration $\tilde{c} = c_4$ with the desired properties. □

Since C^{V} is finite, it is rather evident that the reachability and safety problem are decidable, e.g. by applying a backward induction algorithm. In fact, both problems are PSPACE-complete. Intuitively, this makes sense since each variable can be seen as a single cell of a bounded Turing machine. The extended

version of this paper [23] gives a polynomial-space algorithm to show the upper complexity bound and constructs a reduction from TQBF for the lower bound. These results then immediately translate to the single-process TSO game $\mathcal{G}^{\iota}$, since it is bisimilar to $\mathcal{G}^{\vee}$.

7 Update Fairness in Reachability Games

In this section, we introduce *update fairness*, which we require the update player to satisfy. The core idea of update fairness is that eventually, each buffer message will be updated to the shared memory. This means that the process player can in some sense *wait* for the buffer messages to arrive in the memory. In safety games, delaying the run indefinitely favours the process player. Thus, we will focus on reachability games in this section.

We implement update fairness as follows. Whenever the program is in a configuration at which no program instruction is enabled (a deadlock), the system waits for the update player to update buffer messages to the memory, until the program exits the deadlock (or all buffers are empty). To simplify the formalisation, we will assume that the update player does so in her very next turn. This idea can be equivalently expressed by saying that if it is the process player's turn and the system is deadlocked, then it follows that all buffers must be empty.

Let $\mathcal{P} = \langle \mathsf{Proc}^{\iota} \rangle_{\iota \in \mathcal{I}}$ be a TSO program with induced game $\mathcal{G}(\mathcal{P})$. We define W_U as the set of all plays $\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \ldots$ in $\mathcal{G}(\mathcal{P})$ that satisfy update fairness:

$$\mathsf{P} \in \mathsf{W}_U \iff \forall\, k \in \mathbb{N}, \mathsf{c}_k \in \mathsf{C}_A : (\mathrm{Post}(\mathsf{c}_k) = \emptyset \implies \forall\, \iota \in \mathcal{I} : \mathcal{B}(\mathsf{c}_k)(\iota) = \varepsilon)$$

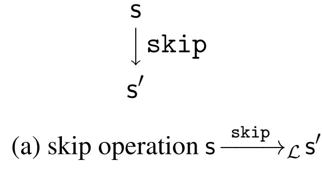
For a reachability condition W_R, let the set $\mathsf{W}_{RU} = \mathsf{W}_R \cup \overline{\mathsf{W}_U} = \{\mathsf{P} \mid \mathsf{P} \in \mathsf{W}_R \vee \mathsf{P} \notin \mathsf{W}_U\}$ be the set of winning plays for the process player, i.e. the set of all plays that either reach a target state or that do not admit update fairness. The remainder of this section will be dedicated to show that the reachability problem under update fairness is undecidable. We will achieve this by reducing the state reachability problem of perfect channel systems, which is undecidable, to the reachability problem of $\mathcal{G}(\mathcal{P})$ with respect to W_{RU}. The main ideas of the reduction are similar to those in [24].

Given a perfect channel system $\mathcal{L} = \langle \mathsf{S}, \mathsf{M}, \delta \rangle$, we construct a TSO program $\mathcal{P}$ that simulates $\mathcal{L}$. The process player will decide which transitions of the PCS to simulate, while the update player only takes care of the buffer updates. The program consists of two processes Proc^1 and Proc^2, which are shown in Figure 3 and Figure 4, respectively.

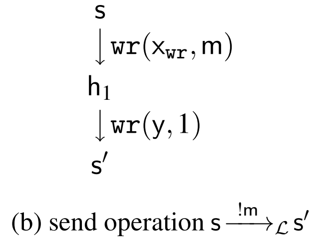
The first process keeps track of the configuration of the channel system and simulates the control flow. For each transition in $\mathcal{L}$, we construct a sequence of transitions in Proc^1 that simulates both the state change and the channel behaviour of the $\mathcal{L}$-transition. To achieve this, Proc^1 uses its buffer to store the messages of the PCS's channel. In particular, to simulate a send operation $!\mathsf{m}$, Proc^1 adds the message $\langle \mathsf{x}_{\mathtt{wr}}, \mathsf{m} \rangle$ to its buffer. For receive operations, Proc^1 cannot read its own oldest buffer message, since it is overshadowed by the more recent messages. Thus, the program uses the second process Proc^2 to read the message from memory and copies it to the variable $\mathsf{x}_{\mathtt{rd}}$, where it can be read by Proc^1. We call the combination of reading a message m from $\mathsf{x}_{\mathtt{wr}}$ and writing it to $\mathsf{x}_{\mathtt{rd}}$ the *rotation* of m.

While this is sufficient to simulate all behaviours of the PCS, it also allows for additional behaviour that is not captured by $\mathcal{L}$. More precisely, we need to ensure that each channel message is received *once and only once*. Equivalently, we need to prevent the *loss* and *duplication* of messages. This can happen due to multiple reasons.

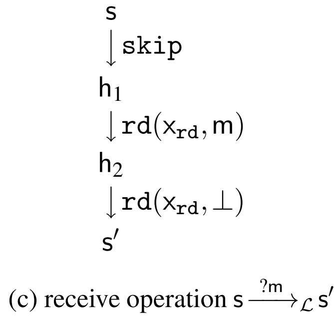
First, the update player might choose to lose a channel message by updating more than one message during a rotation. Consider an execution of $\mathcal{P}$ that simulates two send operations $!\mathsf{m}_1$ and $!\mathsf{m}_2$, i.e.



(a) skip operation $\mathsf{s} \xrightarrow{\mathtt{skip}}_{\mathcal{L}} \mathsf{s}'$



(b) send operation $\mathsf{s} \xrightarrow{!\mathsf{m}}_{\mathcal{L}} \mathsf{s}'$



(c) receive operation $\mathsf{s} \xrightarrow{?\mathsf{m}}_{\mathcal{L}} \mathsf{s}'$

Figure 3: Proc^1 of the reduction from PCS.

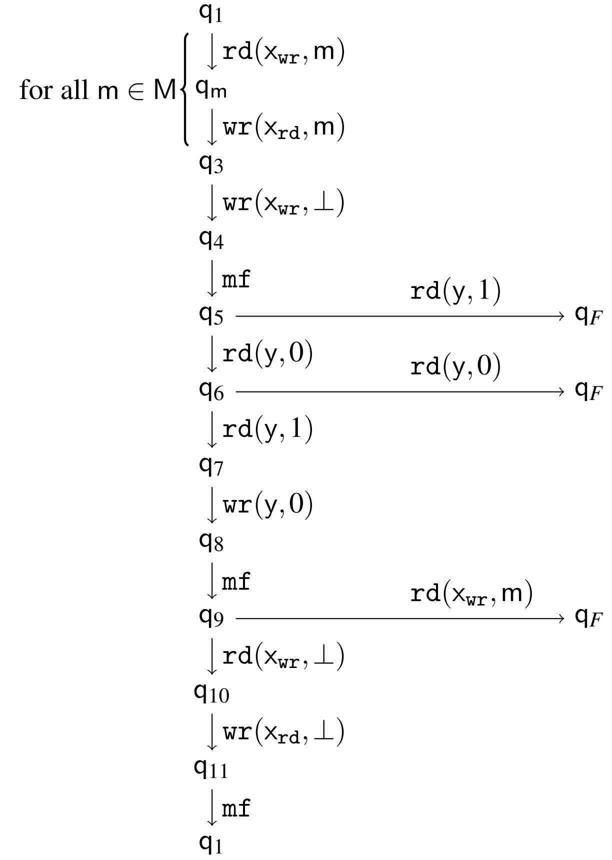


Figure 4: Proc^2 of the reduction from PCS.

Proc^1 adds $\langle \mathsf{x}_{\mathtt{wr}}, \mathsf{m}_1 \rangle$ and $\langle \mathsf{x}_{\mathtt{wr}}, \mathsf{m}_2 \rangle$ to its buffer. Now, if the process player wants to simulate a receive operation and initiates a message rotation, the update player can update both messages $\langle \mathsf{x}_{\mathtt{wr}}, \mathsf{m}_1 \rangle$ and $\langle \mathsf{x}_{\mathtt{wr}}, \mathsf{m}_2 \rangle$ to the memory before Proc^2 reads from $\mathsf{x}_{\mathtt{wr}}$. Thus, the first message m_1 is overwritten by the second message m_2 and is lost without ever being received. To prevent this, we implement a protocol that ensures that in each message rotation, exactly one channel message is being updated.

We extend the construction of Proc^1 such that it inserts an auxiliary message $\langle \mathsf{y}, 1 \rangle$ into its buffer after the simulation of each send operation. After a message rotation, that is, after Proc^2 copied a message from $\mathsf{x}_{\mathtt{wr}}$ to $\mathsf{x}_{\mathtt{rd}}$, the process then resets the value of $\mathsf{x}_{\mathtt{wr}}$ to its initial value $\perp$. Next, the process checks that y contains the value 0, which indicates that only one message was updated to the memory. Now, the update player is allowed to update exactly one $\langle \mathsf{y}, 1 \rangle$ buffer message, after which Proc^2 resets y to 0. To ensure that the update player has actually updated only one message in this step, Proc^2 then checks that $\mathsf{x}_{\mathtt{wr}}$ is still empty. If this protocol is violated at any point, Proc^2 enables the process player to immediately move to a winning state.

Although we have established that during each message rotation exactly one channel message will be rotated, we also need to ensure that for each rotation, Proc^1 will simulate exactly one receive operation. This is achieved by another protocol between Proc^1 and Proc^2, which gives the update player the tools to

enforce correct behaviour. To begin, the process player needs to initiate the simulation of a receive operation by moving to the first auxiliary state h_1 shown in Figure 3c. Only then is the program in a deadlock and the update player is forced to perform a message update. When reaching the first memory fence in Proc^2, the system is deadlocked again. Of course, the update player will not update the next message in the buffer of Proc^1, since it will lead to the process player immediately winning later on. Thus, she updates the message $\langle \mathsf{x}_{\mathtt{rd}}, \mathsf{m}\rangle$ to the memory, which enables both processes to continue. The next time that the update player is forced to update is when Proc^1 reaches the second auxiliary state h_2 and Proc^2 reaches the second memory fence. Only emptying the buffer of Proc^2 allows the program to continue. After three more instructions, Proc^2 will reach the third memory fence. Again, the update player needs to empty the buffer of Proc^2 which updates $\langle \mathsf{x}_{\mathtt{rd}}, \perp\rangle$ and enables Proc^1 to finish the simulation of the receive operation.

This concludes the mechanisms implemented to ensure that each channel message is received *once and only once*. We have constructed a TSO game with update fairness that simulates a perfect channel system. The winning condition of the game will be the reachability condition induced by the final states of the PCS together with the update fairness condition. We summarise our results in the following theorem.

Theorem 8. *The reachability problem for TSO games with update fairness is undecidable.*

8 Process Fairness in Safety Games

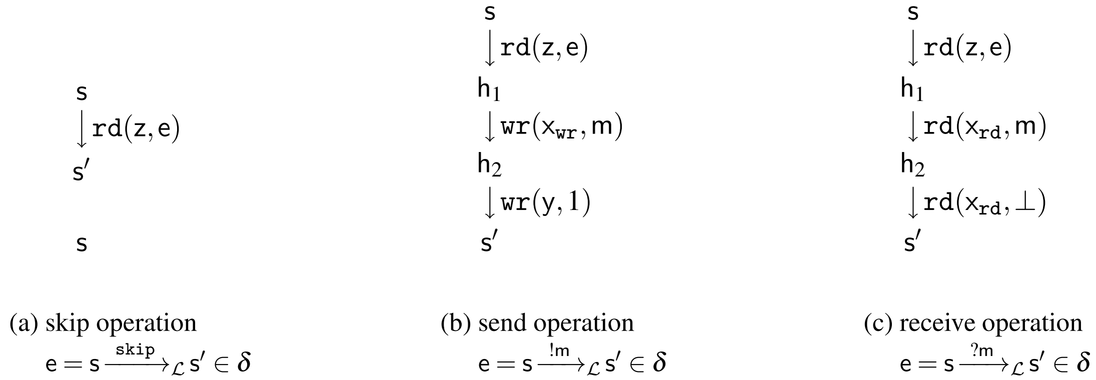
In the previous section, we have limited the behaviour of the update player. Now, we introduce *process fairness*, which will impact the capabilities of the process player. Process fairness means that for each process that is enabled infinitely many times during a run, the process player executes an instruction in that process infinitely often. In reachability games, this is no real restriction to the process player: If she can reach the set of winning states, then she can do so in finitely many moves. Thus, any finite prefix of a play that reaches a winning state can then trivially be extended to an infinite play that admits process fairness. Because of this, we will target our attention only towards safety games, where the process player cannot win in a finite amount of moves.

We formalise process fairness as follows. Let $\mathcal{P} = \langle \mathsf{Proc}^\iota \rangle_{\iota\in\mathcal{I}}$ be a TSO program with induced game $\mathcal{G}(\mathcal{P})$. We define W_P as the set of all plays $\mathsf{P} = \mathsf{c}_0, \mathsf{c}_1, \ldots$ in $\mathcal{G}(\mathcal{P})$ that satisfy process fairness:

$$\mathsf{P} \in \mathsf{W}_P \iff \forall\, \iota \in \mathcal{I} : \left(\exists^\infty\, k \in \mathbb{N}, \mathsf{c}_k \in \mathsf{C}_A : \mathsf{c}_k \xrightarrow{\mathsf{instr}_\iota} \mathsf{c}' \implies \exists^\infty\, k' \in \mathbb{N}, \mathsf{c}_{k'} \in \mathsf{C}_A : \mathsf{c}_{k'} \xrightarrow{\mathsf{instr}'_\iota} \mathsf{c}_{k'+1} \right)$$

Given a safety condition W_S, the intersection $\mathsf{W}_{SP} = \mathsf{W}_S \cap \mathsf{W}_P$ defines the set of winning plays that admit process fairness. In the remainder of this section we will show that the safety problem under process fairness is undecidable. To do so, we use a construction very similar to the one from the previous section to reduce the state reachability problem of perfect channel systems, to the safety problem of $\mathcal{G}(\mathcal{P})$. Before, it was the process player who decided which transition of the channel system to simulate. This time, it will be the update player who has this task.

Consider again a perfect channel system $\mathcal{L} = \langle \mathsf{S}, \mathsf{M}, \delta \rangle$. We modify the construction from the previous section. First, we introduce another shared variable z. Then, for each transition $\mathsf{e} \in \delta$ of the perfect channel system, we add an auxiliary process $\mathsf{Proc}^{\mathsf{e}}$. It consists of exactly one state q_e and one looping transition $\mathsf{q}_\mathsf{e} \xrightarrow{\mathtt{wr}(\mathsf{z},\mathsf{e})} \mathsf{q}_\mathsf{e}$. Furthermore, we prepend the gadget of process Proc^1 that simulates e with a transition $\mathtt{rd}(\mathsf{z}, \mathsf{e})$. The result of this is shown in Figure 5. Process Proc^2 is taken from the previous construction without any changes and can be found in Figure 4.



(a) skip operation $e = s \xrightarrow{\texttt{skip}}_{\mathcal{L}} s' \in \delta$

(b) send operation $e = s \xrightarrow{!m}_{\mathcal{L}} s' \in \delta$

(c) receive operation $e = s \xrightarrow{?m}_{\mathcal{L}} s' \in \delta$

Figure 5: Proc^1 of the reduction from PCS to a TSO game with process fairness.

The main idea of these modifications is that the update player can use the variable z to control which channel operation will be simulated. At the start of the run, both z and $x_{\texttt{wr}}$ contain the initial value $\perp$, which means that neither Proc^1 nor Proc^2 are enabled. Thus, the process player needs to begin playing in some process Proc^e, writing the message $\langle z, e\rangle$ to its buffer. This will continue until the update player decides to update one of these messages. But, due to process fairness, the process player is forced to eventually play in *all* enabled processes. In particular, she has to do so infinitely many times during any infinite play. This means that the update player can simply wait until each transition $e \in \delta$ was sufficiently many times added to the buffer of Proc^e to simulate a run of the PCS that reaches a final state. At that point, the update player starts updating the messages $\langle z, e\rangle$ one by one, each time waiting until the process player has finished simulating the unique operation that is enabled in Proc^1.

In more detail, to simulate the execution of a channel operation $e \in \delta$, the update player updates the buffer message $\langle z, e\rangle$ to the memory. Due to process fairness, we know that the process player eventually has to play in Proc^1, since it is now enabled. She takes the transition $\texttt{rd}(z,e)$ and then proceeds (although not necessarily immediately) with simulating e as was presented for the reachability case in the previous section. If e is a receive operation, the update player has to update a $\langle x_{\texttt{wr}}, m\rangle$ buffer message at some point to enable Proc^2 and start a message rotation. Again, process fairness forces the process player to eventually finish the rotation protocol. In any case, the simulation of the channel operation is guaranteed to terminate after finitely many steps. Now, the update player starts the next simulation by updating the corresponding buffer message.

Also in this construction we need to ensure that we do not introduce any behaviour that does not correspond exactly to what the PCS can do. Message loss due to updating two messages without rotation in between is handled in the same way as previously, using the auxiliary variable y. The same goes for message duplication, which is covered by the protocol between Proc^1 and Proc^2. What is left is message loss due to performing two rotations without simulating a receive operation. In the previous section, this could only happen if the process player decides to do so, since she was the one controlling the simulation. This is still prevented by the aforementioned protocol: The update player is not forced to let Proc^2 proceed beyond the second and third memory fences before Proc^1 keeps up with the protocol. But in this construction, the roles and capabilities of the two players have changed slightly and allow for additional behaviour: Without enabling a receive operation in Proc^1, the update player could update a message $\langle x_{\texttt{wr}}, m\rangle$, which enables Proc^2 instead. Due to process fairness, the process player would eventually have to perform a full message rotation.

We prevent this by adding for every channel message m a transition $\mathsf{q} \xrightarrow{\mathtt{rd}(\mathsf{x}_{\mathtt{rd}},\mathsf{m})} \mathsf{q}_F$ to *every* state q of Proc^1 *except* the states h_1 and h_2 (cf. Figure 5c) of the receive operations of message m. Here, q_F is a sink state which is safe for the process player and blocks the update player from winning the game. The idea of this transition is that during a message rotation, the value of $\mathsf{x}_{\mathtt{rd}}$ in the memory is m. Thus, the update player is not immediately losing only if Proc^1 is currently in one of the intermediate states of a receive operation. In particular, the process player can now move to h_2. Then, after the rotation has finished with the third memory fence, the variable $\mathsf{x}_{\mathtt{rd}}$ contains the value $\perp$ again, which means that Proc^1 is enabled and the process player can finish the simulation of the receive operation. We conclude that she can prevent the update player from performing a rotation without simulating a receive operation.

In summary, we have shown again that each channel message is read once and only once. The winning condition of the TSO game is given by the safety condition induced by the final states of the PCS together with the process fairness condition. This gives rise to the following theorem.

Theorem 9. *The safety problem for TSO games with process fairness is undecidable.*

9 Load Buffer Semantics in TSO Games

In [1], the authors introduced an alternative semantics for TSO, called *load buffer semantics*. It is equivalent to the traditional *store buffer semantics* in the sense that a global state $\mathcal{S}$ of the system is reachable under load-buffer semantics if and only if it is reachable under store buffer semantics. The alternative semantics have been proven to be useful in efficiently performing algorithmic verification or presenting simpler decidability proofs of safety properties. A natural question in the context of this paper is to ask whether these results transfer to the game setting. In particular, we want to know if a game is won by the same player when played under both semantics. Unfortunately, it turns out that this is not the case.

Load Buffer Semantics Under the new semantics, the store buffer between each process and the shared memory is replaced by a load buffer instead. This means that the information flow reverses its direction: Instead of write operations, the buffer now contains potential *read* operations that *might* be performed by the process. Each buffer message is either a pair $\langle \mathsf{x}, \mathsf{d} \rangle$ or a triple $\langle \mathsf{x}, \mathsf{d}, \mathsf{own} \rangle$, where the latter is called an *own-message*.

At any point during the run, the system can nondeterministically choose a variable x and its corresponding value d from the memory and add a message $\langle \mathsf{x}, \mathsf{d} \rangle$ to the tail of the buffer of one of the processes. This is called *read propagation* and speculates on a future read operation on x. Conversely, a *delete* operation removes the oldest message at the head of the buffer of some process and is also performed nondeterministically at any time.

A *write* instruction $\mathtt{wr}(\mathsf{x},\mathsf{d})$ of a process Proc immediately updates the value d of the variable x in the memory. Then, it adds the own-message $\langle \mathsf{x}, \mathsf{d}, \mathsf{own} \rangle$ to the buffer of Proc. The behaviour of a *read* instruction $\mathtt{rd}(\mathsf{x},\mathsf{d})$ depends on the contents of the buffer. If there is an own-message on the variable x, then the most recent one must correspond to the value d. Otherwise, if there is no such message, the head of the buffer must be a message $\langle \mathsf{x}, \mathsf{d} \rangle$. If this is not the case, the read instruction is disabled. The last two instructions, which are *skip* and *memory fence*, work exactly as in the classical TSO semantics: They only change the local state but not the memory or buffer, and the fence is only enabled if its buffer is empty.

For a formal definition of the semantics and the configurations of the induced transition system, we refer to [1].

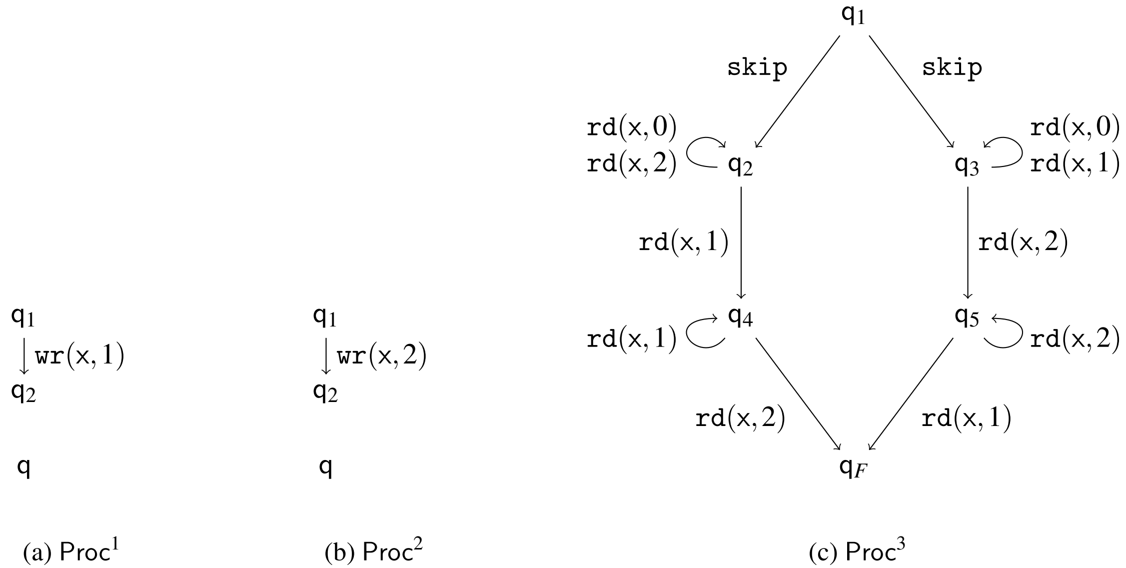


Figure 6: A concurrent program consisting of three processes.

Games Comparing the alternative to the classical TSO semantics, we see that the order in which the variables in the memory are updated, now depends directly on the order of execution of the corresponding write instructions, and not on the update order of the buffer messages. Conversely, the buffer does not delay the time when a write operation arrives at the memory, but instead delays when the change in the memory is visible to each process. Intuitively, we can already guess that the two semantics differ in the game setting, since the information available to the two players during the execution is different in both cases.

In the plain TSO game without fairness, there is actually no change. Since both reachability and safety games degenerate to single-process games with no communication between processes, the exact update semantics do not matter.

This is not the case if we add fairness conditions. Consider the safety game with process fairness played on the program shown in Figure 6. The target state is q_F in Proc^3 and the initial value of x is 0. Since the process player cannot be deadlocked in any other state of Proc^3, the only way for the update player to win is to force the play into q_F. In our classical game setting under store buffer semantics, the update player is able to achieve this. We describe a winning strategy for her.

Due to process fairness, the process player needs to eventually play in all three processes. The update player waits until processes Proc^1 and Proc^2 are in their respective q_2, and Proc_3 is in either q_2 or q_3. In the first case, if Proc_3 is in state q_2, she updates the buffer message of Proc^1, which writes the value 1 to the memory for variable x. The only enabled instruction for the process player is to move from q_2 to q_4 in Proc^3. Now, the update player updates the message from the buffer of Proc^2, which forces the process player to move to the target state and lose. In the other case, the update player performs the two update operations in the reverse order, which again forces the process player to enter the target state after two moves.

Next, consider the same game but under load buffer semantics. We have not yet formally defined how they should work, but this is not necessary for our argument. Assume that the process player as usual controls the program instructions and the update player in some way controls the nondeterministic

buffer behaviour. We will outline how the process player wins this game.

First, she plays in Proc^1, then in Proc^2. At this point, the value of x in the memory is 2, but the buffer of Proc^3 might already contain messages of the form $\langle \mathsf{x}, 1\rangle$ and $\langle \mathsf{x}, 2\rangle$. Note that it is only possible to have them in this exact order, i.e. it cannot be that there is some message $\langle \mathsf{x}, 2\rangle$ that is older than another message $\langle \mathsf{x}, 1\rangle$. Furthermore, since the program has no other reachable write instructions, any message that will be added in the future must be $\langle \mathsf{x}, 2\rangle$. Now, the process player plays in Proc^3 and moves to q_3. The update player needs the process player to eventually move to q_5, which means she has to enable the instruction $\mathtt{rd}(\mathsf{x}, 2)$. To do so, she deletes messages at the head of the buffer of Proc_3 until it reaches a message $\langle \mathsf{x}, 2\rangle$. But due to the order of the messages in the buffer, this means that it lost all messages $\langle \mathsf{x}, 1\rangle$ and also, as said previously, cannot add any more of them. It follows that the process can never execute the next instruction $\mathtt{rd}(\mathsf{x}, 1)$ and is thus stuck in q_5. Since this is not a deadlock for the process player, it results in a winning play for her.

We conclude that TSO safety games with process fairness do not have the same winning configurations under store buffer semantics and load buffer semantics, respectively. The same can be shown for reachability games with update fairness. Since it does not yield any additional insights, we do not present the argument here.

10 Conclusion and Future Work

In this paper, we continue the work on two-player games played on programs running under TSO semantics. We present a game model where one player controls the instructions of the program and the other player controls the buffer updates. Our results show that both the reachability problem and the safety problem for these games reduce to the analysis of games on single-process programs. Moreover, we show a bisimilarity to a game with a finite amount of configurations and use it to prove that the problems are in fact PSPACE-complete.

The reduced complexity comes from the optimal behaviour of the two players. The process player can always win by playing in only one single process, while the best strategy of the update player is to stay passive and not perform any buffer updates. We rectify this by introducing fairness conditions for both players. In reachability games, the update player is required to update each message eventually. This allows the process player to wait for a write instruction to arrive in the memory. In safety games, during an infinite run the process player has to perform instructions in all enabled processes infinitely often. Both restrictions lead to the respective problems being undecidable.

Finally, we connect the game model to the alternative load buffer semantics of TSO. We show that the equivalence between load buffer and store buffer that exists for classical TSO reachability does not carry over to the game setting.

This work analyses the basic winning conditions reachability and safety. Future work may expand the focus to more expressive winning conditions, like Büchi (i.e. repeated reachability), Co-Büchi, Parity, Rabin, Streett or Muller. Another way to expand is to look at other fairness conditions for the two players, for example transition fairness. These two directions of research are not orthogonal to each other since Muller or even Streett conditions might be able to encode some forms of fairness conditions.

References

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Ahmed Bouajjani & Tuan Phong Ngo (2018): *A Load-Buffer Semantics for Total Store Ordering*. Log. Methods Comput. Sci. 14(1), doi:10.23638/LMCS-14(1:9)2018.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson & Ahmed Rezine (2012): *Counter-Example Guided Fence Insertion under TSO*. In Cormac Flanagan & Barbara König, editors: Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings, Lecture Notes in Computer Science 7214, Springer, pp. 204–219, doi:10.1007/978-3-642-28756-5_15.

[3] Parosh Aziz Abdulla & Bengt Jonsson (1994): *Undecidable Verification Problems for Programs with Unreliable Channels*. In Serge Abiteboul & Eli Shamir, editors: Automata, Languages and Programming, 21st International Colloquium, ICALP94, Jerusalem, Israel, July 11-14, 1994, Proceedings, Lecture Notes in Computer Science 820, Springer, pp. 316–327, doi:10.1007/3-540-58201-0_78.

[4] Parosh Aziz Abdulla & Bengt Jonsson (1996): *Verifying Programs with Unreliable Channels*. Inf. Comput. 127(2), pp. 91–101, doi:10.1006/inco.1996.0053.

[5] Renato Acampora, Luca Geatti, Nicola Gigante, Angelo Montanari & Valentino Picotti (2022): *Controller Synthesis for Timeline-based Games*. In Pierre Ganty & Dario Della Monica, editors: Proceedings of the 13th International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2022, Madrid, Spain, September 21-23, 2022, EPTCS 370, pp. 131–146, doi:10.4204/EPTCS.370.9.

[6] ARM (2014): *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. Available at https://developer.arm.com/documentation/ddi0406/latest/.

[7] André Arnold, Aymeric Vincent & Igor Walukiewicz (2003): *Games for synthesis of controllers with partial observation*. Theor. Comput. Sci. 303(1), pp. 7–34, doi:10.1016/S0304-3975(02)00442-5.

[8] Mohamed Faouzi Atig (2020): *What is decidable under the TSO memory model?* ACM SIGLOG News 7(4), pp. 4–19, doi:10.1145/3458593.3458595.

[9] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt & Madanlal Musuvathi (2010): *On the verification problem for weak memory models*. In Manuel V. Hermenegildo & Jens Palsberg, editors: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010, ACM, pp. 7–18, doi:10.1145/1706299.1706303.

[10] Ralph-Johan Back & Cristina Cerschi Seceleanu (2004): *Contracts and Games in Controller Synthesis for Discrete Systems*. In: 11th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2004), 24-27 May 2004, Brno, Czech Republic, IEEE Computer Society, pp. 307–315, doi:10.1109/ECBS.2004.1316713.

[11] Ayca Balkan, Moshe Y. Vardi & Paulo Tabuada (2015): *Controller Synthesis for Mode-Target Games*. In Magnus Egerstedt & Yorai Wardi, editors: 5th IFAC Conference on Analysis and Design of Hybrid Systems, ADHS 2015, Atlanta, GA, USA, October 14-16, 2015, IFAC-PapersOnLine 48, Elsevier, pp. 343–350, doi:10.1016/J.IFACOL.2015.11.198.

[12] Nicolas Basset, Marta Z. Kwiatkowska & Clemens Wiltsche (2014): *Compositional Controller Synthesis for Stochastic Games*. In Paolo Baldan & Daniele Gorla, editors: CONCUR 2014 - Concurrency Theory - 25th International Conference, CONCUR 2014, Rome, Italy, September 2-5, 2014. Proceedings, Lecture Notes in Computer Science 8704, Springer, pp. 173–187, doi:10.1007/978-3-662-44584-6_13.

[13] Daniel Brand & Pitro Zafiropulo (1983): *On Communicating Finite-State Machines*. J. ACM 30(2), pp. 323–342, doi:10.1145/322374.322380.

[14] Alain Finkel & Philippe Schnoebelen (2001): *Well-structured transition systems everywhere!* Theor. Comput. Sci. 256(1-2), pp. 63–92, doi:10.1016/S0304-3975(00)00102-X.

[15] IBM (2021): *Power ISA, Version 3.1b*. Available at https://files.openpower.foundation/s/dAYSdGzTfW4j2r2/download/OPF_PowerISA_v3.1B.pdf.

[16] Intel Corporation (2012): *Intel 64 and IA-32 Architectures Software Developers Manual*. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[17] Leslie Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. IEEE Trans. Computers 28(9), pp. 690–691, doi:10.1109/TC.1979.1675439.

[18] René Mazala (2001): *Infinite Games*. In Erich Grädel, Wolfgang Thomas & Thomas Wilke, editors: Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001], Lecture Notes in Computer Science 2500, Springer, pp. 23–42, doi:10.1007/3-540-36387-4_2.

[19] Scott Owens, Susmit Sarkar & Peter Sewell (2009): *A Better x86 Memory Model: x86-TSO*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings, Lecture Notes in Computer Science 5674, Springer, pp. 391–407, doi:10.1007/978-3-642-03359-9_27.

[20] Philippe Schnoebelen (2002): *Verifying lossy channel systems has nonprimitive recursive complexity*. Inf. Process. Lett. 83(5), pp. 251–261, doi:10.1016/S0020-0190(01)00337-4.

[21] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli & Magnus O. Myreen (2010): *x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors*. Commun. ACM 53(7), pp. 89–97, doi:10.1145/1785414.1785443.

[22] SPARC International, Inc. (1994): *SPARC Architecture Manual Version 9*. Available at https://sparc.org/wp-content/uploads/2014/01/SPARCV9.pdf.gz.

[23] Stephan Spengler (2024): *Reachability and Safety Games under TSO Semantics (Extended Version)*. arXiv:2405.20804.

[24] Stephan Spengler & Sanchari Sil (2023): *TSO Games - On the decidability of safety games under the total store order semantics*. In Antonis Achilleos & Dario Della Monica, editors: Proceedings of the Fourteenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2023, Udine, Italy, 18-20th September 2023, EPTCS 390, pp. 82–98, doi:10.4204/EPTCS.390.6.